E28/CS82: Mobile Robotics, Spring 2005

Dr. Bruce A. Maxwell, Associate Professor
Department of Engineering
Swarthmore College

Course Description

This course addresses the problems of controlling and motivating mobile robots to act intelligently in dynamic, unpredictable environments. Major topics will include robot perception using vision and sonar, kinematics and inverse kinematics, navigation and control, mapping, and motivating robots to act. Labs will focus on programming robots to execute tasks, explore, and interact with their environment.

Prerequisites: ENGR 15 or CPSC 35 or permission of instructor

E28/CS82 Lecture #1 S05

Overall Topic: Introduction, What is a robot?

Administrivia

- Course web page: http://www.palantir.swarthmore.edu/
- Syllabus, homework, and all labs will be on the web page.
  - Homework #1 is already there
- Textbooks & handouts
- Exam dates: Feb 18, April 7 (1 hour exams in-class)
- Grading: HW 10%, Labs 40%, Final project & participation 10%, Exams I & II 10% each, Final 20%
- Weekly homework, late homework receives no credit
- 5 labs over 14 weeks on a 2-3-week schedule, divide into groups of 3
  - Nomad Scout simulator for testing code
  - Scouts and Magellans for running robots around
- First lab meeting is Wednesday at 1:30 in the Mural Room
  - We have the Mural Room booked from 1:30-midnight on Wednesdays
- If you do the required work on a lab, and do it well, your group will receive a B+. An A of some kind requires extension of the lab. I will always suggest several extensions, but you are free to pursue your own.
- Academic honesty
  - I encourage sharing between lab groups. It is part of robotics culture.
  - Cite or notate anything that is shared work. Don't surprise me.

Introductions

1. What was your first memory of a robot
2. What would you like robots to be able to do for you?
3. Your name, major
4. Food and color

Competitions & Opportunties

Robot event at AAAI

- Scavenger Hunt
- Open Interaction
- Robot Challenge
- Conference is early, July 9-13, in Pittsburgh (travel is easy)
- If you have a working system at the end of the semeter, your group may go

Research opportunties

- 1 position for this summer working on the USR project, possibly two
- Decision time will be mid-February to early March

Introduction to Robotics

What is a robot?

What is a robot, (according to Rod Brooks)?

- Situated
  - Has to live in a rich sensory environment
  - Has to be able to sense the richness
- Enabled
  - A robot has to be able to modify its environment
  - If a robot cannot modify its environment, it is only an observer
- Motivated
  - A robot has to be motivated to interact and modify its environment
  - A robot without motivation is just a power tool

Robot Configurations and Designs

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

Most mobile robots use either multi-legged systems or powered wheels

Key issues in locomotion

- Stability
  - Number and geometry of contact points
  - Center of gravity
  - static/dynamic stability
  - inclination of terrain
- Characteristics of contact
  - contact point/path size and shape
  - angle of contact
  - friction
- Type of environment
  - structure
  - medium

Legged Robots

Obstacle, what obstacle?

Advantage is that obstacles to rolling motion are not obstacles to legged motion

- Ground clearance is higher
- Local gradient of the terrain is irrelevant if the feet can find purchase
- Can manipulate the environment with flexible legs

Disadvantage is power, weight, and mechanical complexity

- Legs need several degrees of freedom
- Legs must be strong enough to support part of the robot's weight
- Maneuverability requires many legs to produce forces in the required direction (spiders)

E28/CS82 Lecture #2 S05

Overall Topic: Robot Configurations

Legged Robots

Legged configurations

- Six legs plus: statically stable walking is possible
- Four legs: dynamically stable walking is possible, requires more active control system
- Two legs: dynamically stable walking is possible, requires sophisticated control system
- One leg: unstable configuration statically, requires most sophisticated control system

The number of possible events for a walking machine is

$$N = (2k-1)!$$

This is why learning gaits for complex legged robots is a hard problem.

- A 6 legged robot has over 39 million possible choices at any given time step

Wheeled Robots

Advantage is low power and low complexity

- Variety of configurations
- Efficient designs
- Good maneuverability

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

- Something to keep the wheels in contact with the ground at all times

Wheel geometries (Table 2.1)

- 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 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

Robot Kinematics

Matrices as representations of geometric transformations

- Translations
- Rotation

$$\text{2-D transformations: } R_Z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad T(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$\text{3-D Versions } R_Z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

With rigid bodies, we can describe all motion as a combination of translation and rotation.

- Pick a point P on the robot to be the robot's origin
- Pick an orientation for the robot that sets the local X_r and Y_r axes.
- Rotation sets the robot's orientation relative to a global coordinate frame
- Translation sets its (x, y) position relative to a global coordinate frame

Generally the robot's initial position sets the global coordinate frame and motion is relative to it

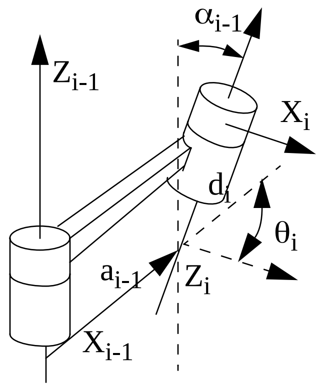
We can parameterize this as a single vector $\alpha = \begin{bmatrix} x & y & \theta \end{bmatrix}^t$

E28/CS82 Lecture #3 S05

Overall Topic: Robot Kinematics

Robot Kinematics

Legs



Legs are rotational or linear joints connected together

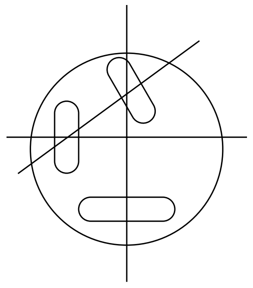
- Same math as a robot arm
- Transformation from link i to link i-1, use (1)

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

- Maneuverability defined by the ability of the robot to apply a force in a particular direction
- Jacobian tells us how the end effector (foot) moves instantaneously in response to changes in the joint angles.
    - First calculate the matrix expressing the end effector in global space
    - Find each degree of freedom expressed as a function of the joint angles (rows)
    - Find partial derivative of each expression with respect to each joint angle (columns)
- Inverse of the Jacobian tells us the joint angle motions needed to execute a spatial motion

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{z}\end{bmatrix} = J(\Theta)\begin{bmatrix}\dot{\theta}_1\\ \dots\\ \dot{\theta}_N\end{bmatrix} \qquad \begin{bmatrix}\dot{\theta}_1\\ \dots\\ \dot{\theta}_N\end{bmatrix} = J^{-1}(\Theta)\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{z}\end{bmatrix} \tag{2}$$

Wheels



Each contact describes a potential kinematic constraint on the robot's motion

- A planar wheeled robot has three independent parameters: (x, y, θ)
- If you have three different constraints, the robot cannot move
    - e.g three fixed wheels whose extended axes do not come to a point
    - Fixed or steed standard wheels each create one constraint

To describe the motion of the robot in response to motion of the wheels we need a forward kinematics model

- r = wheel diameter
- l = distance of wheels from central point on the connecting axis
- θ = current orientation of the robot
- (ϕ_1, ϕ_2) = velocity of each wheel

$$\dot{\zeta} = \begin{bmatrix}\dot{x} & \dot{y} & \dot{z}\end{bmatrix} = f(r, l, \theta, \phi_1, \phi_2) \tag{3}$$

Since we know how to convert the local coordinate system into the global one, we can just find the kinematics model for the robot's local coordinate system first.

Consider first the instantaneous translation of the robot's origin P in the X_r direction

- If one wheel moves while the other is staionary, P moves half as fast
- The contributions of the two wheels add together

$$\dot{x} = \frac{r\phi_1}{2} + \frac{r\phi_2}{2} \tag{4}$$

Instantaneous motion in the Y_r direction is impossible

$$\dot{y} = 0 \tag{5}$$

Finally, consider the instantaneous rotational velocity

- Motion of one wheel causes a rotation about the contact point of the other
- The robot is moving in an arc of a circle of radius 2l
- The robot will complete a circle, or 2π radians in $t = \frac{\pi(4l)}{r\phi_1}$
- To get the radians per second contribution of one wheel, divide 2π by *t*.
- The contributions of each wheel add together, but are inverses of one another.

$$\dot{\theta} = \frac{r\phi_1}{2l} - \frac{r\phi_2}{2l} \tag{6}$$

So the full kinematic model for a differental drive robot is:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = R(\theta)^{-1} \begin{bmatrix} \frac{r\phi_1}{2} + \frac{r\phi_2}{2} \\ 0 \\ \frac{r\phi_1}{2l} - \frac{r\phi_2}{2l} \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = R(\theta)^{-1} \begin{bmatrix} \frac{r}{2} & \frac{r}{2} & 0 \\ 0 & 0 & 1 \\ \frac{r}{2l} & -\frac{r}{2l} & 0 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ 0 \end{bmatrix} \tag{7}$$

Note that $R(\theta)^{-1}$ is simply the transpose of $R(\theta)$.

Note also that (7) looks a lot like (2), and we have the Jacobian of the differential drive robot.

Mobility v Steerability

Mobilty refers to the degrees of freedom of motion available to a robot by spinning the wheels

- Omnidirectional wheels--castor, Swedish, and spherical wheels--generate no constraints
- Fixed and steered standard wheels constrain possible motions

The constraint for a single standard wheel is that there can be no lateral motion

- This generates an equation that says lateral motion equals zero for each fixed wheel
- α is the angle of the a line from the origin to the center of the wheel, relative to the X-axis
- *l* is the distance from the origin to the center of the wheel
- β is the angle the wheel's axis of rotation makes with the line from the origin through the center of the wheel

$$\begin{bmatrix}\cos(\alpha+\beta) & \sin(\alpha+\beta) & l\sin\beta\end{bmatrix} R(\theta)\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = 0 \tag{8}$$

If we collect together the constraints for all of the wheels on the robot, we get a matrix

$$C_f R(\theta)\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = 0 \tag{9}$$

The solutions that satisfy this constraint form the NULL space of the matrix C_f.

- In the case of the differential drive robot, the matrix C_f has at most rank 2
- It turns out that, since the constraints are identical, the actual rank of C_f is only 1

The degree of mobility of a robot is the degree of the NULL space of the constraint matrix.

- So the degree of mobility of a differential drive robot is 2
- It can rotate or go forward instantaneously, just not sideways

Steered wheels add to the overall maneuverability of a robot

- A steered wheel adds a constraint to mobility because it can have no lateral motion
- A steered wheel adds the ability to steer, however, which cancels the constraint overall

Overall, the degree of mobility plus the degree of steerability defines the overall maneuverability of the robot

Holonomic v. Non-holonomic

A holonomic robot is one that has three degrees of mobility on the plane

- It can move in any direction on the plane without rotating (instantaneously)
- It can rotate in place

A non-holonomic robot cannot move instantaneously in any direction and/or rotate in place

We use non-holonomic robots for stability, speed, and terrain issues

- Fixed wheels, by definition, cause a robot to be non-holonomic in the plane
- Fixed wheels, however, permit passive resistance to forces
    - For example, cornering in a car exerts significant lateral accelerations
    - Fixed wheels passively resist the accelerations
    - An omnidirectional wheel would have to apply torque to the wheels to counteract it

Whe designing and following paths through the robot’s workspace, holonomy is important

- Holonomic robots can follow any path that is free of obstacles and achieve any pose
- Non-holonomic robots can achieve any pose, but not necessarily using the same trajectory
- Steerability is not equvalent to mobility - it takes time to steer the wheels

You can, theoretically, parallel park a car in a very tight space

- It takes a lot of time doing very small motions

E28/CS82 Lecture #4 S05

Overall Topic: Robot Control

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

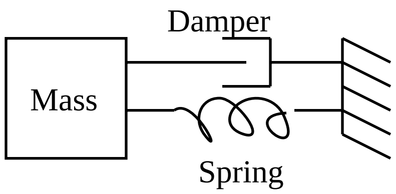
- It's like walking down a corridor with your eyes close and hands behind your back
- Eventually you run into the wall

Feedback control uses internal or external sensing to determine the current 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

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

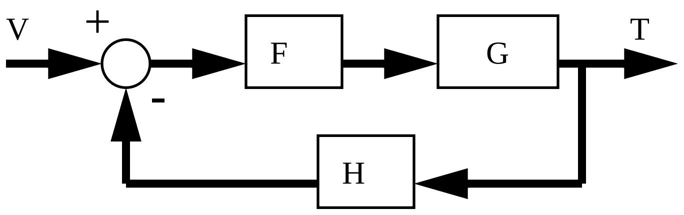
- Mass: 2nd order term, matched with acceleration
- Damper: 1st order term, matched with velocity
- Spring: 0th order term, matched with position
- $m\ddot{x} + b\dot{x} + kx = F(t)$



Motors are generally masses with damping

Most systems are run as closed loop control systems

- Feedback is almost always negative (positive feedback is bad)
- Transfer function of the closed loop system is generally expressed as a fraction
- $T = \frac{FG}{1 + FGH}V$
- H is the feedback function
- F can be a compensator function



Proportional control

The simplest control law is proportional control.

$$c(t) = k_p e(t) \tag{1}$$

The control signal is proportional to the error between the desired and actual system outputs

- The constant, k_p determines the speed and character of the response
- Think of it as setting $F = k_p$ and $H = 1$

There are four regimes in the response of the system to the control law

- $k_p < k_c$: overdamped region, 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.

- $k_u > k_p > k_c$: underdamped, but not unstable. The response will overshoot the desired output, but eventually settle to the desired output value.
- $k_p > k_u$: unstable, the response will never settle to the desired output, and will eventually reach the physical limits of the system 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

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

- If there is a step-disturbance in the system this will handle it
- May make the system respond in an underdamped fashion (with oscillations)

```
Error = Reference - Output
Integral = Integral + Error * T     // T = sampling rate
ControlSignal = K1 * Error + K2 * Integral
```

Now you have 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

Proportional-integral-derivative [PID] control

Trying to predict where the circuit is going also helps to improve the response of the system

If you also add a derivative function to F it will help the system look ahead

- If the error is decreasing (E < E1) the derivative will slow down the response
- If the error is increasing (E > E1) the derivative will increase the response

```
Error = Reference - Output
Integral = Integral + Error * T
Derivative = KD * (E - E1) / T
E1 = E
ControlSignal = K1 * Error + K2 * Integral + 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 reponsive

E28/CS82 Lecture #5 S05

Overall Topic: Robot Control

Some Clever Control Systems

Siegwart & Nourbakhsh goal achievement

- In the robot's frame of reference, the robot is at $\begin{bmatrix} x & y & \theta \end{bmatrix}^t = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^t$
- The goal is an arbitrary location $\begin{bmatrix} x & y & \theta \end{bmatrix}^t$ expressed in the robot's coordinate system
  - Note that each time the robot moves, the goal location moves as well
  - We can also express both the robot and the goal in global coordinates

$$E = \begin{bmatrix} x_r \\ y_r \\ \theta_r \end{bmatrix} - \begin{bmatrix} x_g \\ y_g \\ \theta_g \end{bmatrix} \tag{1}$$

- We want to be able to express a proportional control law in terms of E

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} k_{vx} & k_{vy} & k_{v\theta} \\ k_{\omega x} & k_{\omega y} & k_{\omega\theta} \end{bmatrix} E \tag{2}$$

- $v(t)$: forward velocity
- $\omega(t)$: steering angle

Mathematically, this is identical to moving the robot to the zero location in the global frame
- We know the kinematic equation of motion in the global frame

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3}$$

Transform the robot's position into polar coordinates relative to the goal position
- ρ: distance from the goal to the robot's center
- β: angle of the line connecting the robot to the goal point
- α: orientation of the robot relative to the line connecting the robot to the goal point

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -\cos\alpha & 0 \\ \frac{\sin\alpha}{\rho} & -1 \\ \frac{-\sin\alpha}{\rho} & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{4}$$

Notes on the polar kinematic model

- If $\alpha \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right]$ then the robot moves forward towards the goal
- If $\alpha \in \left(-\pi, -\frac{\pi}{2}\right] \cup \left(\frac{\pi}{2}, \pi\right]$ then the robot moves backwards towards the goal
- To keep the robot moving forwards, α must start in the first set and stay in the first set
- The velocity v must start positive and stay positive

Now go back to equation (2), and consider E in the polar coordinate notation with the following proportional control system.

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} k_p & 0 & 0 \\ 0 & k_\alpha & k_\beta \end{bmatrix} \begin{bmatrix} \rho \\ \alpha \\ \beta \end{bmatrix} \tag{5}$$

If we then substitute the right side of (5) into (4) we get the kinematic equation of motion due to the control system.

$$\begin{bmatrix} \dot{\rho} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = \begin{bmatrix} -k_\rho \rho \cos\alpha \\ k_\rho \sin\alpha - k_\alpha \\ -k_\rho \sin\alpha \end{bmatrix} \tag{6}$$

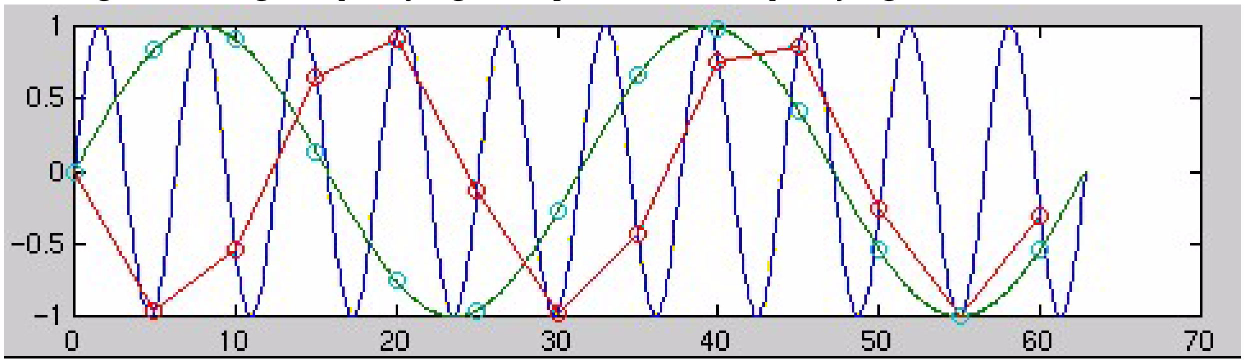
The important thing to note here, is that (6) does not have a singularity at $\rho = 0$

- The simple kinematic motion equation in (4) does have a singularity at that point
- In order to guarantee that that angles stay bounded

$$k_\rho > 0 \quad k_\beta < 0 \quad k_\alpha + \frac{5}{3}k_\beta - \frac{2}{\pi}k_\rho > 0 \tag{7}$$

Sampling

Aliasing: when a high frequency signal maps to a lower frequency signal



Nyquist Criterion: sample at more than twice the frequency at which events are happening that you care about

Sensors

Sensors sample the world provide feedback to the robot about the environment.

There are many types of sensors, and they have been categorized many different ways

- Axis 1: Exteroceptive v. Proprioceptive
  - Does it provide information about the external world (exteroceptive) or the robot itself (proprioceptive)
- Axis 2: Passive v. Active
  - Does the act of sensing involve emitting energy or changing the environment?
  - A passive sensor observes energy emitting from the environment
  - An active sensor emits energy into the environment and observes the result

Sensor vocabulary

- **Precision**: number of significant digits in the result $p = \frac{\text{range}}{\sigma}$
- **Accuracy**: how close you are to the true value
- **Sensitivity**: what is the smallest change you can sense reliably
- **Resolution** (**signal**): # bits in an ADC
- **Resolution** (**time**): sampling frequency
- **Resolution** (**spectral**): how precisely can you measure frequency?
- **Dynamic Range**: range within which the sensor operates
- **Bandwidth**: Bits/s at which the sensor provides data

Errors

- Random
- Systematic
- Form of the error function
  - Gaussian
  - Multi-modal
  - Symmetric
  - Random (generic histogram)

Example sensors:

- Proprioceptive sensors
  - Optical encoders on the motor shaft
    - Generally assumed to be 100% accurate (and they are compared to other errors)
  - Motor current
    - Sense stalling or level of traction
  - Inertial sensors
    - Mechanical or optical gyroscopes provide angular velocity information
    - Newest sensors are laser based with no moving parts: watch frequency changes
  - Hall-Effect compass or Flux gate compass
    - Latter is more expensive, but more accurate and precise
    - Get general heading information
    - Subject to systematic errors and random errors
  - Battery power remaining
  - Error conditions

E28/CS82 Lecture #6 S05

Overall Topic: Robot Sensors

Sensors

- Bump sensors
  - Generally binary switches that provide an on/off signal
  - More sophisticated versions provide a signal that relates to the force of the contact
- Sonar
  - High frequency sound waves
  - Send a burst of sound, wait for the echo to return
  - Reasonable data from 0.5m to 4m, depending upon the sonar type.
  - Each sonar type has a particular transmission profile
    - Generally defined as a cone of sensitivity, typically 20º-40º
    - As distance increases, the cone gets very large
  - Sonar provides reasonable accuracy in distance, but not in angle to the target
    - Target can be anywhere on an arc within the zone of sensitivity
  - Sonar is subject to specular bounces
    - At some angles, the sound bounces off the object as a reflection
    - A specular bounce makes it look like there is nothing there
  - Some materials reflect better than others
- IR intensity-based range sensors
  - Uses an IR LED to send out focused light
  - Senses the strength of the light with an IR photoresistor or phototransistor
  - Signal is based on the strength of the reflected signal
    - As the distance to the object increases, the reflected signal decreases
    - Objects may also have differing reflectance coefficients
  - IR sensors need to be calibrated to convert reflected signal strength into distance
  - Range depends upon the sensor, but they work better close up than far away
    - Useful range is 0.1 to 1.0m on the robots
- Laser range sensors
  - Numerous laser range sensor geometries
    - Point: single beam
    - 1D: line of light, uses a prism or a rotating mirror to generate the beam
    - 2D: plane of light, uses a rotating mirror and/or prism to generate the pattern
  - Time of flight systems are based on one of three principles
    - True time of flight estimation: send a pulse, wait for it to return (expensive)
    - Frequency modulated signal and beat shifting
    - Phase-shift: send a sine wave and measure the phase shift with the returning signal
    - Typical phase-shift sensor uses a frequency of 5MHz, with maximum distance of 60m
  - Triangulation-based systems use a 1D or 2D sensor to find the laser spot
    - A lookup table converts the reflected spot location to distance (based on geometry)
    - Some IR sensors are based on a 1D triangulation
    - Max range of triangulation-based systems is lower, accuracy decreases w/distance

E28/CS82 Lecture #7 S05

Overall Topic: Robot Sensors, Configuration space

Robot Sensors: Cameras

Greyscale camera: single sensor plane with panchromatic sensitivity

Color camera

- Single CCD system
  - Get color by having very small triples of RGB filters and CCD elements
  - Get color by using depth to which photons penetrate: blue goes furthest
- 3-CCD system with filters to give RGB
  - Requires precise calibration and strong mounting
  - Get full size pixels, highest resolution at lowest noise
  - Expensive

Stereo camera systems: usually two or more greyscale cameras

- Precision is dependent upon the baseline distance: wider is more precise/sensitive
- Wider baseline, however, makes matching harder, decreasing accuracy
- Calibration is critical

Structured light stereo systems

- Active system where a light pattern is projected on to the scene
- Light pattern can be static, or a series of different patterns

Panoramic/omnidirectional cameras

- Have the camera look up into a convex conic-like mirror
- 360^o view of the world, certain geometries permit the ground plane to be flat

Catadioptic cameras

- Cameras with mirrors in front of the lens
- Can provide single camera stereo or execute certain types of preprocessing

Infrared cameras [FLIR]

- 2D plane of IR sensitive elements
- Calibrated cameras can provide accurate temperature measurements

For all cameras, distance has more error than the angle of the object, the opposite of sonar.

- Microphones/audio sensors
  - Can use an array of microphones to identify the direction of incoming sound (need 4)
  - Useful for human-robot interaction (speech recognition)
- Wind/wind speed sensors
- Motion sensors (usually IR based)
- Temperature sensors
- Chemical sensors (robot noses)
- Capacitive sensors
- Force sensors

E28/CS82 Lecture #8 S05

Overall Topic: Configuration Space and Robot Motion

Robot Motion

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

- Random walks
  - Collision avoidance and collision prediction
- Fixed goals
  - Go to a point or series of points
- Coverage
  - Explore or cover an area
  - Try not to backtrack to much
- Dynamic goals
  - Go to a possibly moving point
  - Specify the goal as an abstract concept (serve people)

We're 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
- The robot moves in Euclidean space W (***workspace***), represented as R^N (N=2,3)
- Let B_1 to B_Q be Q fixed rigid objects distributed in W. These are ***obstacles***
- Assume that the geometry of A, the B_i and the locations of the B_i are accurately known
- Assume there are no kinematic constraints on A (A is ***free-flying***)
  - A circular differential drive robot is close enough
  - Can rotate in place before heading in any direction

Now we can state the problem 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.

Configuration space

Given the case of simple motion, 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 of all possible configurations of the robot is known as the ***configuration space***

- A is a closed subset of W
- The B_i are all closed subsets of W
- Attach coordinate systems to both A (F_A) and W (F_W)
- The position and orientation of F_A with respect to F_W is the configuration of the robot

We know we can describe the relationship of two coordinate systems

We can also describe paths in configuration space

- A ***path*** is a mapping T[0,1] -> C
  - T(0) is the initial state
  - T(1) is the final state
  - The function T(p) is a parametric representation of the curve, where $p \in [0, 1]$
- To define a path and continuity we need to define a ***distance metric***
  - d(q, q') = $\max_A$ || a(q) - a(q') ||
  - This is the maximum distance between related points of the robot's subset
- We can define ***continuity*** by saying that for all $s_0 \in [0, 1]$, as you get close to s_0 on the path defined by T the distance metric goes to zero (you can get there).
- Within these definitions, ***free-flying*** means that 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
  - CBi = {q in C | A(q) union Bi != 0}
- Obstacle space ***C-obstacle*** = union of Bi
- ***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

- A point robot or a disc robot is easy to map to configuration space
- C is equal to R^N, as long as the orientation of the robot can be ignored
  - Since a circular robot can rotate in place, orientation doesn't matter wrt paths

Mapping obstacles to the configuration space of a simple robot

- Grow the obstacles by the radius of the robot plus a safety factor
- Then make the robot a point, reducing the complexity of intersection calculations
  - 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.

Planning Techniques

There are three major approaches to symbolic planning in mobile robotics

- Visibility graphs and Retractions (Roadmaps)
- Cell Decompositions
    - Exact
    - Approximate
- Potential Fields

The first two require some kind of global planning and search, the third can be a local method

- All of them can be used in both static or dynamic environments
- All can be used whether the obstacles are known a priori or discovered en route.

From limited observations (i.e. homeworks) visibility graphs seem to be closest to the method people use to navigate between locations.

Visibility Graphs: Concept

Assumptions: A, B are polygonal, W = R^2, translational robot, so C = R^2.

The idea of the visibility graph is to construct a semi-free path as a polygonal line from Q_0 to 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 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, the 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 vertices of CB.

Therefore, it is sufficient to consider the space of lines between vertices when planning a path.

- Nodes: Q_0, Q_F, and the vertices of CB
- Visbility graph: all lines segments between vertices that are edges of CB or that lie in C_{free} except for possibly their endpoints.

In English, if you have a set of obstacles, you go around them as closely as possible at the corners.

- Since configuration space has grown the obstacles, the corners are “safe”

Drawback is that you come close to obstacles and have to rely on your safety factor

E28/CS82 Lecture #9 S05

Overall Topic: Planning Methods

Planning Methods: Visibility Graph

Assumptions: A, B are polygonal, W = R^2, translational robot, so C = R^2.

The idea of the visibility graph is to construct a semi-free path as a polygonal line from Q_0 to 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 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, the 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 vertices of CB.

Therefore, it is sufficient to consider the space of lines between vertices when planning a path.
- Nodes: Q_0, Q_F, and the vertices of CB
- Visbility graph: all lines segments between vertices that are edges of CB or that lie in C_{free} except for possibly their endpoints.

In English, if you have a set of obstacles, you go around them as closely as possible at the corners.
- Since configuration space has grown the obstacles, the corners are "safe"

Drawback is that you come close to obstacles and have to rely on your safety factor

Visibility Graphs: Algorithm

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
- $O(n^3)$

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
- $O(n^2 \log n)$

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
  - Result is a map of shortest paths through the space from a source point P
- Searching the path is O(logN) once you have the subdivision (Dijkstra's algorithm)
- Overall complexity is O(nlogn)

Retractions

Let X be a topological space, Y a subset of X. A mapping X->Y is a retraction if and only if it is continuous and its restriction to Y is the identity map. (The piece of X that is 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 Cfree -> R, where R, a subset of Cfree, is a network of 1D-curves. There exists a free path between two free configurations Q0 and QF if and only if there exists a path in R between P(Q0) and P(QF).
>
> Proof: If a path exists in Cfree, then P preserves connectivity and Q0 and QF -> R. If a path exists in R, then a path in Cfree exists that has three components: 1) a path from Q0 to P(Q0), 2) a path from P(Q0) to P(QF), and 3) a path from P(QF) to QF. 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 Cfree
- 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(NlogN) algorithms for generating Voronoi diagrams

- Use a library like CGAL to make them

Once you have a retraction, you need to search it for the shortest path

- Dijkstra-type algorithm
- A-star type search algorithm

Generic Visibility Graph/Retraction Planning Algorithm

1. Construct the visibility graph or retraction
2. Search the graph
    - A* search
    - Dijkstra’s algorithm
3. Return either the path or failure

E28/CS82 Lecture #10 S05

Overall Topic: Planning Methods

A* Search Algorithm

General search algorithms

- Terms
  - State space: the space of all possible configurations for the task
  - Root node: initial starting point
  - Child nodes: places you can get to from a given configuration
  - OPEN list: places to explore that are connected to places you've been
  - CLOSED list: places you have already explored
  - Order the OPEN list by increasing cost
    - g() function: cost from root to the current node
    - h() function: estimated cost from the current node to the goal
- Algorithm
  - Put the root node on the OPEN list
  - 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
      - If the child is on the OPEN list
        - Carefully consider what to do if it is
        - 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 order
      - 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
  - If the OPEN list is empty, return failure in the search

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

Anytime Search Algorithm: IDA*

- It turns out that iterative deepening is a real-time/anytime version of A*
- Run the A* algorithm to a certain f value, return the best result
- Increase the maximum f value and run it again
- Since you always expand most of the nodes at the bottom level, the cost increase is not significant compared to the total cost of the search
- If you have to stop searching because of time constraints, you can always return the result of the deepest search that completed

Dijkstra's Algorithm

This is a greedy algorithm, and basically a simpler, less general version of A*

- Still is proven to work

Dijkstra's algorithm builds a spanning tree

- A graph with no loops that gives the shortest path from each node to the source node
    - Note that getting all shortests paths is no harder than just getting one

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;
```

E28/CS82 Lecture #11 S05

Overall Topic: Grid-based Planning

Approximate Decomposition

Approximate decomposition involves approximating the world with a set of polygons--usually regular polygons such as squares or rectangles.

If you can find a path through the regular polygons, then you can find a path from Q0 to QF

- Exact decomposition is the same idea, but uses potentially irregular polygons to fit Cfree exactly.

Approximate decompositions can be very useful when developing maps, because they are easy to manipulate.

- Decompose the world into regular blocks on a grid
- Mark blocks that contain obstacles you know about
- As you get new information, mark or un-mark blocks depending upon the information

A grid maps onto a graph with connections between a grid cell and each of its neighbors

- For smoother motion, you can also connect each cell with others inside a certain radius

In the simplest case (perfect knowledge) search for a free path by using A* search on the graph.

Demo example here

Occupancy grid

A regular grid that has enumerated values: obstacle, no obstacle, hard obstacle, soft obstacle, etc.

Evidence grid

A regular grid that has continuous values at each grid point representing the probability that the grid cell is occupied by something.

- Each cell has a log probability that something is there

Sensor models tell us what a particular reading from a sensor means

- A sonar provides a single number: a distance d
- A sonar model is a function that tells us the probability of detecting an obstacle given that there is something at a distance x.
- For most sensors, the likelihood of detecting an obstacle at distance d < x is low, the likelihood of detecting an obstacle at distance d >= x is high. We can learn this function by taking lots of measurements.

To combine information, or add new information, we need to update the cell probabilities, which give the probability that the cell is occupied, *p(o)*

- First, consider Bayes rule: $p(A|B) = \frac{p(B|A)p(A)}{p(B)}$
- Let the probability of occupation be p(o), the probability of non-occupation be p(o')

$$\frac{p(o|M)}{p(\hat{o}|M)} = \frac{p(M|o)p(\hat{o})}{p(M|\hat{o})p(o)} \tag{1}$$

- Suppose we have *p(o | M_1)* already in a cell (likehood of occupation given a measurement)
  - Note that the sensor model is actually *p(M | o)*, which is why we use Bayes rule
- Assume the errors of our sensor readings are independent
  - p(M_1 and M_2 | o) = p(M_1 | o)p(M_2 | o)
- We can no rewrite (1) as

$$\frac{p(o|M_1 \wedge M_2)}{p(\hat{o}|M_1 \wedge M_2)} = \frac{p(o|M_1)p(o|M_2)p(\hat{o})}{p(\hat{o}|M_1)p(\hat{o}|M_2)p(o)} \tag{2}$$

- Assume that the initial probability of occupation is 0.5, and the p(o) terms cancel
- The values p(M | o) values are called the sensor model values, which can be learned

By using a log representation of the probabilities--in addition to being more numerically tractable--we just add the sensor model values to the occupation grid to update the map.

Building an Evidence Grid

An evidence grid gets built as sensor data accumulates

- Each sensor response affects a particular portion of the grid
- Each sensor has a sensor model, which represents p(M|o)
  - Probability of a measurement given an obstacle
  - Can generate this by taking lots of measurements of obstacles at known distances
  - Often a normal distribution, although it may have non-normal characteristics (sonar)
- A sensor model has a coverage area where probabilities are non-zero
  - Also called the sensor mask
  - Example: a sonar's mask is generally a clipped cone or a triangle

For each sensor reading

- Place the mask at the appropriate location on the evidence grid
  - Accurate localization is important
  - Accurate sensor mask is important
  - Accurate sensor calibration in the robot's coordinate frame is important
- For each grid cell within the mask, update the odds
  - odd1 = cur[j] / (1-cur[j])
  - odd2 = new[j] / (1-new[j])
  - newOdds = odd1*odd2
  - cur[j] = newOdds / (1 + odds)

Storing the results this way means you don't have to use logs: all values are between 0-1, and as they get close to zero or 1 precision doesn't matter so much (never let it quite get to either).

Using an evidence grid

You can execute an A* search on an evidence grid

- Grid cells with high odds of occupation should get very high costs
- Grid cells with low odds of occupation should get small costs
- Occupation costs add to the distance traveled, and may want to allow diagonals or jumps

E28/CS82 Lecture #12 S05

Overall Topic: Potential Fields & Behavior-based planning

Potential Fields

Potential fields are a non-global method of path planning
- Simple to compute
- Get a global solution for non-convex obstacle sets
- Form a control scheme in addition to a being a path-planning method

Concept is that the goal attracts the robot, while objects repel
- Computation of the repellant force is based on all the obstacles in a local window
- Computation of the attracting force is based on distance to the goal

The direction of motion is in the downhill direction (gradient) of the potential field U

$$F(q) = -\nabla U(q) \qquad \nabla U = \begin{bmatrix} \frac{\partial U}{\partial x} \\ \frac{\partial U}{\partial y} \end{bmatrix} \tag{1}$$

$$U(q) = U_{\text{att}}(q) + U_{\text{rep}}(q) \tag{2}$$

Just as the potential field has two additive components, so does the force on the robot:

$$F(q) = -\nabla U_{\text{att}}(q) - \nabla U_{\text{rep}}(q) \tag{3}$$

A simple method of setting up the attractive force (generated by a goal) is to use a parabolic function of the Euclidean distance to the goal.

$$U_{\text{att}}(q) = \frac{1}{2}k_{\text{att}}\rho^2 \qquad \nabla U_{\text{att}}(q) = k_{\text{att}}(q - q_{\text{goal}}) \tag{4}$$

Note that this makes up a simple proportional control law based on distance to the goal.

The repulsive field is a little more dificult, since we want to put hard boundaries on the extent of the repulsive field (distance objects should not repulse). One example is:

$$U_{\text{rep}}(q) = \begin{cases} \frac{1}{2}k_{\text{rep}}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho(q) \le \rho_0 \\ 0 & \text{if } \rho(q) \ge \rho_0 \end{cases} \tag{5}$$

$$\nabla U_{\text{rep}}(q) = \begin{cases} \frac{1}{2}k_{\text{rep}}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)\frac{1}{\rho^2(q)}\frac{(q - q_{\text{obstacle}})}{\rho(q)} & \text{if } \rho(q) \le \rho_0 \\ 0 & \text{if } \rho(q) \ge \rho_0 \end{cases} \tag{6}$$

There are several problems that exist with potential fields
- Concave or nearby obstacles can result in local minima in the potential field
- Concave obstacles can result in a non-continuous potential field

In short, a potential field is not a global planning solution in complex obstacle spaces
- For hallways and most open human spaces, however, it works very well
- You can generate the potential field based on local sensor information
- Good at obstacle avoidance

Potential field combined with a global search technique can produce an effective hybrid system

How do you convert the force on the robot into velocity and translation components?
- The gradient gives you a direction in which you need to travel
- The error between current orientation and desired orientation gives you rotational velocity
- The subtractive inverse and distance to the goal gives you translational velocity
    - Max velocity is set by the distance to the goal and a proportional constant.
    - At 90º or greater to the proper orientation, translation should be zero
    - Proportionally increase velocity as angular error decreases

Extended Potential Field

Combining veleocity-space concept with potential field background
- Don’t turn unless you really have to

Concept is to reduce the effect of an obstacle’s repulsive force if it is not in front of the robot
- Able to drive along the side of a hill in the potential space rather than only along valleys
- Improved wall-following

One way to implement it is to use the velocity-space concept
- Given current sensor readings, calculate your dynamic velocity space
    - Obstacles will cast shadows in the velocity space that indicate bad speed/angle choices
- If you can continue on your trajectory at the desired speed, do so (ignore repulsive forces)
- Any obstacles that block the desired trajectory should repulse the robot
- Be sure to include the obstacle growth and safety factors in the velocity space calculations

Obstacle Avoidance v. Navigation

Obstacle avoidance means not running into things
- Does not imply the robot has a goal
- Does not guarantee the robot will achieve a particular location

Navigation implies a goal, strategies achieve the goal or return failure
- Obstacle avoidance should be a part of navigation

Robot Software Architectures

First robot systems were monolithic: one program did everything

- Sense-Plan-Act paradigm
- Really slow, and the world could change between sensing and acting

Reactive Systems/Subsumption Architecture

Rod Brooks changed the robotics world by showing that you could sense-act in a fast loop

Concept is that the world forms the state vector

- What you do at any given time is determined by the robot's state
- The world stores the state, the sensors give you access to it
- No stored state is required, all behavior is a reaction to current sensor

Concept is similar to a local potential field at the base level

- Obstacles push
- Robot wants to move forward

Multiple behaviors may be vying for control over the motors

- Forward motion in the absence of obstacles
- Obstacle avoidance through local potential fields
- Wall-following when close enough to an obstacle (bug-style algorithm)

Different architectures combine these in different ways

- Subsumption: one behavior takes over based on current sensor inputs
- Fuzzy logic: behavior outputs get combined using a fuzzy control system
- Case-based: sensor states are divided into cases (crisp system)
- ANN: low-level behaviors may be trained using ANNs
    - ANN outputs may be combined using any of above methods

E28/CS82 Lecture #13 S05

Overall Topic: Robot Architectures

Robot Software Architectures: Multi-Layer Architectures

Problem with a simple reactive system is that it is difficult to execute long-term sequences

- Possible, but really tough to implement and get working robustly

Problem with monolithic architectures is you can't react fast enough to dynamic environments

Multi-layer architectures try to get the best of both worlds

- Use a reactive system at the bottom layer
    - Obstacle avoidance
    - Emergency maneuvers (i.e. stopping)
    - Wall-following
    - Potential field type goal achievement
- Use a planning system at the top layer
    - Build maps
    - A* or Dijkstra's algorithm to plan paths
    - Update maps and paths as necessary
- Use an arbiter as the middle level
    - Determines what the robot will do each time through the loop
    - Listens to all of the layers for information about what to do

REAPER Architecture

REAPER architecture concept is a cross between a blackboard and a multi-layer architecture.

- Blackboard concept is originally from speech and natural language understanding
- Blackboard is a central repository for current hypotheses about the world
- Modules can read and modify the current hypotheses based on their knowledge

REAPER integrates multiple sensing modalities into a multi-layer architecture

- Sensing is not relegated to the lowest level, except for sensors integrated with the robot hardware (sonars, IRs, bump sensors)
- Sensors like microphones and cameras have their own modules
    - The modules are in charge of extracting information from the data
    - The information is passed on (much lower data rate, much higher utility/bit)

The central module still decides what the robot needs to be doing, but the nav module actually makes the final decision about the robot's physical actions, permitting a reactive system to handle the details, if necessary.

- The central module is listening to all of the sensors
- The central module may also ask other modules for information
    - May as mapping/localization/planning module for a path to a goal
    - May ask a knowledge DB module to intepret some information or plan an action
    - May ask a sensing module for different information (turn on an operator)

Reasoning

There are many ways the central module of a robot can make decisions about what to do next and how to combine the suggestions of competing behaviors.

Subsumption
- One behavior wins, world holds state information

Case-based reasoning
- Divide all sensor responsing into cases, determine an action in each case

Fuzzy logic
- Use fuzzy logic to implement a soft case-based reasoning system
- Can combine the output of multiple behaviors

ANNs
- Can also use ANNs to implement local decision-making
- ANNs can be behaviors, and can learn to modulate between behaviors

There are also several methods of having the robot execute sequential actions

World-as-state
- Map sensor readings into control signals and use a subsumption-type architecture

State machines
- Define action sequences as states in a state machine
- Use sensor readings and internal information to determine when to switch states

Reasoning systems
- Define rules sets that describe how to achieve goals
- Set the robot a goal
- Let the robot query its knowledge DB (rule set) to figure out what it needs to do next

Hybrid systems
- Generally built in a multi-layer framework
- Low-level provides reactive control to handle emergency/local situations
- Central-level includes a state machine to manage actions
- High-level responds to queries from the central level about how to achieve goals

Fuzzy Logic

Programming and traditional control system techniques allow us to create two different kinds of control systems.

- Programming: if-then style rules combined with continuous functions
- Traditional control: continuous functions of the error between target state and actual

Neural networks provide one alternative to these methods that permits more flexibility

- Control function is defined by example
- Network can learn arbitrary continuous functions

However, ANNS are difficult to analyze, which makes their application in critical systems questionable unless you have a lot of faith in the network designer and the network's training set.

Fuzzy logic offers another alternative to the traditional “crisp” methods

- Permits control rules to be stated in an if-then style
    - Provides transparency to design and analysys
- Offers a way to mathematically define adjectives and adverbs modifying those rules
- Offers a way to produce a continuous output

Fuzzy logic has been successfully used to develop analysis and control systems

Fuzzy set definitions

Fuzzy logic is based on the notion of a fuzzy set

- All the terminology is fuzzy, but the definitions are quite crisp

A crisp set is a binary-valued operator on a “universe of discourse” that divides the universe

The **fuzzy set** A is defined as a set of ordered pairs:

$$A = \{(x, \mu_A(x)) \mid x \in X\} \tag{1}$$

Each ordered pair consists of two parts: an element *x* and a membership value $\mu_A(x)$.

- The function $\mu_A(x)$ is called the **membership function**
    - Crisp sets have a binary valued membership function
    - The membership function can be continuous or discrete
- X is the **universe of discourse**, or simply the universe

Given this definition, it is easy to construct a fuzzy set given a universe and membership function

- The membership function integrates subjectivity into the definition of the set
- Differences in how people define sets for a universe introduces nonrandom variation

Fuzzy sets are a method of quantifying and studying subjectivity and nonrandomness

- Traditional statistics apply objective criteria to random phenomena

In order to standardize membership functions, it is common to use **normalized fuzzy sets**:

- For a normalized (or normal) fuzzy set, the membership function is in the range [0, 1] and there is at least one element *x* for which the membership value is equal to 1.

The **support** of a fuzzy set consists of all elements x in X such that the membership is non-zero

$$support(A) = \{x \mid (\mu_A(x) > 0)\} \tag{2}$$

The **core** of a fuzzy set is the set of all elements x in X such that the membership value is one

$$core(A) = \{x \mid (\mu_A(x) = 1)\} \tag{3}$$

A **fuzzy singleton** is a fuzzy set whose support is a single point X with $\mu_A(x) = 1$

An **α-cut** or an **α-level** set is a crisp subset of a fuzzy set defined by

$$A_\alpha = \{x \mid (\mu_A(x) \ge \alpha)\} \tag{4}$$

E28/CS82 Lecture #14 S05

Overall Topic: Fuzzy logic

Fuzzy Logic: Fuzzy relationships

Common crisp set operators include subset, union, intersection, and complement. We can also define these operations for fuzzy sets. This set of definitions for union, intersection and complement are referred to as the **classical** or **standard fuzzy operators**.

Fuzzy set A is a **subset** of, or **contained** in fuzzy B if and only if for all elements *x*, the element's membership value in A is less than or equal to its membership in B.

$$A \subseteq B \Leftrightarrow \mu_A(x) \le \mu_B(x) \tag{1}$$

The **union** (**or**) of fuzzy sets A and B, $A \cup B$, has the membership function:

$$\mu_{A \cup B}(x) = max(\mu_A(x), \mu_B(x)) = \mu_A(x) \vee \mu_B(x) \tag{2}$$

The **intersection** (**and**) of fuzzy sets A and B, $A \cap B$, has the membership function:

$$\mu_{A \cap B}(x) = min(\mu_A(x), \mu_B(x)) = \mu_A(x) \wedge \mu_B(x) \tag{3}$$

The **complement** (**not**) of a fuzzy set A, $\overline{A}$, has the membership function:

$$\mu_{\bar{A}}(x) = 1 - \mu_A(x) \tag{4}$$

Note that Boolean algebra is a special case of these definitions where the membership functions are binary-valued.

- All the nice little logical relationships on Boolean sets hold under these definitions
- (This includes DeMorgan's laws)

There are other definitions for these operators that satisfy appropriate rules (boundary, monotonicity, commutatitivity, associativity, and a generalized DeMorgan's law). The generalized intersection operator is called a T-norm function, the generalized union operator is called a T-conorm.

Example:

- Algebraic product/sum/complement
    - $T_{ap}(a, b) = ab \ / \ S_{ap}(a, b) = a + b - ab \ / \ N(a) = 1 - a$

Membership functions [MF]

One-dimensional membership functions come in a variety of flavors, but there are some standard functions for continuous variables that are used in the literature (and some robotics packages).

A **triangular** MF is specified by three parameters {a, b, c}, which specify the beginning, peak, and end of a triangular function on the real number line.

$$triangle(x;(a,b,c)) = \begin{cases} 0 & x \le a \\ \frac{x-a}{b-a} & a \le x \le b \\ \frac{c-x}{c-b} & b \le x \le c \\ 0 & c \le x \end{cases} \tag{5}$$

A **trapezoidal** MF is specified by four parameters {a, b, c, d}, which specify the four corner locations of the trapezoid along the real number line.

$$trapezoid(x;(a,b,c,d)) = \begin{cases} 0 & x \le a \\ \frac{x-a}{b-a} & a \le x \le b \\ 1 & b \le x \le c \\ \frac{d-x}{d-c} & c \le x \le d \\ 0 & d \le x \end{cases} \tag{6}$$

For smoother functions, common choices include:

$$Gaussian(x;(c,\sigma)) = e^{\left(-\frac{1}{2}\right)\left(\frac{x-c}{\sigma}\right)^2} \tag{7}$$

$$bell(x;(a,b,c)) = \frac{1}{1+\left|\frac{x-c}{a}\right|^{2b}} \tag{8}$$

$$sigmoid(x;(a,c)) = \frac{1}{1+e^{-a(x-c)}} \tag{9}$$

The **Gaussian** and **bell** curves produce convex, closed fuzzy sets, while the **sigmoid** produces a convex open fuzzy set.

- The parameters select the center and the steepness for each function

Combinations of membership functions is common using a Left-Right MF

$$LR(x;(c,\alpha,\beta)) = \begin{cases} F_L\left(\frac{c-x}{\alpha}\right) & x \le c \\ F_R\left(\frac{x-c}{\beta}\right) & x \ge c \end{cases} \tag{10}$$

Using a left-right function, for example, you can generate a closed, convex fuzzy set using two sigmoids that overlap and point in opposite directions.

Membership functions generalize to two dimensions and higher

- A 2D normalized histogram is an example of a 2D fuzzy membership function

Defining functions on fuzzy sets

The **extension principle** lets us define standard crisp functions on fuzzy sets

- Given a fuzzy set $A = \{(x_1, \mu(x_1)), (x_2, \mu(x_2)), \ldots, (x_N, \mu(x_N))\}$ and $y = f(x)$
- Define a new set:

$$B = \{(f(x_1), \mu(x_1)), \ldots, (f(x_N), \mu(x_N))\} \tag{11}$$

The issue here is what happens when the function $f(x)$ is not one-to-one, but many-to-one?

- Could have multiple $f(x)$ values with different membership values
- In general, keep the max membership value for all the variables that map to $f(x)$

If the function takes multiple fuzzy sets as inputs, then keep the max of the min

- $f(x, y)$ is a function of multiple fuzzy variables
- Define a new set:

$$C = \{(f(x_1, y_1), min(\mu(x_1), \mu(y_1)), \ldots, f(x_N, y_N), min(\mu(x_N), \mu(y_N)))\} \tag{12}$$

- If multiple variable pairs map to the same $f(x, y)$, keep the max membership value

We can also define other relationships, or binary functions, on fuzzy sets

$$R = \{((x, y), \mu_R(x, y)) \big| (x, y) \in X \times Y\} \tag{13}$$

This simply says that using a 2D membership function we can define binary relationships on two fuzzy variables. Examples of 2D fuzzy functions include:

- x is close to y
- x depends upon y
- x looks like y
- if x then y

We may also want to compose binary relationships $R_1 \otimes R_2$.

- Example: x depends upon y, y depends upon z, how does x depend upon z?

Ideally, we want the composition operator to have the nice properties of

- Associativity $R \otimes (S \otimes T) = (R \otimes S) \otimes T$
- Distributivity over union $R \otimes (S \cup T) = (R \otimes S) \cup (R \otimes T)$
- Weak distributivity over intersection $R \otimes (S \cap T) \subseteq (R \otimes S) \cap (R \otimes T)$
- Monitonicity. $S \subseteq T \Rightarrow R \otimes S \subseteq R \otimes T$

There are two methods for composing the membership functions of two binary relationships

Max-min composition: $\mu_{S \otimes T}(x, z) = max_y[min[\mu_S(x, y), \mu_T(y, z)]]$ (14)

Max-min composition says the relationship between x and z is the maximal path cost between x and z, where the path cost is the minimum of the steps along the path. The max is taken overall all paths, which is all possible values of y since that is the free variable (x and z are fixed).

Max-product composition: $\mu_{S \otimes T}(x, z) = max_y[\mu_S(x, y)\mu_T(y, z)]$ (15)

Max-product composition says the relationship between x and z is the maximal path cost between x ans z, where the path cost is the algebraic product of the steps along the path.

Linguistic variables

Fuzzy variables and rules or operations on those variables are often phrased in English. Often there is an underlying variable (universe) over which we want to define multiple fuzzy sets based on modifiers to the variable.

- Example: Let *age* be the variable. Sets might be young, middle-aged, old.

A linguistic variable is defined as a quintuple (x, T(x), X, G, M)

- *x* is the name of the variable (age)
- *T*(*x*) is the term set of *x* (young, middle-aged, old, very old, not very old, ...)
- *X* is the universe of discourse (numbers between 0 and 150)
- *G* is the syntactic rule for generating the terms *T* (mixing terms and adverbs/adjectives)
- *M* is the set of membership rules defining the space of fuzzy sets for the variable *x*

Concentration and dilation of fuzzy sets

It is possible for the syntactic rules of a linguistic variable to allow the use of modifiers such as “very”, “somewhat”, “extremely”, and so on. It is useful to have a standard method for modifying an existing membership function to account for these linguistic hedges without have to design a new membership function from scratch.

The generic concepts of concentration and dilation are defined in a standard way. First, we can define the modification of a fuzzy set A^k as in (16).

$$A^k = \int_X [u_A(x)]^k / x \qquad (16)$$

Then **concentration** is defined as A^2, and **dilation** is defined as $A^{0.5}$. Using these terms in combination with the AND, OR, and NOT relationships defined above, we can now define sets like

$$\text{“not very young and not very old”} = \neg young^2 \cap \neg old^2 \qquad (17)$$

simply based upon the membership functions defined for *young* and *old*.

E28/CS82 Lecture #15 S05

Overall Topic: Fuzzy logic

Fuzzy inference

Now that we can define fuzzy sets, functions and compositions on them, and apply them to a range of linguistic expressions, it would be nice to start creating fuzzy rules for decision-making and control.

The basis for fuzzy reasoning and control is the **if-then** rule: if A then B ($A \rightarrow B$). We can re-write this as an expression on sets as in (1)

$$A \rightarrow B = \neg A \cup B \tag{1}$$

Unfortunately, this variant of inference is only definitive for two-valued logic (Boolean). There are several other variations on inference including:

$$\text{A coupled with B: } A \rightarrow B = \int_{X \times Y} \mu_A(x) \bullet \mu_B(y) / (x, y) \tag{2}$$

$$\text{Propositional calculus: } A \rightarrow B = \neg A \cup (A \cap B) \tag{3}$$

$$\text{Extended propositional calculus: } A \rightarrow B = (\neg A \cap \neg B) \cup B \tag{4}$$

These variations, combined with the possible variations on AND, OR, and NOT means there is no single definition for fuzzy inference. The following are definitions that have been used:

$$\text{A coupled with B (Classical): } A \rightarrow B = \mu_{A \times B}(x, y) = \mu_A \wedge \mu_B \tag{5}$$

$$\text{A coupled with B (Algebraic): } A \rightarrow B = \mu_{A \times B}(x, y) = \mu_A \mu_B \tag{6}$$

$$\text{Material implication (Classical): } A \rightarrow B = \neg A \cup B = (1 - \mu_A) \vee \mu_B \tag{7}$$

$$\text{Material Implication (Algebraic): } A \rightarrow B = 1 \wedge 1 - \mu_A + \mu_B \tag{8}$$

$$\text{Propositional calculus (Classical): } A \rightarrow B = (1 - \mu_A) \vee (\mu_A \wedge \mu_B) \tag{9}$$

$$\text{Extended propositional calculus (Classical): } A \rightarrow B = (1 - \mu_A \cap 1 - \mu_B) \cup \mu_B \tag{10}$$

Modus ponens reasoning

Reasoning based on modus ponens (if-then statements) is well-defined for two-valued logic.

- Identify a predicate A that is true
- Identify if-then rules for which A is precondition
- Apply the rule to discover the postconditions given that A is true

Reasoning in fuzzy logic requires some generalization of modus ponens reasoning

- The predicate A is true to a degree given by its membership function
- Identify if-then rules for which A is a precondition
- Apply the rule to discover the degree to which the postconditions of the rule are true

Example:

Given: Prof. Maxwell is *old* with membership 0.4

Rule: If a person is old, they have grey hair

Result: Apply standard material implication with the classical set operators.

- $\mu_A = 0.4$, $\mu_B(g) = \dfrac{1}{1 + e^{-(g + 0.5)}}$

$$A \bullet (A \to B) = \mu_A \wedge ((1 - \mu_A) \vee \mu_B) \tag{11}$$

$$\text{Maxwell has grey hair} = min(0.4, max(0.6, \mu_B(g))) = 0.4 \tag{12}$$

Note: this case is not that interesting because of the use of the material implication definition

In general, the result will be a function of g and represent a membership function over g.

Note: the output is not a single value, but an actual fuzzy set defined over g

- The inference changes the output (B's) membership function

Calculating the result of an inference

Recall that to calculate the result of an inference we need to consider both the truth of the precondition *P* and the relationship specified by the inference $A \to B$:

$$P \bullet (A \to B) = C \tag{13}$$

It is important to note that *P* and *A* do not have to be the same fuzzy set, although they must be fuzzy sets on the same linguistic variable.

Picking some combination of operators for the AND and implication rules we get a wide variety of possible expressions for *C*. Using the material implication and classical definitions, we get the result that:

$$P \bullet (A \to B) = \mu_P \wedge ((1 - \mu_A) \vee \mu_B) \tag{14}$$

Which has the following effects:

- If the predicate P has a value less than the inverse of A, then the membership function C is a constant value (μ_P)
- If the predicate P is greater than the inverse of A, then the membership function C is the function μ_B bounded by μ_P above and $1 - \mu_A$ below.

Unfortunately, this does not make for very interesting consequent membership functions, and can be difficult to evaluate when trying to get a crisp output for a fuzzy controller.

A more interesting combination of operators is the "coupled with" definition for implication and the algebraic product for AND.

$$P \bullet (A \to B) = \mu_P \bullet (\mu_A \mu_B) = (\mu_P \mu_A)\mu_B(z) = C \tag{15}$$

In this case, *C* is simply a scaled version of the output membership function. This turns out to be much more desirable for use in a fuzzy controller because it retains the distribution properties of the output membership functions.

- In most fuzzy controller situations, $\mu_P(x)$ is taken to be a fuzzy singleton (crisp input) in which case the weighting factor is simply $\mu_A(x)$.

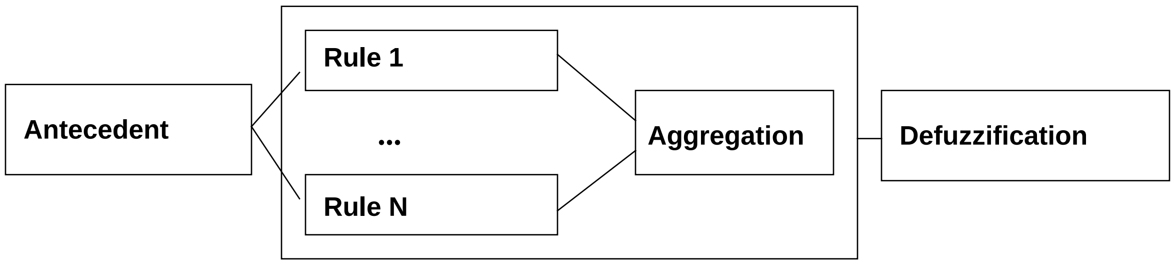
Combining multiple antecedents and multiple rules

When an inference rule has multiple antecedents (e.g. if X and Y or Z then B) these are generally combined using standard fuzzy set combination to obtain a single "activation" value for the rule.

When there are multiple rules that generate consequent fuzzy sets for the same linguistic variable, their outputs are generally combined using an OR operator such as algebraic sum or max.

The whole process of inference has a number of stages, each of which can use different definitions for the standard set of fuzzy operators. For any given fuzzy system that uses implication, the designer must select definitions for each of the following stages:

- Antecedent combination: generally AND, OR, NOT relationships
- Implication: the "coupled with" definition seems to be popular
- Aggregate operator: aggregates consequent membership functions from multiple rules
- Defuzzification operator: transforms a output membership function into a crisp value



Defuzzification

The task of defuzzification is selecting a crisp value given a membership function (or multiple membership functions, in some cases).

So how in the world do we get an answer from a membership function?

- If we use max-product to get the consequent, then we have a distribution function
- If this were a probability distribution, the centroid would give us the expected value

We can calculate the centroid using numerical integration:

$$z_{COA} = \frac{\int_Z \mu_A(z) z \, dz}{\int_Z \mu_A(z) \, dz} \tag{16}$$

This gives us the centroid of the membership function as the most likely value for *z*. This is not always satisfactory (although it is commonly used).

Other possibilities include:

- Mean of the max, or mean of the core
- Area bisector
- Smallest *z* of the core, or largest *z* of the core

A common variation in fuzzy controllers is to use **sum-product composition** for computing fuzzy inferences.

- Use the "coupled-with" and algebraic product definitions for inference

- Use a weighted sum to combine the results of multiple rules

For rule *i*, the weight is the activation of the predicate multiplied by the area of the consequent membership function.

$$z = \frac{w_1 z_1 + \ldots + w_N z_N}{w_1 + \ldots + w_N} \tag{17}$$

$$w_i = \mu_{A_i}(x) \int_Z \mu_{C_i}(z)dz \tag{18}$$

$$z_i = \frac{\int_Z \mu_{C_i}(z) z dz}{\int_Z \mu_{C_i}(z)dz} \tag{19}$$

Note: this does not strictly follow the usual rules for composition (straight summation is not a true fuzzy set operator), but it is a fast way to calculate the outputs.

Variations on fuzzy controllers: Sugeno fuzzy models

What if a particular fuzzy inference rule simply selected a polynomial function of its antecendent arguments:

- if x is A and y is B then z = f(x, y)

This produces a nice crisp output without having to calculate centroids or do any other form of defuzzification. All we need to do is be able to aggregate multiple rules:

- Calculate a weighted average of the outputs based upon the rule activations.
- Could also use a weighted sum if the input membership functions are close to orthogonal

A first-order Sugeno fuzzy model uses linear consequent functions. A zero-order Sugeno fuzzy model uses constant consequent functions.

Issues in fuzzy controllers

Defining the antecedent membership functions is the most important step in designing a fuzzy controller.

- Orthogonality is actually a useful thing to shoot for
- If you have an area of the input space that has low membership values in all fuzzy sets, then you end up with a garbage consequent membership function and incorrect behavior.

Subdividing the input space appropriately is important.

- A uniform subdivision can be prohibitive as the number of input dimensions grows.
- Adaptive subdivision can allow you to cover large portions of the input space with a single fuzzy rule, while still obtaining high resolution where the behavior changes quickly.

Designing a fuzzy controller

Given the above discussion, we can now design a fuzzy controller for a subway.

Linguistic Variable: distance from platform d = m

- Term set: at platform, close, medium, far

- For each of these, define a membership function

Linguistic Variable: s = speed in m/s

- Term set: fast, medium, slow, stopped
- For each of these, define a membership function

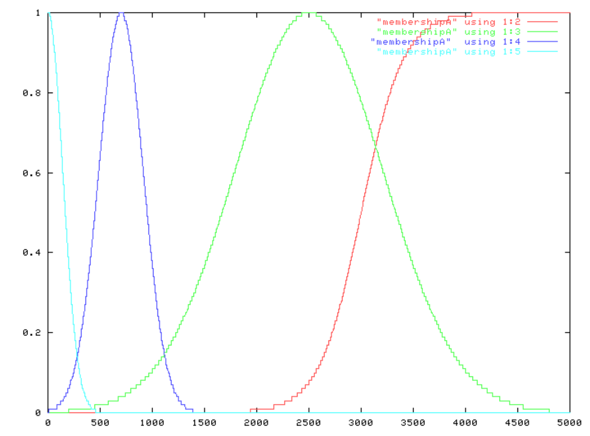
Rules:

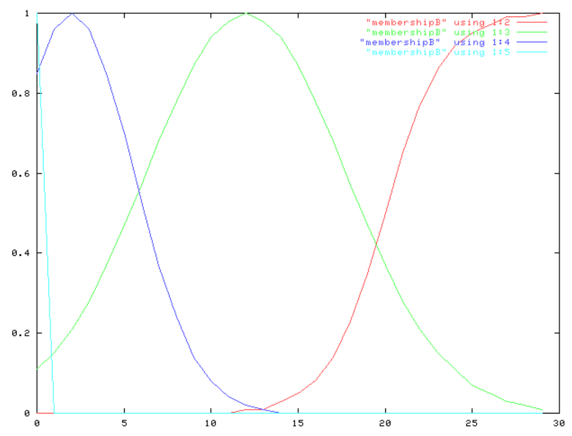
- If the train is far from the platform, then speed should be fast
- If the train is at a medium distance from the platform, then speed should be medium
- If the train is close to the platform, then speed should be slow
- If the train is at the platform, then speed should be stopped

Each of the above rules is active at all times

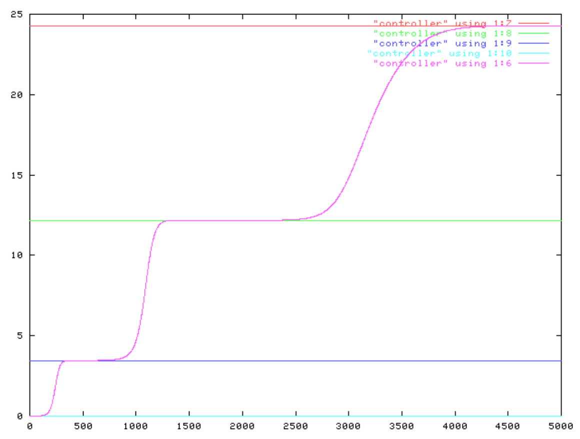
- The consequent membership function is the max of all of the rules
- The current speed is the defuzzified output of the consequence membership function

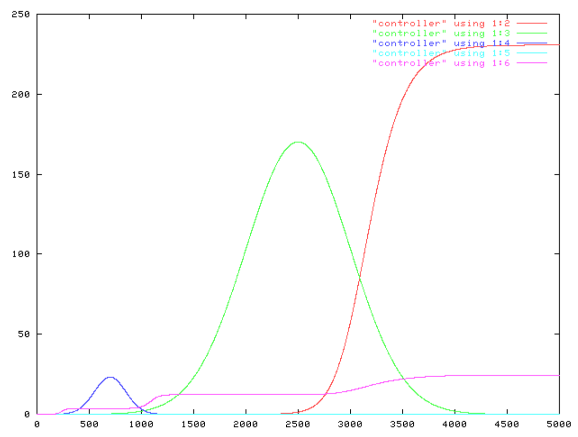
Using the Mamdami shortcut method, we get the following:





Membership functions for the input (left) and out-
put (right). The input variable is distance to the platform, and the fuzzy sets from left to right are (at, close, medium, far). The output variable is speed, and the fuzy sets are (stopped, slow,





Centroids of consequent functions and the final defuzzified output (left) and the activation strengths of the fuzzy rules (right).

E28/CS82 Lecture #16 S05

Overall Topic: Localization & Pose Estimation

Localization & Pose Estimation

Localization is necessary because odometry drifts over time

Methods of localization:

1. Beacon-based localization
   - GPS
   - Ultrasonic beacons
   - Radio beacons
   - Magnetic dipoles
2. Target-based localization
   - Bright color blobs
   - Text
   - P-similar patterns
3. Map-based localization
   - Generally a metric localization strategy
   - Physical features
   - Visual features
4. Topological localization
   - Generally a non-metric localization strategy
   - Map provides connectivity, but not necessarily quantitative relationships
   - Usually a target or physical feature that identifies a topologically important feature

In addition, localization methods fall into two broad categories

1. Single hypothesis estimation
   - Kalman filters
   - Triangulation methods
2. Multiple hypothesis methods
   - Markov chains
   - Particle/Bayesian filters

Kalman Filtering

Kalman filtering is optimal state estimation from noisy measurements

- Each measurement contains some information
- In addition, you may know something about how the state changes
- The better your measurement accuracy, the more you trust it
- The more noise that exists in your process, the more you trust the measurements

State can be anything you are trying to measure, including things you measure indirectly

- Temperature: best estimates of a parameter
- Location: robot localization, GPS navigation systems
- Trajectory: radar and missile tracking

The Kalman filter is an optimal 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)

Setting up the Kalman filter

- A measurement z_k is linearly related to a system state x_k as $z_k = Hx_k + v_k$, where the noise term is a Gaussian with zero mean and covariance R_k $v_k = N(0, R_k)$
- If variations in x and z are Gaussian (normally distributed), then the optimal estimator for the new state $\hat{x}_k(+)$ is also the optimal linear estimator, and the estimate of the new state is a linear combination of an a priori state estimate $\hat{x}_k(-)$ and the measurement z_k.

$$\hat{x}_k(+) = A\hat{x}_k(-) + Bz_k \tag{1}$$

- 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 $w_{k-1} = N(0, Q_k)$. Note that, in the absense of measurement, the error in the system state estimation will grow without bounds at a rate determined by Q_k, the process covariance matrix.

$$x_k = \Phi_{k-1}x_{k-1} + w_{k-1} \tag{2}$$

- The system model 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.

$$\hat{x}_k(-) = \Phi_{k-1}\hat{x}_{k-1}(+) \tag{3}$$

- 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 (3). The new covariance also increments by the process covariance matrix (the inherent error in the state update equation).

$$P_k(-) = \Phi_{k-1}P_{k-1}(+)\Phi_{k-1}^T + Q_{k-1} \tag{4}$$

- 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 (1). The Kalman gain needs to balance 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. The expression $P_k(-)H_k^T$ 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.

$$\overline{K}_k = P_k(-)H_k^T[H_kP_k(-)H_k^T + R_k]^{-1} \tag{5}$$

- 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.

$$P_k(+) = [I - \bar{K}_k H_k] P_k(-) \tag{6}$$

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

$$\hat{x}_k(+) = \hat{x}_k(-) + \bar{K}_k[z_k - H_k \hat{x}_k(-)] \tag{7}$$

Kalman Filtering Summary

Kalman filter variables:

- Φ_k = process matrix at step k
- Q_k = process noise covariance matrix at step k
- H_k = measurement matrix at step k
- R_k = measurement noise covariance matrix at step k
- $\hat{x}_k(+)$ = state estimate at time k
- P_k = *s*ystem error covariance matrix at step k
- $\bar{K}_k$ = Kalman gain estimate at step k

Kalman filter equations:

$$\hat{x}_k(-) = \Phi_{k-1} \hat{x}_{k-1}(+) \tag{8}$$

$$P_k(-) = \Phi_{k-1} P_{k-1}(+) \Phi_{k-1}^T + Q_{k-1} \tag{9}$$

$$\bar{K}_k = P_k(-) H_k^T [H_k P_k(-) H_k^T + R_k]^{-1} \tag{10}$$

$$P_k(+) = [I - \bar{K}_k H_k] P_k(-) \tag{11}$$

$$\hat{x}_k(+) = \hat{x}_k(-) + \bar{K}_k[z_k - H_k \hat{x}_k(-)] \tag{12}$$

E28/CS82 Lecture #17 S05

Overall Topic: Localization & Pose Estimation

Localization Using Kalman Filters

Kalman filtering is an obvious method when you have active beacons that are providing a direct estimate of the robot's position.

For targets, or features you have expected estimates of angles or distances to the targets

- Collect a set of actual sensor observations
- Given the sensors, a map, and the robot's predicted location, predict the observations
- Match up the predicted and actual observations
- Calculate the prediction error, which provides the "innovation"
- Calculate the Kalman gain
- Calculate the update to the robot's position due to the innovation and the Kalman gain

The transformation between the sensors and the robot's position is used to connect the robot's position to the predicted sensor outputs and then move from the innovation back to a position update. Both a general outline and a specific example are given in the book.

Update Loop Details

The update in position for a differential drive robot is, unfortunately non-linear. For velocity ΔV and steering angle $\Delta\theta$ the update equation is:

$$x_{k+1} = \begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = f(x_k, y_k, \theta_k, \Delta V, \Delta\theta) = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} \Delta V \cos\left(\theta + \frac{\Delta\theta}{2}\right) \\ \Delta V \sin\left(\theta + \frac{\Delta\theta}{2}\right) \\ \Delta\theta \end{bmatrix} \qquad (1)$$

To linearize this, we need to create the Jacobian matrix of first derivatives, which is our instantaneous plant matrix for the Kalman filter equations

$$\Phi_k = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial\theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\Delta V \sin\left(\theta + \frac{\Delta\theta}{2}\right) \\ 0 & 1 & \Delta V \cos\left(\theta + \frac{\Delta\theta}{2}\right) \\ 0 & 0 & 1 \end{bmatrix} \qquad (2)$$

In the extended Kalman filter outlined in the book, the true nonlinear update equations are used to get the a priori state estimate, but the linearized plant matrix is used to generate the new error estimate, so that the error update remains a linear matrix multiplication.

However, the error update contains two components: propagated noise, and noise introduced via commands to the robot (process noise). The model makes the further assumption that the noise introduced via robot motion is independent for the motion of the two wheels.

The matrix that relates the noise to the individual wheel motions is derived in the book. However, we do not necessarily have access to the raw wheel values. Instead, we can take the derivative of the update equations with respect to the steering and translation values.

$$J_k = \begin{bmatrix} \frac{\partial f}{\partial \Delta V} & \frac{\partial f}{\partial \Delta \theta} \end{bmatrix} = \begin{bmatrix} \cos\left(\theta + \frac{\Delta\theta}{2}\right) & -\frac{\Delta V}{2}\sin\left(\theta + \frac{\Delta\theta}{2}\right) \\ \sin\left(\theta + \frac{\Delta\theta}{2}\right) & \frac{\Delta V}{2}\cos\left(\theta + \frac{\Delta\theta}{2}\right) \\ 0 & 1 \end{bmatrix} \tag{3}$$

Then the complete a priori update equation for the error is

$$P_{k+1}(+) = \Phi_k P_k(-) \Phi_k^t + J_k \Sigma_\Delta J_k^t \tag{4}$$

where the errors in the translation and steering commands (Σ_Δ) are assumed to be independent.

$$\Sigma_\Delta = \begin{bmatrix} K_1 |\Delta V| & 0 \\ 0 & K_2 |\Delta\theta| \end{bmatrix} \tag{5}$$

Note that because the modifier matrices are not diagonal, P, in general, will not be diagonal

- P can be initialized to zero, because any motion of the robot will make P non-zero
- Use matrix multiplication to update P

E28/CS82 Lecture #18 S05

Overall Topic: Localization & Pose Estimation

Localization Using Kalman Filters

Matching observations and predictions

Consider the case of localization using line features

- Actual line features are extracted from the actual sensor readings
  - Some true line features may not be visible or may not be detected
- A set of potential predicted line features are extracted from the map in an area around the robot’s predicted location
  - There will often be more predicted features than are actually detected
  - The prediction is based on the a priori state update results (where the robot is now)

The robot has to match the predicted and actual features into pairs

- For each predicted measurement, calculate how close each actual measurement is
  - Generates an N x M matrix, N actual features, M predicted features
  - Each element in the matrix indicates distance between actual and predicted feature
  - Generally a multi-dimensional Gaussian (Mahalanobis distance)

$$D_M(x_1, x_2, \Sigma) = (x_1 - x_2)\Sigma^{-1}(x_1 - x_2) \qquad (1)$$

- Generate a validation gate
  - Generally a threshold for the Gaussian/Mahalanobis model
  - Can be loose or tight, depending upon the number of predicted features
  - Measurements outside the validation gate for a predicted measurement are ignored

Given the validation gate for each predicted measurement, there are three cases

- If only one actual measurement is in the validation gate, it becomes a match
- If many actual measurements are in the validation gate, pick the best one
  - May have to do some kind of optimization
- There may be actual measurements not in any predicted measurement’s validation gate
  - Not used for localization
  - May belong to a new object in the environment, in which case the robot can use the measurement to update the map.

Updating the Prediction (a posteriori update)

Kalman filter determination is based on the quality of the match

- Innovation matrix is the denominator of the Kalman gain equation

$$\Sigma_{\mathrm{IN}} = \nabla h_{k+1} P_{k+1}(+) \nabla h_{k+1}^t + R_{k+1} \qquad (2)$$

- If the sensor/state relationship is non-linear, H is the Jacobian of the relationship
  - R is the inherent noise in the sensor system
- Have to take the inverse of the innovation matrix (since it’s a denominator)

Example: Kalman filter localization using sonar

Standard update equations for a differential drive robot.

$$\Delta s_r = \frac{2\Delta s + \Delta\theta b}{2} \qquad \Delta s_l = \frac{2\Delta s - \Delta\theta b}{2} \tag{3}$$

System covariance specifies likely positions

- Sample from the 3D Gaussian distribution (position + orientation)
- For each sample, calculate the 16 sonar values from that position
  - Intersect with all obstacles on the map (sample the cone space)
  - Match with actual sonar readings using L2-norm (sum-squared distance)
  - Calculate the average L2-norm error per sonar, proportional to error in measurement
- Keep the best sample (could also do a weighted average if it were truly Gaussian; it’s not)
  - The degree of the match is related to the variance of the noise in the measurement
  - Note we have state directly from this method (H is the identity matrix)
- Gate the sample, so we don’t do a closed loop update if the match is too bad
- If the sample is sufficiently good:
  - Use the best sample to calculate the Kalman gain
  - Update the state
  - Update the probability matrix
- Otherwise, the error increases, and the next time we sample more widely

Details:

- Quality of the map and simulation of the sensors is very important
- The angular measurement error is not well represented by the sonar SSD
- Balancing the covariance matrix after each iteration is also important
  - Has to be a symmetric matrix
  - Small errors creep in over time and multiply in the iterations
- The covariance terms of the Kalman gain cause bad effects, I’m only using the digaonals

E28/CS82 Lecture #19 S05

Overall Topic: Localization & Pose Estimation

Particle Systems: Multiple Hypothesis Localization

Particle filters are a way of creating and representing probability distributions from measurements

- Let $Z_k = (z_1, z_2, \ldots, z_k)$ be a set of measurements
- Let X be a state vector
- We would like to know $P(X|Z_t)$, which is the probability of the state X given the history of measurements Z_t.
- Knowing this function lets us select high probability states or evaluate existing estimates

But the function $P(X|Z_t)$ is not necessarily nice to calculate because you have to know the true state X in order to know the probability of it given a set of measurements.

Bayes rule offers us a potential way out:

$$P(x_k|Z_k) = \frac{p(z_k|x_k)p(x_k|Z_{k-1})}{p(z_k|Z_{k-1})} \tag{1}$$

This says that we can define the probability of a state x_k given three things:

- the best estimate of the current state given all but the last sensor reading
- the probability of the kth sensor reading z_k if the state were in state x_k
- the probability of the last sensor reading z_k given all the previous sensor readings

Note that these are three things that we can actually calculate, and the denominator is generally assumed to be constant.

- We can always renormalize the probability distribution to sum to 1

So what we really need to know are how to represent and calculate the numerator in (1).

- Think like a Kalman Filter: open loop update phase, closed loop measurement phase

Given:

A set of samples $\{(s^0_{k-1}, \pi^0_{k-1}), \ldots(s^N_{k-1}, \pi^N_{k-1})\}$ that consist of pairs of state vectors and weights drawn from a probability density function representing $p(s^i_{k-1} = x_i|z_0, \ldots z_{k-1})$

Prediction phase:

For each sample (s^i_{k-1}, π^i_{k-1}), calculate a new state s^{i-}_k using a motion model, which may just be a simple noise model for the process.

This generates a new set of particles that represents the distribution of next state locations. This represents the function $p(x_k|Z_{k-1})$

Update phase:

Weight the new samples according to the current measurement z_k.

$$\pi_k^{i+} = p(z_k | x_k) \tag{2}$$

Re-sample phase

If we just stopped there, our samples would generally just get less and less likely because we wouldn't necessarily be sending samples towards high probability locations (which is where the interesting information really sits).

So next we generate a new set of samples $\{(s_k^{0+}, \pi_k^{0+}), \ldots, (s_k^{N+}, \pi_k^{N+})\}$ by re-sampling the a priori set of samples according to the π_k^{i+}.

- High probability samples will get selected multiple times
- Low probability samples will eventually be dropped
- The weights need to be re-normalized to sum to one: $\sum_N \pi_k^{i+} = 1$

Summary:

1. Given: a set of samples $\{(s_{k-1}^0, \pi_{k-1}^0), \ldots (s_{k-1}^N, \pi_{k-1}^N)\}$
2. For each sample (s_{k-1}^i, π_{k-1}^i), create a new state s_k^{i-} using a probabilistic motion model.
3. Weight the new samples according to the current measurement: $\pi_k^{i+} = p(z_k | x_k)$
4. Generate a new set of samples $\{(s_k^{0+}, \pi_k^{0+}), \ldots, (s_k^{N+}, \pi_k^{N+})\}$ by re-sampling using the π_k^{i+}.
5. Re-normalize the weights to sum to one: $\sum_N \pi_k^{i+} = 1$
6. Repeat for each new measurement.

Modifications to the basic particle filter

Order of operation

- It turns out that you can sample based on the π_k^{i+} or you can sample based on the π_{k-1}^{i+}

Either method seems to work, but drawing from π_{k-1}^{i+} actually lets you draw samples from different distributions

- Sometimes you just want to insert random particles into the system drawn from $p(x_k)$
- Sometimes you know something about where particles ought to be

Importance sampling

- Sometimes you know something about where the important parts of the distribution are
- If you are tracking, sometimes you have new targets enter your measurement space
  - Some particles will eventually find the new target, but it can be slow to respond
  - You would like some particles to be attracted to the indication of a new target

Importance sampling lets you draw some of the particles from a function g(x)

- Need to correct for the fact that g(x) is not f(x)

$$\pi_k^{i+} = \lambda_k^i p(z_k|s_k^i) = \frac{f_k(s_k^i)}{g_k(s_k^i)} p(z_k|s_k^i) = \frac{\sum_{j=1}^{N} \pi_{k-1}^j p(s_k^i|s_{k-1}^j)}{g_k(s_k^i)} p(z_k|s_k^i) \tag{3}$$

- The summation is simply the Monte Carlo Integration of the previous probability distribution representing likely states. It is the total probability of s_k^i given the probability distribution of the state vectors at the previous time step.
- If f(x) is equal to g(x), then the correction factor is 1.
- The particle weight gets reduced is we “unfairly” selected it: g(x) > f(x)
- The particle weight gets increased if it squeaked in: g(x) < f(x)

Linear Selection Algorithm

The simple way of selecting a next generation is O(N^2).
- Pick a random number
- Sum the weights of the particles until you reach the number.

This requires N summations of N/2 (on average) particles to select the next generation.

A more clever method of sampling is only O(N)
- Pick a random number
- Sum the weights of the particles until you reach the number
- Select that particle
- Add 1/N to the random number
- Repeat from step 2

The algorithm samples from the weight space uniformly and regularly, but by aligning the grid at a random location you avoid aliasing, because any particle has a probability of landing on the grid that is proportional to its weight. The algorithm is order N because you pass through the particles at most twice (on average, 1.5 times).

Avoiding the O(N^2) sampling time makes each iteration of the algorithm linear in the number of particles.

E28/CS82 Lecture #20 S05

Overall Topic: Particle System Localization, Simultaneous Localization and Mapping

Localization Using Particle Filters

Particle system localization has worked well in practice on numerous systems.

- Robust to noise in the system
- Robust to divergence in the state tracks (unlike a Kalman Filter)
- Fast

Example 1: Ceiling Tracking

Dellaert, Fox, Burgard, Thrun, “Monte Carlo Localization for Robots”, *ICRA’99*, May 1999.

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

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

Had the robot move 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

Example 2: Sonar Profile

Given a map of polygonal obstacles

While the robot is moving

- Generate new particles from the motion model (no linearization necessary)
- For each particle
  - Estimate the sonar hits from the robot’s estimated pose
  - Calculate the SSE between the estimated and actual sonar readings
  - Weight by a Gaussian based on the average SSE (σ is an acceptable average SSE)

$$\pi_i = e^{-\left(\frac{E}{\sigma^2}\right)} \tag{1}$$

- Resample the particles based on the weights
- Most likely particle is returned as the robot’s location
  - Weighted average location (not good for divergent situations)
  - Particle density + weight (selects denser and more likely of two peaks)
  - Mean shift with weighting (similar strategy)

Getting information out of a particle system can be harder than designing and iterating it

Simultaneous Localization and Mapping

First paper on SLAM is Smith and Cheeseman 1987.

Original method was based on Kalman filtering

- Robot pose is represented as part of the state vector (x, y, θ)
- Obstacle poses are also represented as part of the state vector (x, y)
- Robot pose and the obstacle poses are all related through the covariance matrix
  - Updates are quadratic in the number of obstacles (hard limit in number of obstacles)
- Data association is required at each step
  - Recovery is unlikely after divergence, and difficult to detect

Research on SLAM progressed slowly until 2002, when FastSLAM was proposed

Montemerlo, Thrun, Koller, Wegbreit, "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem", *AAAI02*, July, 2002.

Basic idea is that you can separate the landmark estimates from one another, making them only dependent upon the robot's path.

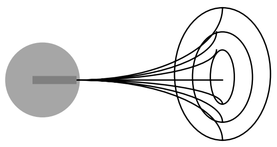
Pose and map representation

- Each particle represents the robot's pose (x, y, α)
- Each particle includes all of the found landmarks so far (a complete map) $\{(\theta_x^i, \theta_y^i, \Sigma^i)\}$
  - The landmark state is a mean and a covariance (Kalman filter state)

Process

1. Robot motion update

Select a new robot location given the motor commands



2. Landmark position update
   - If the landmark was observed
   - Use an EKF to update the landmark position

$$p((\theta_{i = n_t} | (s^t, z^t, u^t, n^t)) \propto p(z^t | (\theta_{n_t}, s^t, n^t)) p(\theta_{n_t} | (s^{t-1}, z^{t-1}, u^{t-1}, n^{t-1}))) \tag{2}$$

3. Calculate the importance weights for each particle
   - Weight is proportional to the combined probability of seeing the observation for each landmark and the probability of the landmark being there

$$w_t = \sum_{\theta} p(z^t | (\theta_i^t, s^t)) p(\theta_i^t) \tag{3}$$

4. Resample the particles using the weights calculated in step 3

E28/CS82 Lecture #21 S05

Overall Topic: Simultaneous Localization and Mapping

Simultaneous Localization and Mapping

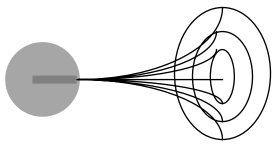
Review of the process:

- Each particle represents a robot pose and a map (set of landmarks, or evidence grid)

$$q_i = \{x_r, y_r, \theta_r, (\mu_1, \Sigma_1), \ldots, (\mu_N, \Sigma_N)\} \tag{1}$$

1. Robot motion update

Select a new robot location given the motor commands



2. Landmark position update
   - If the landmark was observed
   - Use an EKF to update the landmark position

$$p((\theta_{i = n_t} | (s^t, z^t, u^t, n^t)) \propto p(z^t | (\theta_{n_t}, s^t, n^t)) p(\theta_{n_t} | (s^{t-1}, z^{t-1}, u^{t-1}, n^{t-1}))) \tag{2}$$

Specifically, let each landmark θ_n be represented by a 2D global location and 2x2 covariance matrix.

$$\mu_n = (x, y) \qquad \Sigma_n = \begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y \\ \sigma_x\sigma_y & \sigma_y^2 \end{bmatrix} \tag{3}$$

Consider the case where the measurement z_t is local estimate of the landmark in the robot's coordinate system. Then we can write the Kalman state μ_n in terms of the robot's state and the local measurement.

$$\mu_{n^t} = \begin{bmatrix} x_r \\ y_r \end{bmatrix} + \begin{bmatrix} \cos\theta_r & -\sin\theta_r \\ \sin\theta_r & \cos\theta_r \end{bmatrix} \begin{bmatrix} z_x \\ z_y \end{bmatrix} \tag{4}$$

This is a nonlinear equation, so we need to calculate the Jacobian with respect to errors in the measurement values, since we are assuming the robot position is correct.

$$G = \begin{bmatrix} \cos\theta_r & -\sin\theta_r \\ \sin\theta_r & \cos\theta_r \end{bmatrix} \tag{5}$$

The matrix G is then used in place of the measurement matrix H in the Kalman update equations. The plant matrix for the landmarks should be the identity matrix (landmarks don't move), which makes the Kalman update equations straightforward.

$$K_t = \Sigma_{n^t}(-)G_t(G_t^T\Sigma_{n^t}(-)G_t + R_t)^{-1} \quad (6)$$

$$\mu_{n^t}(+) = \mu_{n^t}(-) + K_t(z_t - \hat{z}_t)^T \quad (7)$$

$$\Sigma_{n^t}(+) = (I - K_t G_t^T)\Sigma_{n^t}(-) \quad (8)$$

$$\hat{z}_t = \begin{bmatrix} \cos\theta_r & \sin\theta_r \\ -\sin\theta_r & \cos\theta_r \end{bmatrix}\left(\begin{bmatrix} \mu_{nx} \\ \mu_{ny} \end{bmatrix} - \begin{bmatrix} x_r \\ y_y \end{bmatrix}\right) \quad (9)$$

The last equation is the predicted measurement given a feature and the robot's pose. It is just the inverse of (4).

3. Calculate the importance weights for each particle
   - Weight is proportional to the combined probability of seeing the observation for each landmark and the probability of the landmark being there

$$w_t = \sum_{\theta} p(z^t | (\theta_i^t, s^t)) p(\theta_i^t) \quad (10)$$

When the whole update is executed for a single sensor measurement, this reduces to the evaluation of a single Gaussian.

$$w_t \propto |2\pi Q_t|^{-\frac{1}{2}} e^{-\frac{1}{2}(z_t - \hat{z}_t)Q_t(z_t - \hat{z}_t)} \quad (11)$$

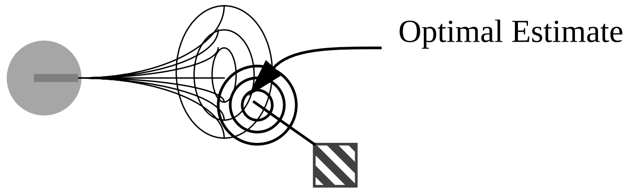
$$Q_t = G_t^T\Sigma_{n^t}(-)G_t + R_t \quad (12)$$

When the update is executed for multiple sensor readings, the weight would be their sum.

4. Resample the particles using the weights calculated in step 3

FastSLAM 2.0: Provably Convergent FastSLAM

FastSLAM 2.0 is a slight modification of the original FastSLAM algorithm. The primary modification is the stage one update of the robot's state vector
- In FastSLAM 1.0, the state vector is updated only by the motion model
- In FastSLAM 2.0, the state vector is updated conditioned on both the motion model and the current sensor reading



It turns out this is a hard thing to do, since both the motion model and the measurement model may be non-linear and the integral of their combination is not necessarily calculable in closed form.

The solution is to linearize the motion model, as we did in FastSLAM 1.0.

- Through a series of steps, the result can be approximated as a Gaussian distribution
- The mean and covariance matrix are calculable from known parameters
- The new robot location is sample from the Gaussian approximation

Step 2 (landmark estimation) remains the same in FastSLAM 2.0 ans in FastSLAM 1.0.

Step 3 is slightly different, in that the normalization factors need to be treated differently, since we updated the particles differently in step 1. However, the same process applies, and the result is a single Gaussian approximation to the non-linear target distribution.

Note: sampling from the advanced distribution will work for both localization and SLAM problems, and would likely result in similar improvements in performance on the simpler task of localization within a known environment.

Comparisons between FastSLAM 1.0, FastSLAM 2.0, and EKF

The major advantage of FastSLAM (both versions) is their speed and ability to scale to arbitrary map sizes. EKF-based SLAM is slow to update and is not scalable in its simple form.

A second advantage is that FastSLAM 1.0, at least, is simple to implement (no more difficult than a particle filter plus a Kalman filter).

A third advantage is that data association is simpler with FastSLAM than with EKF because you don’t have to be perfect all the time.

In terms of performance on typical tasks, FastSLAM 1.0 requires more particles than FastSLAM 2.0 to get the same results, and cannot be proven to converge to a single solution. FastSLAM 2.0 under certain restrictions can be proven to converge, as can the EKF method.

FastSLAM 2.0 with a single particle performs almost as well as the EKF method on real tasks (and is roughly equivalent mathematically, although much faster).

Where you see the effect of making the landmarks independent is when the robot makes loops that allow it to visualize previously seen landmarks.

- EKF immediately reduces the error in both the robot and the landmarks upon seeing and correctly matching a previous landmark
  - The covariances actually reduce the error in all landmarks simultaneously
- The particle filter can only reach back in time so far as it has differences between particles
  - Only landmarks that are visible can have their locations improved
  - If there are particles with different landmark locations in the past, resampling can improve prior landmark estimates by eliminating poor estimates
- FastSLAM 1.0 with more particles can go farther back in time, in general
- FastSLAM 2.0 needs to loop around a lot to keep reducing the error in landmark locations

E28/CS82 Lecture #22 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision

What is an image?

- 1 or 3 arrays of type unsigned char, or some variation thereof
- Each array element represents an intensity value [0, 255]
- Color images consist of 3 arrays: red, green, and blue
- Often, the RGB values are interleaved so each pixel is 24 bits (or 32 with padding)
    - White = [255, 255, 255]
    - Black = [0, 0, 0]
    - Bright blue = [0, 0, 255]
- Note: humans use a log scale, and some CV algorithms do to: log(intensity)

How do we define color?

- RGB: response of three characteristic filters to different areas of the visible color space
- Chromaticity, or normalized color

$$\{r, g, b\} = \frac{\{R, G, B\}}{R + G + B} \tag{1}$$

- YUV/YIQ is the NTSC (US broadcast) standard, some framegrabbers provide it natively

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \qquad \begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ -0.30 & -0.59 & 0.383 \\ 0.577 & -0.59 & -0.11 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.105 & 1.702 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \tag{3}$$

- Hue [of HSI]: angle on a color wheel

$$I = \frac{1}{3}(R + B + B) \qquad S = 1 - \frac{3}{R + G + B}[\min(R, G, B)] \tag{4}$$

$$H = \operatorname{acos}\left\{ \frac{\frac{1}{2}[(R - G) + (R - B)]}{[(R - G)^2 + (R - B)G - B]^{\frac{1}{2}}} \right\} \tag{5}$$

- Ohta color space (opponent color space)

$$O_1 = \frac{1}{3}(R + G + B) \quad O_2 = R - B \quad O_3 = \frac{1}{2}(2G - R - B) \tag{6}$$

- Other color spaces, but not particularly useful in real time

E28/CS82 Lecture #23 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision

Camera technology: CCD & CMOS

- CCD: higher resolution spatially and spectrally (maximizes sensor area)
- CMOS: cheaper, and up to 1/100th the power consumption, less area per pixel for sensor
- Many cameras interleave the odd and even rows of the image (60Hz sampling)
  - Things like gradient operators must take this into account
  - Calculate odd and even gradients, not adjacent rows
- Progressive scan: grabs all of the pixels at the same time (30Hz sampling)

Camera parameters:

- Color balance: Auto, color temperature, manual adjust
- Gain: how much the signal is boosted by a multiplier (multiplies noise as well)
  - Cameras often have automatic gain control, sensitive to maxima or averages
- Black Level: some cameras allow you to manually specify a black level
  - Minimum number of photons you need to have an intensity value > 0
- Shutter speed: how long the CCD integrates the incoming light
- Aperture: how much light is let into the camera
- Focus/Zoom: where the camera is focused and the overall focal length
  - Cameras with auto-focus or zoom are changing the focal length of the camera

How does the world map onto an image?

- orthographic projection: x = X
  - Useful for situations where the distance to the object is large compared with the width of the object $\Delta Z \ll Z$
- weak perspective: x = kX
  - k is a scale factor that depends upon the distance and size of the object
- perspective projection: $x = \frac{-fX}{Z}$
  - The way the world really works
  - Something that is twice as big and twice as far away looks just the same

How do we get the image onto a computer

- Some cameras are digital and send the data to the computer digitally (QuickCam)
  - Query a port (parallel port/USB port/parallel port) and get the data
- Framegrabber: needed for cameras with analog outputs
  - Usually a PCI/PC104 card
  - Formats: NTSC/ PAL/ S-video
  - Usually 30fps is the maximum achievable rate (NTSC signal)
  - Framegrabber dumps the image to some specified memory
    - Usually handled by a driver program
- Firewire or Camera Link
  - Faster digital protocols that are permitting much higher speed frame rates

Quick and Dirty Operations

- Thresholding (binarization)
  - Point-wise operation
  - Pick a lower bound in one or more color bands
  - Pick an upper bound in one or more color bands
  - Any pixel within the bounds -> 1
  - Any pixel outside the bounds -> 0
- Picking a threshold
  - Manually select one by looking at histograms
  - Train it using a set of training data and a search algorithm (Linear, GA)
    - Linear search: iterate over each variable, holding the rest constant
    - GA: set up a GA to optimally select the parameters
  - ISODATA
    - Pick a threshold
    - Calculate the weighted average Alow of the pixel values below the threshold
    - Calculate the weighted average Ahigh of the pixel values above the threshold
    - If the threshold is equal to (Alow + Ahigh)/2 then stop
    - Otherwise, make the threshold (Alow + Ahigh)/2 and repeat from step 2

E28/CS82 Lecture #24 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Quick and Dirty Operations

Pixel connectedness

- Which pixels are neighbors?
- Answer is either 4-connected or 8-connected pixels
  - 4-connected regions can have 8-connected boundaries
  - 8-connected regions must have 4-connected boundaries

Grouping: two-pass connected component labeling

- Define connected: 4 or 8 connected neighbors
- Set the region counter to 0
- Pass 1: look up and back (previously visited pixels)
  - If the pixel is not touching anything, assign it the value of the counter++
  - Else assign it the region ID of the lowest valued neighbor
  - If two neighbors have different region IDs, record that
- Pass 2: fix the region ID values
  - Give each pixel its true region ID

Grouping: region growing connecte component labeling

```
RegionLabel = 1;
Label the first "on" pixel
Push the first "on" pixel on the stack
While the stack is not empty
      Pop the top pixel off the stack
      For each of its neighbors (4 or 8 conn)
            If the pixel is in the image, "on" and not labeled
                  label the pixel
                  push the pixel on the stack
end while
```

Region Properties

$$\text{Moment: } M_{pq} = \sum_X \sum_Y x^p y^q f(x, y) \tag{1}$$

$$\text{Size: } M_{00} = \sum_X \sum_Y f(x, y) \tag{2}$$

$$\text{Position (1st moment)} \quad x_c = \frac{M_{10}}{M_{00}} \qquad y_c = \frac{M_{01}}{M_{00}} \tag{3}$$

$$\text{Central Moment } (pq \text{ order}) \quad \mu_{pq} = \sum_X \sum_Y (x - x_c)^p (y - y_c)^q f(x, y) \tag{4}$$

Moments about the central axis

- To calculate a moment about the central axis of the object, we need orientation

$$\text{2nd order row moment } \mu_{02} = \frac{1}{A} \sum_Y (y - y_c)^2 f(x, y) \tag{5}$$

$$\text{2nd order column moment: } \mu_{20} = \frac{1}{A} \sum_X (x - x_c)^2 f(x, y) \tag{6}$$

The angle of the axis with the least central moment (central axis) is given by:

$$\alpha = \frac{1}{2} \operatorname{atan}\left[\frac{2\mu_{11}}{\mu_{02} - \mu_{20}}\right] \text{ (Note that you must check the denominator for a zero case)} \tag{7}$$

Then you can get the second moment about the central axis using:

$$\mu_{\bar{r}, \bar{c}, \bar{\alpha}} = \frac{1}{A} \sum_X \sum_Y ((y - \bar{y}) \cos\beta + (x - \bar{x}) \sin\beta)^2, \text{ where } \beta = \alpha + 90° \tag{8}$$

Bounding box: minimum size box (orientated with x-y axes) that contains the region

- %filled: how much of the bounding box is filled
- For a rotation invariant measure, orient the bounding box along the major/minor axes

Projections

Axial projections

- X-projection: build a histogram of the # of pixels in each row
- Y-projection: build a histogram of the # of pixels in each column

Central axis-projections: do the projections along the major and minor axes

- These should be rotation invariant

Radial projections

- Starting at the major axis, for each angular bin, count the # of pixels

Scaling projections by the # of pixels makes them more scale invariant

- Always have to balance invariance with discrimination
- Sometimes size is a good discriminator, while in other cases it separates an object class

Morphological filters and binary operations:

Apply masks to ’on’ pixels in the image to turn on or off other pixels

- Dilation: for each "on" pixel, **or** the structural element with the result (aka “growing”)

$$B \oplus S = \bigcup_{b \in B} S_b \tag{9}$$

- Erosion: keep "on" pixels for which the mask covers only "on" pixels (aka “shrinking”)

$$B \otimes S = \{b | b + s \in B \quad \forall (s \in X)\} \tag{10}$$

- Closing: dilation, followed by erosion

$$B \bullet S = (B \oplus S) \otimes S \tag{11}$$

- Opening: erosion, followed by dilation

$$B \cdot S = (B \otimes S) \oplus S \tag{12}$$

If you use a box (8 connected) or a cross (4 connected) to grow, use the opposite to shrink.

E28/CS82 Lecture #25 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Quick and Dirty Operations

Thinning

- Want to reduce the "thickness" of regions without harming connectedness

```
While the last loop removed a pixel
   Increment the current direction {N, W, E, W}
   For each "off" pixel touching an "on" pixel in the current direction
      If the pixel does not break or shorten a line segment, eliminate it
```

Grassfire transform

- Useful for doing multiple grows or shrinks in a row
- Initialize the background to zero and foreground to large numbers
- Pass from upper left to bottom right: look up and back, pick the lowest # and add 1
- Pass from bottom right to upper left: look down and right, pick the lowest # and add 1, but only replace the value if it’s lower than the current value

Template Matching

- Get an example image of what you want to find
- Move the sample over the image and calculate the convolution with the image
  - For a small template:
    - implement as sum-of-squared-differences in image space
    - Can use mean-subtracted, variance normalized data
  - For a large template:
    - Take the Fast Fourier Transform of the image and the filter
    - Multiply them in frequency domain
    - Take the inverse Fourier Transform to get the result of the convolution

Histogram-based filtering

Generate a fuzzy histogram from training data
- Cut out lots of images of stuff you want to find
- Generate a cumulative histogram
- Divide by the largest element in the histogram -> fuzzy histograms

Point-wise application
- Give each pixel its membership value in the fuzzy histogram
- Option1: take an alpha-cut of the fuzzy values to get a binarized image
- Option 2: deal directly with the probabilities with a box filter

The result is either a binary image or a greyscale image where intensity is the membership value
- If you make it a binary image you can do tracking of color blobs after region growing
- If you make it a probability image you can do particle filter tracking (easily)

E28/CS82 Lecture #26 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Quick and Dirty Operations

Finding Edges

Canny Process

- Run a Sobel over the image (3x3, or 5x5)
  - Executes both smoothing and gradient
  - Run in both X and Y orientations
  - Use combinations of 1-D versions [-1 0 1] x $[1\ 2\ 1]^t$
- Threshold in the gradient direction
  - Only keep maxima in the gradient direction (non-maxima suppression)
- Thinning + Chain code or Hough Transforms

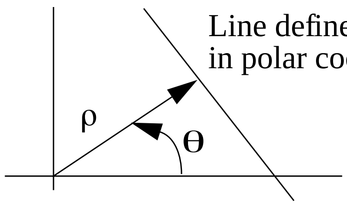
3x3 Sobel
X Direction
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Hough Transforms

1. Quantize the parameter space appropriately
   - divide theta into 360 buckets
   - divide rho into half the diagonal size of the image, assuming the (0,0) point is the center
2. Treat each point in the parameter space as an accumulator, initialized to zero
3. For each edge point (x,y), increment each accumulator that satisfies the equation
   - As a speedup, only increment buckets that fit with the gradient direction of that point
4. Look for maxima in the parameter space to find the parameters of the most likely models.

Looking for lines: use polar coordinates to represent lines

- $\rho/b = \cos(90 - \Theta) = \sin(\Theta)$
- $m = dy / dx = -\cot(\Theta) = -\cos(\Theta) / \sin(\Theta)$
- $y = mx + b = -\cos(\Theta)/\sin(\Theta) * x + \rho / \sin(\Theta)$
- $\rho = x\cos\theta + y\sin\theta$



Finding maxima in the accumulator for the Hough transform

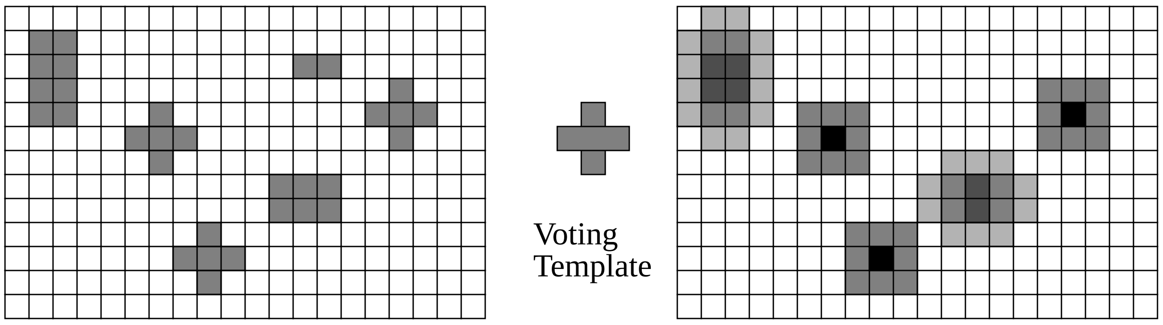
- Can't just pick the top N numbers: might get the same line 1-2 times
  - Find the maximum value and return the line corresponding to that bin.
  - Suppress the values in a neighborhood around it
  - Repeat N times, where N is the number of lines you want

Finding line segments from the Hough Transform

- If you have a line, now you need to find the pixels that are part of that line
- Find one intersection of the line with the edge of the image
- Walk along the line
  - At each row/column collect any pixels that are within distance d of the line
  - Keep track of contiguous segments along the line
- End result is one or more line segments corresponding to that geometric line

Use the Hough transform to find certain shapes in the environment

Example: Let the Hough parameters be x and y location of a cross target



Motion Detection

Differential motion analysis

- Subtract the gray/color values for two consecutive images
- Create a binary mask based on the magnitude of the difference

There are five possible explanations for image differences

- The pixel is a on a moving object in the first image, and part of the background in the second
- The pixel is part of the background in the first image, and on a moving object in the second
- The pixel is on a moving object in the first, and part of a different moving object in the second
- The pixel is on a moving object in the first, and on a different part of the same object in the second
- The difference is due to noise or slight camera motion

A threshold helps us to remove differences due to the last item

Using more than one image can help us to better identify what is happening

- Set up a cumulative difference mask
- $d_{cum}(i, j) = \sum_{k=1}^{N} a_k |f_1(i, j) - f_2(i, j)|$
- The a_k can indicate the significance of a frame (more recent = more significant)

If you can take a static image of the scene’s background, then you can use it as a reference image

- Any difference between the background an the current frame indicates a non-background object/thing

Can you build a background scene even when there are moving objects in it?

- You could look at the d_{cum} values over windows of time. Any time a d_{cum} was less than a threshold for a minimum window of time, note that pixel as a background pixel.
- Keep watching the scene until you have background pixel estimates for the entire image

Building estimates of motion trajectory are difficult using motion differencing

- For some objects, only the edges of the objects appear in difference images, unless you have a good reference background

Differential edge motion analysis

Sometimes straight differencing causes too much noise in the system

- Light sources can appear to pulsate (60Hz, sampled at 30Hz)
- You get a lot of holes in surfaces that have uniform intensity

Taking the difference of edge images can be more robust to noise in the system

- $M(i, j) = E_2(i, j)E_2(i, j) - E_2(i, j)E_1(i, j)$
- The E_i are the edge images
- A positive value indicates edge points in the second image that are not in the first
- A zero value indicates edge points that are the same in both images
- On a continuous edge image, a negative value would be edges that were stronger in the first image, but still existed somewhat in the second

E28/CS82 Lecture #27 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Motion and Stereo

Heuristic Motion Detection

- Divide the image into small blocks
- Figure out what the best new location of the block is using correlation
    - Move the block around its current location and test lots of positions
    - Select the position with minimum error correlation--so long as it's under a threshold
- (New location - Old location) gives an estimate of the magnitude and direction of motion
- This is an approximate method of determing ***optical flow***

Optical Flow

Optical flow is a basic staple of computer vision

- Given: two or more images in time sequence
- Question: where did each pixel move to from one image to the next?
- Answer: a velocity field that represents motion in the scene over time

Almost all optical flow techniques are based on the assumption that intensity is conserved

- Pixel intensities in the world don't change, they just move around
- A second implicit assumption is that pixels don't move very far between frames

For each pixel, we can express this as the **Optical Flow Constraint Equation**

- $\frac{d}{dt}g(x, y, t) = g_x u + g_y v + g_t = 0$ or $-g_t = g_x u + g_y v = \nabla g \cdot \vec{v}$
- This says that the change in overall intensity of the image (g) over time is zero
- gx is the gradient in the x direction
- gy is the gradient in the y direction
- gt is the gradient in time
- (u, v) is the motion of a given pixel between two images
- All three can be calculated using, for example, Sobels in both time and space

The Optical Flow Constraint Equation gives us one equation for each pixel in the image

- We want a 2D velocity at each point (u, v) so we need a second constraint for each pixel

Usually the second constraint is a constraint on the smoothness of the resulting velocity field

- Velocities of adjacent points should not change quickly
- This can be expressed as a constraint on the second derivative of the image
    - A very noisy computation
- This can also be expressed as a constraint on the gradient of the velocity field (still noisy)
    - We want to minimize the expression: $E = \int_D ((\nabla g \cdot v + g_t)^2 + \lambda^2(\|\nabla u\|^2 + \|\nabla v\|^2))dx \backslash$
    - The integral is over the image (approximate as a sum over the pixels)
    - This says that you are trying to both fit the OFCE and minimize the magnitude of the changes in velocity over the flow field.
    - Lamda indicates the importance of the smoothness term, and a good value is 0.5

Example: Horn & Schunk algorithm

1. Initialize all velocity vectors c(i, j) = 0 for all pixels (i, j)
2. Letting k indicate the number of iterations, compute c(i, j) = (u_{ij}, v_{ij}) for all pixels (i, j)
   - $u^k(i, j) = \bar{u}^{k-1}(i, j) - f_x(i, j)\dfrac{P(i, j)}{D(i, j)}$
   - $v^k(i, j) = \bar{v}^{k-1}(i, j) - f_y(i, j)\dfrac{P(i, j)}{D(i, j)}$
   - $P = f_x\bar{u} + f_y\bar{v} + f_t$
   - $D = \lambda^2 + f_x^2 + f_y^2$
   - $(\bar{u}, \bar{v})$ are the mean values of the velocity in a local area around (i, j)
   - The partial derivatives f_x, f_y, and f_t are computable from the pair of images
3. Stop if the error is less than a threshold, otherwise return to step 2.

It is important to have an estimate of confidence in the optical flow value. If the gradient magnitudes are very small (in space or time) then there is not much information with which to make an estimate (like figuring out how a smooth wall is moving by watching a a small patch).

Determining motion while the robot is moving

Two general methods based on calculation of the ***Focus of Expansion*** [FOE]

The FOE is the point in the image from which the scene appears to be coming

- Motion with a translational component always has an FOE that is a point not at infinity
- Rotational motion has an FOE at infinity in the direction of rotation.
- Backwards and forwards motion are similar, except backwards is a focus of contraction

Optical flow methods

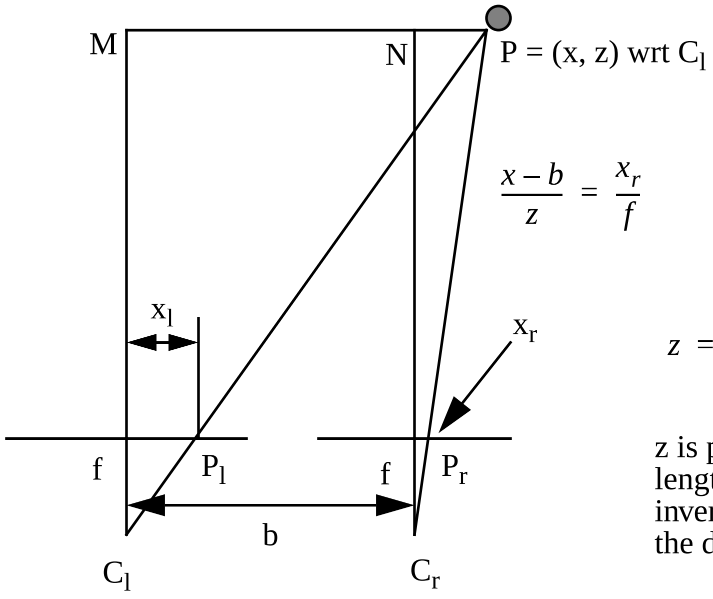
- Optical flow attempts to answer the question “where did it go?” for some number of pixels
- If you have enough answers, then you can do the following
  - Estimate camera motion from the robot odometry and/or the image sequence
    - One way to estimate the FOE is to have every pixel vote for a direction to the FOE
    - If more than half the pixels are moving due to the robot motion only, FOE wins
  - Subtract the component due to camera motion from the optical flow vectors
  - Remaining motion is due to an object moving independently from the robot
  - Group pixels that are moving together to find coherently moving blobs

Heuristic methods

- Determine the focus of expansion [FOE]
  - Example: Nair & Aggarwal, IEEE R&A, June 1998
    - Calculate vertical lines and horizontal lines (in two 3-D directions)
    - Given an a priori estimate of the vanishing point, estimate a new vanishing point
- Map the image into polar coordinates using the FOE as the origin
  - Motion along the radial axis is due to robot motion
  - Motion along the angular axis is due to non-robot motion
- Locate lines along the angular axis and at ±45° to it.
  - These are the edges of things that moved independently of the robot
- Map those lines back to the cartesian space and deal with them

Stereo Vision

$$\frac{x}{z} = \frac{x_l}{f}$$ similar triangles



$$\frac{x - b}{z} = \frac{x_r}{f}$$ similar triangles

$$z = \frac{bf}{(x_l - x_r)} = \frac{bf}{\Delta x}$$

z is proportional to focal length and baseline, and inversely proportional to the disparity.

Two-camera setup

- C_l, and C_r are the centers of projection for the left and right images
- The image planes are sitting between C_l, C_r, and the scene
- The scene projects onto the image planes according to the effective focal length f
- A point P (x, y, z) in the scene projects onto both image planes at locations x_l and x_r
- The disparity is the difference between x_l and x_r
- The baseline is the distance between C_l and C_r
- Using similar triangles we can extract a relationship between the disparity, baseline, and f

To do stereo

- Calculate f (focal length) for each camera
- Calculate b (baseline) for each camera pair
- For each pixel in a pair of images, calculate d = disparity

To calculate f and b for each camera (and other parameters) we have to do calibration

- A camera calibration tells you about the intrinsic and extrinsic parameters of a camera
    - Intrinsic parameters: things like focal length, which are fixed for a given camera
    - Extrinsic parameters: camera location and orientation (6 parameters)

For a given stereo setup you usually fix the cameras relative to one another and calibrate once

- This usually means bolting a portable stereo fixture to steel beams and gluing the lenses

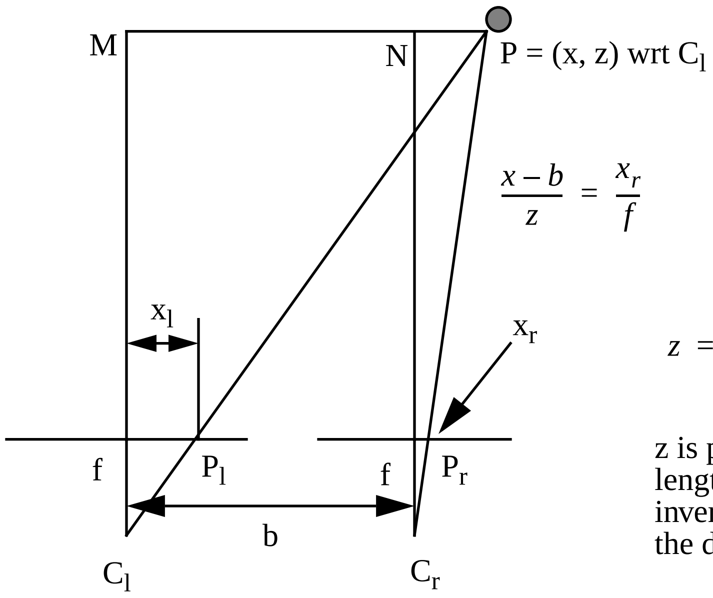
E28/CS82 Lecture #28 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Stereo

Stereo Vision

$$\frac{x}{z} = \frac{x_l}{f} \qquad \text{similar triangles}$$



$$\frac{x-b}{z} = \frac{x_r}{f} \qquad \text{similar triangles}$$

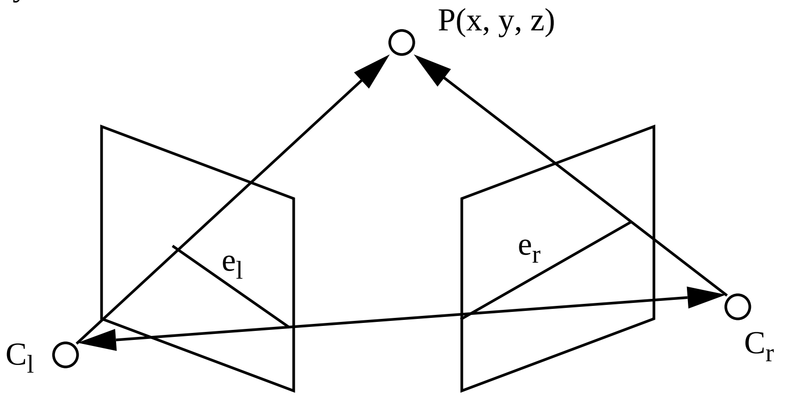
$$z = \frac{bf}{(x_l - x_r)} = \frac{bf}{\Delta x}$$

z is proportional to focal length and baseline, and inversely proportional to the disparity.

Where do we have to look for matching pairs

Assuming we have f and b, how do we find d?

Epipolar geometry:



- Each pixel in an image represents a ray into the world
- The projection of a ray onto the image plane is a line
- A point P in the image projects onto a single pixel in the left image
- The corresponding ray coming from the left image projects onto a line in the right image
- Therefore, the point P must also fall on that projected line
- Therefore, to find the corresponding point, you just have to look along the epipolar line

Epipolar constraint

- You only have to look along the epipolar lines to find corresponding points (1D problem)
- You can rectify images so that the epipolar lines lie along rows in the image
- Most solutions to stereo correspondence are based on rectified images

Area-based correspondence

A bottom-up method or correspondence that identifies matches based on local correspondence

- Generally assumes rectified images so that rows in the left and right images match up
- Grab a window in the left image around the pixel you want to match
- Compare it to a set of disparities (offsets) in the right image and pick the best match
  - A pixel at a far distance will have the same coordinates in both rectified images
  - A pixel at a closer distance will have a larger disparity
  - The closest object allowed in a scene sets the maximum possible disparity
- Flag pixels who best match is still really poor (occlusion, foreshortening, specularities)

Characteristics of area-based matching

- Easy to implement, gives a dense depth map
- Accuracy is potentially not as great as with sub-pixel feature-based methods
- Requires texture in the image
- Can be implemented in real time using fast hardware

Example: Sum of Sum of Squared Differences [SSSD]

The basic tenet of SSSD is that having more stereo pairs is better

- Use multiple cameras in some kind of L or T arrangement (vertical and horizontal pairs)
  - Calibrate the cameras relative to each other, and consider all possible stereo pairs
- Matching for an image pair is uses SSD of a local area around the epipolar line
  - The point with the minimum SSD is the best match and gives the disparity
- You want to average the error results together to get a more robust result
  - You can't use disparity space since each stereo pair will have different disparities
  - You can use Z-space to combine things (error v. Z-value graph)
    - Think about one of the cameras as a basis camera
    - Consider, to start, all stereo pairs that include this camera
    - For each pixel and for each stereo pair calculate the error for each depth value
    - Sum the SSD values together to get a global error v. Z curve
    - Fit a 2nd order parabola to the curve and pick the minimum as the true depth

One way to try and deal with photometric constraint violations (pixels that correspond to identical points in the world, but that are not the same color in two images) is to work in an intensity independent space like the gradient or Laplacian image

- Run a 1st or 2nd order gradient (Laplacian) over the image prior to SSD matching
  - 2nd order is rotationally symmetric, so it is usually preferred for this application
  - Generates lots of texture in otherwise apparently feature poor images

In addition, we can be intelligent about what we try to match.)

- Calculate the variance of a window to determine whether it's worth trying to match
- You can end up losing walls or other flat surfaces, but it's probably better than noise

We can also get confidence in our result by doing reverse matching (right to left) to see if it gives us a similar result

Calibration and Geometry

When you are working with cameras, you have to know the coordinate systems in which various things reside. This also helps us to think about how to move between coordinate systems--like between the image and the world.

Basics of Projective Geometry

The basis of projective geometry is an equivalence relation between (n+1) dimensional vectors

$$[x_1, \ldots, x_{n+1}] \equiv [x_1', \ldots, x_{n+1}']$$

if and only if the left vector is a scaled version of the right. A projective space consists of vectors like these where there is a canonical representation for a set of equivalent vectors.

Projective points are usually expressed as with a 1 in the last position: e.g. [x, y, z, 1]

- Each such point represents a set of similar points
- The point [x, y, z, 0] represents a point at infinity in the direction [x, y, z]
- Projective spaces are useful because we can easily apply projective transforms to them

Coordinate systems for a single camera sitting on a robot

To calibrate a single camera we first need to define the coordinate systems of interest

- World coordinates: Origin O_w at an arbitrarily selected point, a point = [X Y Z 1]
- Robot world coordinates: Origin Rw at an arbitrarily selected point on the robot
  - World and Robot coordinates are related by a translation and a rotation
  - Rotation can be 1D (if vertical axes align) or 3D if they don’t
- Camera Euclidean coordinates: has the focal point of the pinhole camera O_c as its origin
  - The z-axis Z_c, is aligned with the optical axis and points away from the image plane (therefore, all of the points you see in an image have positive z values)
  - A translation and rotation is all that is necessary to transform the Robot coordinates to Camera Euclidean coordinates
  - A second transformation moves the camera to world coordinates
- Image Euclidean co-ordinate system: axes are aligned with the camera coordinates
  - X_i, Y_i lie on the image plane and are aligned with the camera coordinates
  - Origin is on the image plane
  - Camera and image Euclidean coordinates are related through a projective transformation
- Image affine co-ordinate system: coordinates u, v, w
  - v, w are aligned with the Y and Z axes of the image Euclidean system, respectively
  - u may have a different orientation and scaling
  - Origin is coincident with the image Euclidean system
  - The principal point (the place where (0, 0, Z) in camera coordinates projects to) is located at $U_0 = [u_0, v_0, 1]$

E28/CS82 Lecture #29 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Camera Calibration

Representing the relationships between coordinate systems

To move from world coordinates to robot coordinates a point has to undergo a rotation R and a translation t.

- $$X_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = R(X_w - t)$$
- R_r and t_r provide all of the parameters necessary to locate the robot in the world
- A second R_c and t_c provide all of the information necessary to locate the camera relative to the robot.
- If we combine these two transformations we can define the camera in world coordinates
- These are called the extrinsic parameters (X, Y, Z, $Ø_x$, $Ø_y$, $Ø_z$)

To move from camera space to the affine image space we need to go through a perspective projection, a scale, a shear, and a translation.

- $$\tilde{u} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} a & b & -u_0 \\ 0 & c & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{-f x_c}{z_c} \\ \frac{-f y_c}{z_c} \\ 1 \end{bmatrix} = \begin{bmatrix} -fa & -fb & -u_0 \\ 0 & -fc & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x_c}{z_c} \\ \frac{y_c}{z_c} \\ 1 \end{bmatrix}$$
- The matrix on the right is sometimes called the camera calibration matrix K.
- K contains all of the intrinsic, or optical parameters of the camera
  - -fa = u scaling (f in pixels along the u axis)
  - -fc = v scaling (f in pixels along the v axis)
  - -fb = u shear
  - [u_0, v_0] = principal point

We can express the complete relationship between world Euclidean coordinates and image affine coordinates as a single 3 x 4 matrix M, which is composed of the submatrices

- M = [KR | - KRt]

Single-camera calibration

Known-scene

- If you have a scene in which you know the World coordinates of enough locations, then you can solve for M directly as an over-determined system of linear equations
- Usually use a special calibration object, usually a cube or two surfaces at right angles
- Multiple images of a planar target works as well

Unknown scene

- You need to have multiple views of the same scene
- You need to make correspondences between the multiple views
- The intrinsic parameters do not change between the views
- The world parameters of the corresponding points don't change between views
- If you know the camera motion you can solve the problem
- If you don't know the camera motion, you need three views and it's still hard numerically

If you want a good calibration for a camera in a given location with given settings, then you need some calibrated points (3D world locations and 2D image locations)

- Common targets are two planes or three planes at 90° angles from one another
  - Usually circles are printed on the planes, often with unique identifying characteristics
  - You can segment the circles automatically and find their centroids
  - You can also use a single plane at two or more distances from the camera
  - Checkerboard patterns work pretty well
- The points on the targets must be at regular intervals in some coordinate system (e.g. m)
- Once you have an image of the target you can correspond the known locations of the target points with image locations and get a point cloud of 3D points and their 2D projections

With the point cloud you can run a camera calibration algorithm, which is a non-linear optimization problem

- Tsai & Lenz developed a robust algorithm for finding calibrations; Reg Wilson's code is available freely from the CMU CV home page
- Matlab Calibration Toolbox is a good, general purpose, freely avaible toolbox
  - Same code has a C implementation in the OpenCV code base

The output of the camera calibration is the calibration matrix M = [KR | KRt]

- The algorithm also outputs the errors associated with the calibration

Two-camera calibration: the fundamental matrix

In stereopsis, you have two images of the same thing from different viewpoints. The epipolar geometry gives us a set of relationships that tell us how the images are related.

- If you consider the vectors projecting from a point in the scene (X) to the camera centers and the vector connecting the camera centers (**t**), they are coplanar
  - **t** is the translation between the camera centers
  - R is the rotation from the left camera axes to the right camera axes
  - X is the vector from the left camera to the point, K is the left calibration matrix
  - X' is the vector from the right camera to the point, K' is the right calibration matrix
  - We can express X' in terms of the left camera axes as $X'_L = R^{-1}X'_R$
- We can express co-planarity as a dot-product and cross-product: $X_L^t(t \times X'_L) = 0$
- We can also express the point X in the scene as the image points (u and u') projected into the scene from each camera through each camera's calibration matrix (K and K')

If you make the substitutions for X and X' in the above equation, then you get:

- $(K^{-1}u)^T(t \times R^{-1}(K')^{-1}u') = 0$

- Let the matrix S(t) represent a vector-product (cross-product)

$$S(t) = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$$

- $u^T(K^{-1})^T S(t) R^{-1} (K')^{-1} u' = 0$
- Let $F = (K^{-1})^T S(t) R^{-1} (K')^{-1}$ be the fundamental matrix
- $u^T F u' = 0$
    - This is called the fundamental matrix
    - It contains all the information we can extract from point matches in a pair of images

Note that the center of the fundamental matrix is a translation and a rotation

- It is the coordinate change between the left and right images
- This is called the essential matrix $E = S(t)R^{-1}$

If you have calibration matrices for the left and right images, then you can extract the essential matrix, which is the motion of the camera between the two images.

- $\hat{u} = K^{-1}u$ and $\hat{u}' = (K')^{-1}u'$, so $\hat{u}^T S(t) R^{-1} \hat{u}' = 0$

E28/CS82 Lecture #30 S05

Overall Topic: Robot Vision

A Practical Guide to Robot Vision: Fundamental and Essential Matrices

Fundamental Matrix Review

The fundamental matrix encapsulates everything we know about two views

- Encodes epipolar geometry
- Position and view dependent (always exists for 2 views, but there are degenerate cases)

One way to derive it is to use an explicitly geometric argument based on the epipolar plane

- Consider two lines from the optical center o' to the point x' and to the epipole e'
- The epipole e' is the image location of the optical center of the other image
  - e' may not actually be in the image proper
- The line l' is the cross product of x' and e', and is orthogonal to the epipolar plane
- We know there is a homography H such that x' = Hx

$$x' = Hx \qquad l' = e' \times x' \qquad l' = e' \times Hx \tag{1}$$

- We can represent the cross product with e' as a skew-symmetric matrix [e']
- The dot product of x' and l' is zero (orthogonal vectors

$$x^{t}{}'l' = 0 \qquad x^{t}{}'[e']Hx = 0 \qquad x^{t}{}'Fx = 0 \tag{2}$$

Calculating F given known point correspondences

Linear Method

- The fundamental matrix is only known up to a scale (8 unknown parameters)
- Need at least 8 points (preferably lots more)
- Generate a matrix with at least 8 point correspondences (1 per row)
- $Af = \begin{bmatrix} u_iu'_i & u_iv'_i & u_i & v_iu'_i & v_iv'_i & v_i & u_i & v'_i \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ \dots \\ f_{32} \end{bmatrix} = 0$
- Prior to generating the matrix, you have to normalize the image coordinates (**important**)
  - Translate and scale the points so their average is 0 and distance from the origin is $\sqrt{2}$
  - Represent the scale and transations as T (for x) and T' (for x')
  - Final $F = (T')^{-1}FT$
- We want a value for f that also follows the constraint that $\|F\| = 1$
  - F can only be calculated up to a scale factor, in any case
- Execute SVD on A, to get $A = UDV^T$
- The columns of V are the eigenvectors of A^tA,
- D will only have eight non-zero (not almost zero) singular values
  - The normal vector corresponding to the 0 singular value is F

The value (3x3 matrix) calculated for F may not have a rank of 2 (it could have a rank of 3)

- Decompose F one more time using SVD $F = UDV^T$
- If $D = \text{diag}(r, s, t)$ where $r > s > t$, then recompute F as $F = U\text{diag(r,s,0)}V^T$

Polishing off

- When you have an initial estimate of F using the linear method, you need to improve it
- Set up an expression to minimize
  - For example , SSE of points and the projections of their counterparts using F
- Use a non-linear minimization routine like Levenberg-Marqhart to improve the estimate

Calculating F in the real world (unkown point correspondences)

1. Use something like the Harris corner detector or SIFT features to calculate interest points
   - SIFT features are more complex, but more robust
   - Harris corners are the current favorite and are fast to calculate

2. Match interest points between images
   - Grab a piece of the image around each interest point
   - Execute an SSD with interest points in the other image to find the best match
   - Keep matches that are bi-directional (going both ways it is the best match)

3. Execute RANSAC to calculate an initial F
   - Randomly pick 8 points
   - Execute the linear algorithm to estimate F
   - Calculate how many other points support the estimate of F (calculate x'Fx as the error)
   - If enough support exists for a particular F, terminate
   - Else, repeat with a new set of points
   - Terminate after N iterations

$$N = \frac{\log(1-p)}{\log(1-(1-\varepsilon)^s)} \tag{3}$$

   - p = probability of success if finding 8 good inliers
   - s = number of points that need to be picked to estimate the model
   - ε = percent of outliers in the data
   - Need 1177 points to get 99% success at picking 8 points with 50% outliers

4. Calculate F using support points (inliers)
   - Calculate how many inliers there are for the F with the most support
   - Recalculate F using all of the inliers and the linear method
   - Calculate if there are any new inliers with the updated estimate, if so repeat
   - Can also use F to improve upon the initial point matches between images

5. Polish F
   - Use a non-linear maximization algorithm to improve the estimate of F

Given F, how do you get E?

$$F = (K^{-1})^T S(t) R^{-1} (K')^{-1} \qquad E = K^T F K' \tag{4}$$

- If you have F, use camera calibration matrices to get E
- use SVD on E to break it down and calculate R and S(t)
    - $E = UDV^T$
    - By its form, E has two non-zero singular values, and they are both equal
- As with F, we may want to recompose E using the two largest singular values and averaging them.

$$E = U \begin{bmatrix} \frac{r+s}{2} & 0 & 0 \\ 0 & \frac{r+s}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5}$$

- To decompose E into a rotation and a skew-symmetric translation, use

$$R = UGV^T = U \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \qquad S(t) = VZV^T = V \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \tag{6}$$

What can you do with E

Once you have E for two views, then you know the rotation and translation of the camera center between them.

- This is the basis for visual odometry

Algorithm

- Calculate interest points in each new image
- Correspond the interest points with the previous image (images)
- Calculate F (or the trifocal tensor, an equivalent entity for 3 images)
- Use the camera calibration matrices to calculate E
- Decompose E into R and S(t)

You can use visual odometry in conjunction with regular odometry, particle filters, Kalman filtering, and SLAM.

- Along the way you get 3D information about the interest points
- Can put those locations into a map of obstacles in the environment

E28/CS82 Lecture #31 S05

Overall Topic: Visual Tracking, Face detection, HMMs

A Practical Guide to Robot Vision: Tracking

To track something in a visual field you need to have

- A method of finding candidate image locations to track
- Associating measurements with targets
- A method of maintaining the state of the tracked objects
- Creating or terminating target tracks
- Moving the robot/camera to follow a target

Finding things to track

Color & thresholding

- Robot soccer (artificial environments with colored landmarks, illumination dependent)
- Template-based tracking (known environments or objects, or short-term tracking)
- Histograms (known objects or short-term tracking, illumination dependent)
- Feature-based tracking
  - known-objects with good training data
  - tend to be illumination independent

Data Association

The data association problem is the task of attaching a cause to each sensor reading

- A reading can be a false target (noise)
- A reading can be an actual known target
- A reading can be an actual unkown target

Maximum likelihood association with gating

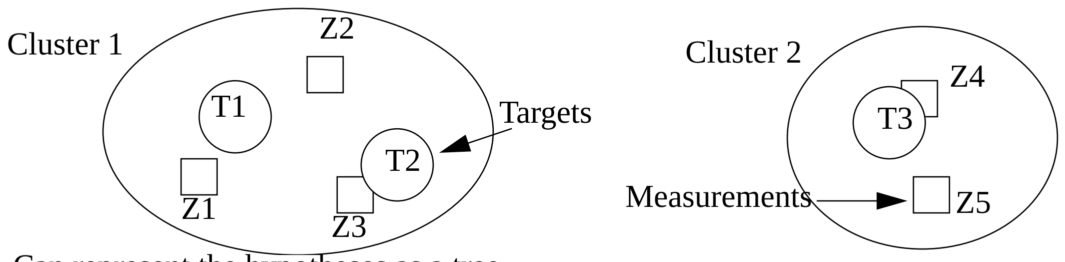
- Associate the most likely measurement with each target
- Don’t consider measurements that are outside of the gate for a target
  - Gate is generally a threshold based on the target’s inherent error distribution
  - For a Kalman filter, the inherent error is P(k)
- May have a linear programming task when targets are close to one another

Joint probability distributions with gating

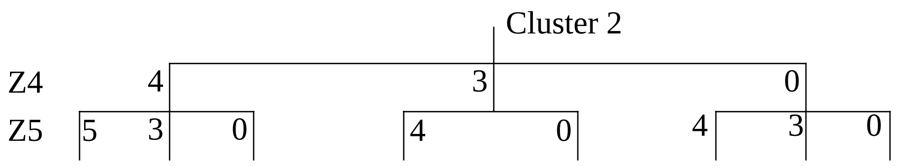
- Gate measurements around targets
- Update each target according to the joint probability distribution
  - What is the probability of the target creating each of the measurements within the gate
  - Update the target by weighting the measurements by their probability

Clustering and N-scan methods (Reid) with gating

- Cluster the measurements into non-overlapping groups
  - Targets that are well separated will each have their own group
- Within each cluster there may be multiple targets and measurements
- Given N measurements and T targets within a cluster, generate all the possible hypotheses
  - A target can only be associated with a single measurement



- Can represent the hypotheses as a tree
  - Each measurement increases the depth of the tree
  - The tree grows over time
  - Each path from leaf to root represents a complete data association
  - Start with the tree at a new time step (so the tree also represents time)



- How to trim the tree?
  - Get rid of hypotheses with very low probabilities (pruning)
  - Combine hypotheses with similar characteristics
    - N-scan method: look back N steps in time and combine similar associations
    - N-scan with N= 1 turns out to work pretty well
    - Similar outcomes method: combine hypotheses with similar outcomes
  - Remove targets from the tree that have a unanimous data association
- How to initiate a new track?
  - When all of the valid hypotheses in a tree unanimously vote for an unknown target
- How to delete a track?
  - When there is no more support for it (no measurements within its gate)

Maintaining track states

Many of the above methods will work for either of our two main state filters

Kalman filtering
- Have to use multiple Kalman filters
- Data association is important (just like in localization)

Particle filtering
- Have to use multiple particle filters
- Crossing tracks is hard

Handling multiple tracks explicitly is hard
- Joint probability spaces are very large
- Creating and ending tracks is heuristic
  - End up doing clustering and looking at what is unaccounted for

Moving the robot or camera to follow

Have to determine the relationship between the possible motions and the object

- Can move the camera (2 DOF)
- Can move the robot (2 DOF)

What is the relationship between camera motions and object motions?

What is the relationship between robot motions and object motions?

Answer: the Jacobian tells you the relationship between image velocities and parameter velocities

- Usually wiggle the camera a bit, or wiggle the robot a bit to calculate the Jacobian
  - Empirically defined, and may be different for different parts of the images

If you have a calibrated camera and know where the object is, you can calculate it analytically

E28/CS82 Lecture #32 S05

Overall Topic: Face detection, HMMs

A Practical Guide to Robot Vision: Face Detection

One of the hottest detection algorithms right now is based on a concept first developed by Viola and Jones.

- Integral image
    - Each pixel stores the sum of the values to the left and up
    - Can be computed in a single pass
- Rectangular features
    - Two-rectangle features (horizontal or vertically adjacent)
    - Three-rectangle features (one in the middle)
    - Four-rectangle features (arranged in a checkerboard pattern)
    - All can be calculated in constant time using the integral image
- Adaboost for learning
    - Test all features
    - Pick the one with the best performance (but no false negatives) and add it to the mix
    - Weight the samples so that samples that are consistently mis-classified get more weight (which means a classifier that gets it right is more likely to get picked)
    - Stop when your accuracy is sufficiently good
- Detect as a cascade of weak detectors

State Machine-based Pattern Recognition

Think of a pattern as a 1-D sequence in an image

- Represent the 1-D sequence as a string of symbols
- The pattern you want to find is specified as a set of possible words
    - e.g: R*G*B* = one or more reds, one or more greens, then one or more blues.
- We can develop a finite state automata that tests words for validity

For example: we are looking for a vertical flag that is red, followed by white, followed by green. We want to detect the flag if it is at least 30 pixels long in the image.

Given: three classifiers for the colors {red, white, green}

Algorithm:

```
Scan the image for a Red pixel
Start the FSA in the SeenRed state, with a Counter = 0, and Noise = 0
Start moving down the column
While in the SeenRed state
      If the next pixel is Red
            Stay in the SeenRed state and increment Counter
      Else if the next pixel is White
            If Counter > 10
                  Set state equal to SeenWhite, Counter = 0, Noise = 0
            Else
                  break and keep scanning
```

```
        Else (pixel is not {Red, White})
            increment Noise
            If Noise > threshold break and keep scanning
    While in the SeenWhite state
        If the next pixel is White
            Stay in the SeenWhite state and increment Counter
        Else if the next pixel is Green
            If Counter > 10
                Set state equal to SeenGreen, Counter = 0, Noise = 0
            Else
                break and keep scanning
        Else (pixel is not {White, Gree})
            Increment Noise
            If Noise > threshold break and keep scanning
    While in the SeenGreen state
        If the next pixel is Green
            Stay in the seenGreen state and increment Counter
        Else
            Increment Noise
            If Noise > threshold break and keep scanning
        If Counter > 10 return positive detection
```

FSA's are fine for recognizing valid strings in a language

- What if we want to train them?
- What if we also want to train the classifiers?
- What if we don't actually know all of the words in the vocabulary?
- What if we want to get something in between "yes" and "no" for an answer?

Dynamic Time Warping

What if we had an example pattern P and wanted to find the best match with a new pattern?

- Let P have some "stretchiness" to it

Given two patterns that extend arbitrary distances along a dimension T, we want to know the optimal match between the two patterns given two constraints.

- The patterns begin and end together
- A specified amount of contraction and expansion is permitted for each pattern

This method has been used in speech recognition, but is also useful for matching 1D histograms

- Could also be used to match 2D patterns by using one spatial dimension for T

The dynamic time warping is the minimum cost path, defined by $(\phi_x(k), \phi_y(k))$ that starts at element (0, 0) and goes to elements (T_x, T_y). The minimum is taken over all possible paths.

A recursive definition of the dynamic time warp distance is:

$$D(i_x, i_y) = min[D(i_x', i_y') + \zeta((i_x', i_y'), (i_x, i_y))] \qquad (1)$$

which says that the distance to (i_x, i_y) is the minimum over all possible previous locations (i_x', i_y') of the dynamic time warp to the previous location plus the cost of getting to the current location from the previous location. The cost function has two parts: 1) distance between the corresponding patterns from the two input vectors, 2) a weighting that is a function of the distance traveled in the DTW graph (i.e. change in coordinates).

In dynamic time warping the key is defining what kinds of steps are permitted in moving from one location to the next.

As an example, consider permitting three previous locations from (i_x, i_y), namely $[(i_x - 1, i_y), (i_x - 1, i_y - 1), (i_x, i_y - 1)]$.

This results in three cases to test for the optimal previous location. The weighting functions for these three previous locations might be [1, 1.414, 1]. Note that this weighting prefers diagonal motion rather than a stairstep progression through the DTW graph.

The algorithm is now as follows

1. Initialization

$$D(1, 1) = d(1, 1)m(1) \tag{2}$$

2. Recursion

For $1 \le i_x \le T_x, 1 \le i_y \le T_y$ such that the path stays within the allowable grid, compute

$$D(i_x, i_y) = min[D(i'_x, i'_y) + \zeta((i'_x, i'_y), (i_x, i_y))] \tag{3}$$

where $\zeta((i'_x, i'_y), (i_x, i_y))$ is defined as the path cost between its arguments.

3. Termination

$$d(X, Y) = \frac{D(T_x, T_y)}{M_\phi} \tag{4}$$

where M_ϕ is a normalizing constant usually defined simply as T_x.

Dynamic time warping works with a single example, or with K examples of each class

E28/CS82 Lecture #33 S05

Overall Topic: Face detection, HMMs

Hidden Markov Models for Vision (and anything else)

Hidden Markov Models [HMMs] are probabilistic state machines

- There are a set of N states (S) with a specified connection topology
- There are a set of possible output values (V)
- Variation in timing can be represented by state transition probabilities (A)
- Variation in value can be represented by output probabilities (B)
- Variation in starting values can be represented by initial state probabilities (π)

A model is represented by three sets of probabilities: $\lambda = (A, B, \pi)$

The output values may be discrete or continuous

- If they are discrete, then the output probabilities B are the probabilities associated with each discrete output
- If they are continuous, then the output probabilities B are the parameters of one or more statistical distributions (like means and standard deviations)

Synthesis

Given a model, we can generate random numbers and create a likely observation sequence

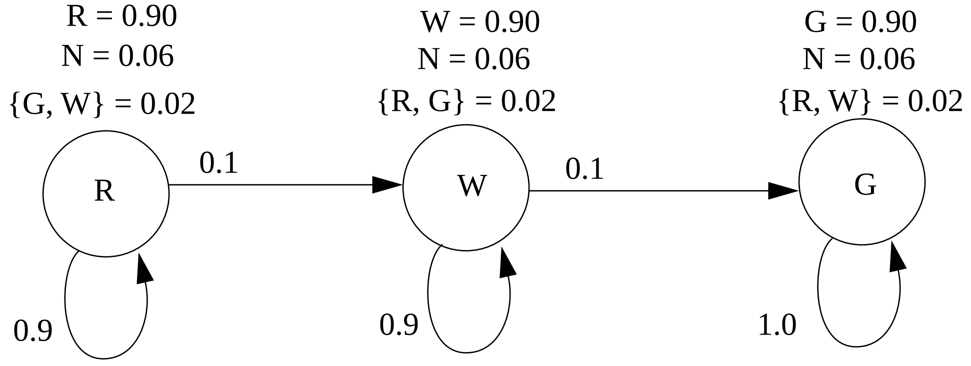
- Use the initial probabilities to generate a starting state, then repeat the following
- Use the output probabilities for the current state to generate an output value
- Use the transition probabilities for the current state to move to the next state

Analysis

Given a model, we can generate two things given an observation sequence

- The most likely state sequence given the set of observations
- The total probability of the model generating the observation string
    - This is the sum of the probabilities of any possible state sequence through the model generating the observation

Example: Red/White/Green flag sequence



Given a state sequence, it is straightforward to calculate the probability of that state sequence generating the observation

- Use the initial probabilities to get an initial probability
- Use the output and transition probabilities (all multiplied together) to get the rest
- e.g. observation sequence RRWNGG given the state sequence RRWWGG is
    - (1.0*0.94)(0.9*0.94)(0.1*0.94)(0.9*0.06)(0.1*0.94)(0.9*0.94) = 3.2 x 10-4

The Viterbi algorithm gives us an efficient method of calculating the most likely state sequence

Forward procedure

Define the forward variable $\alpha_t(i)$ that is the probability of having seen a particular observation sequence and being in state i at time t. We can inductively solve for this variable on the order of N^2T calculations.

Definition: $\alpha_t(i) = P(o_1 o_2 \ldots o_t, q_t = i | \lambda)$

Initialization: $\alpha_1(i) = \pi_i b_i(o_1) \qquad 1 \le i \le N$

Induction: $\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_i(i) a_{ij}\right] b_j(o_{t+1}) \qquad 1 \le t \le T-1 \qquad 1 \le j \le N$

The basic idea is that the likelihood of being in state i at time t is the sum of the likelihoods of being in all possible previous states at time t-1 and then moving to state i and seeing o_t at time t. This leads to a recursive definition that we can calculate as a lattice in a forward manner.

- Procedure: calculate all $\alpha_0(i)$, then all of them for time step 2, and so on.
- Note that the sum of the $\alpha_T(i)$ will be equal to the total probability of the observation given the model parameters $P(O|\lambda)$ (sum of probability of all state sequences)

Backward procedure

We can also define the backward variable $\beta_t(i)$, which is the probability of seeing an observation sequence from time t+1 to the end, given that the model is in state i at time t. Again, there is an inductive proof that forms the basis for a recursive definition and calculation procedure.

Intialization: $\beta_T(i) = 1 \qquad 1 \le i \le N$

Induction: $\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \qquad t = T-1, T-2, \ldots, 1 \quad 1 \le i \le N$

The main idea is that in order to have been in state i at time t, and to account for the observation sequence from time t+1 on, you have to consider all possible states j at time t+1, accounting for the transition from i to j and the observation o_{t+1} which is seen in state j.

Current state probability

A third useful variable is $\gamma_t(i)$, which is the probability of being in state i and time t given an observation sequence O. We can define this in terms of probabilities

Definition: $$\gamma_t(i) = \frac{P(O, q_t = i | \lambda)}{P(O|\lambda)} = \frac{P(O, q_t = i | \lambda)}{\sum_{j=1}^{N} P(O, q_t = j | \lambda)}$$

It turns out that gamma is simply the product of α and β, as α is the probability of seeing the first part of the sequence and being in state i, while β is the probability of seeing the latter part of the sequence and being in state i.

Definition II: $$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}$$

So γ (calculable in $O(N^2 T)$ time) tells us some very useful information. This information can be used for training the HMM parameters because it lets us weight the observation and transition values appropriately given a set of training data.

Viterbi Algorithm

The Viterbi algorithm tries to find the best state sequence given a model and a set of observations. It is similar to the calculation of alpha, but is a little faster because we don't have to consider all paths through the model.

The basic idea is that all substrings of the optimal path through the model must also be optimal paths themselves. Hence, we start at the beginning and track the optimal path to each state at each time step, then trace back from the best state at the end to get the answer.

Initialization: $\delta_1(i) = \pi_i b_i(o_1) \qquad 1 \le i \le N$

$\psi_1(i) = 0$

Recursion $\delta_t(j) = \max_{1 \le i \le N}[\delta_{t-1}(i) a_{ij}] b_j(o_t)$

$\psi_t(j) = \arg\max_{1 \le i \le N}[\delta_{t-1}(i) a_{ij}]$

$2 \le t \le T \qquad 1 \le j \le N$

Termination $P = \max_{1 \le i \le N}[\delta_T(i)]$

$q_T = \arg\max_{1 \le i \le N}[\delta_T(i)]$

Path Backtracking $q_t = \psi_{t+1}(q_{t+1}) \qquad t = T-1, T-2, \ldots, 1$

Training HMMs

The Baum-Welch training algorithm is a gradient descent method of training HMMs. It works by taking a set of positive training examples and calculating improved parameters for the HMMs through a gradient descent algorithm. It is a relatively fast procedure, and usually only needs to be run a few times to get decent results.

You can also try to train HMMs with something like genetic algorithms or another stochastic search method, but these have not been conclusively shown to produce better results.

There are tens, if not hundreds of modifications of HMMs that people have tried to improve them. The most important being the introduction of continuous HMMs with covariance matrices to represent multiple element feature vectors of continuous variables.

The key to training HMMs is a large training set. Many states have small probabilities of seeing various inputs, and there has to be a lot of training data in order to tune these parameters.

Application Example: Face recognition using video sequences and HMMs

Represent each person with an N state fully-connected continuous HMM

Each subject has a training video sequence containing T images of the subject's face

- Each image in the video sequence has only the face portion
- The faces are projected into a face space developed from all of the training data
- The covariances of the face space coefficients can be calculated from the training data

To initialize the HMMs

- Vector quantize the feature vectors into N classes (given N states in each HMM)
- Initialize the covariance and mean values for each HMM using the class means

To train the HMM

- Use the Baum-Welch EM algorithm (gradient descent)
- Continue B-W training until the HMMs converge

To calculate who an unknown person is given the video sequence:

- Project each frame into the face space to get a feature vector sequence
- Calculate the probability that each HMM generated the sequence
- Recognize the individual as the best match, unless no match is good

In comparison with IPCA based face recognition, HMMs do better

- IPCA is projecting into an individual's face space and calculating the residual error

E28/CS82 Lecture #34 S05

Overall Topic: Purposive Robots

Robot: “What do I do now?”

The ultimate goal of robotics has two parts:

1. To get the robot to do about the right thing most of the time, and
2. To avoid doing the wrong thing all of the time.

The phrasing of these two goals is intentional, and asymmetrical. A robot doing the wrong thing just once can result in bad consequences for the robot, both short and long-term. On the other hand, doing exactly the right thing is not really necessary in most situations, especially when dealing with people; doing about the right thing is good enough.

- People personify robots, which means they can make small mistakes and it’s ok
- People adapt to robots so long as their expectations are not strongly contradicted
- If the robot does something close enough to expectations, people are happy.

Assume you have a robot that has at least the following capabilities

- Safe navigation
- The capability to wander (free-space following) or explore (wall-following)
- The ability to find or detect people (a camera)
- Some ability to localize itself, recognize landmarks, or build maps

The missing piece is intentionality

- What should the robot do?
  - Vacuum
  - Take photographs
  - Serve snacks
  - Map an area
  - Look for victims (USAR task)
  - Pick up trash
  - Deliver the mail
  - Assist someone with information or guidance
  - ...

For each of this tasks, the robot needs to make decisions at periodic intervals about what to do

- These tasks all require sequential decision-making
- They all require some kind of state estimation
  - Implicit: world as state and sensors as returning state information
  - Explicit: internal representation of the world, updated by sensors

Behavior-based stragies (implicit state read from the world) are not robust enough and are too difficult to design for long-term sequential actions, so to effectively solve real-world problems we some kind of internal state representation

The paradigms within the explicit state representation are ones we have met before:

1. Exact knowledge of current state, exact knowledge of the effect of actions
    - The state estimate is assumed to have no uncertainty
    - The effects of actions are assumed to be known, or can be measured to a sufficient degree
    - Like relying on odometry for navigation
2. Incomplete knowledge of state, uncertain effects of actions
    - The state itself is uncertain, or may take on multiple values simultaneously
    - The effects of actions are uncertain, but we can take measurements to reduce uncertainty
    - Similar to the SLAM problem

Robot control not only requires knowledge of current state, however, we also need to know what to do next given that we are in a current state (probabilistic or deterministic)

- We need to have a model for action
- Given that the robot is in State S, and is observing O, what action should it take?

There are a number of ways to approach that problem

1. Programmer knowledge: encode the sequential information
    - Do X, then Y, then Z
2. Rule-base knowledge: encode the knowledge about how to complete a task in searchable form
    - Encode rules about how the world works
    - Give the robot a search mechanism for the knowledge base
    - If the robot wants to achieve a goal, it can search for the sequence of actions that achieve the goal (Monkey, chair, stick, banana type problem).
3. Learned knowledge: let the robot learn sequential action knowledge by providing rewards
    - Let the robot explore
    - When it does something right, give it a treat
    - When it does something wrong, give it a slap on the wrist
    - Learn the sequences of actions that provide rewards and avoid slaps

Options 1 and 2 work best when you have good state estimates (low uncertainty)

- Not so hot when state is uncertain

Option 3 can integrate state uncertainty into the learned sequential knowledge

- Still need to design the reward structure and execute the learning
- Reward structure has similar effects to fitness functions in GAs
- Learning has to balance exploration with pursuing the optimal reward

Making Decisions: Finite-State Autonomata

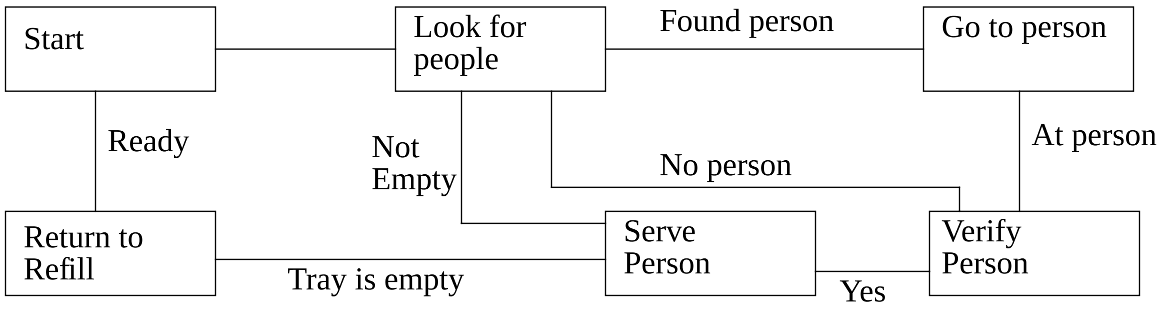
Encode sequential behavior as a state machine

- Tend to be programmer designed
- Use sensor information to move from one state to another
- Generally provide high-level guidance rather than control loop level management

Model

- Known set of states
- Known set of deterministic transitions
- Intentionality (reward structure) built into the FSA

Example: Serving Snacks



Making Decisions: Markov Decision Processes

When your state is known, the results of actions are stochastic, but the reward structure is known, 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 $T:(S \times A \rightarrow \Pi(S))$
    - $\Pi(S)$ is a probability distribution over the states S (state transitions are probabilistic)

The goal of the robot is to take the action at each time step that maximizes rewards

- Over what time period?
- How important are future rewards?

There are three models for making decisions

- Fixed-horizon decision-making
    - Horizon is k steps away
    - Rewards beyond step k are irrelevant
    - Horizon can be fixed (agent resets or dies after k steps)
    - Horizon can be moving (agent always looks ahead k steps)
- Infinite horizon decision-making
    - Consider rewards far into the future
    - Not particularly realistic
    - Very difficult to model
- Discounted infinite-horizon decision-making
    - Rewards in the future are not as important as rewards now
    - Discount factor $\gamma < 1$ is an exponential factor
    - Discount factor for the reward k steps out is γ^k
    - There are finite-time solutions for solving the finite horizon model

E28/CS82 Lecture #35 S05

Overall Topic: Purposive Robots

Making Decisions: Markov Decision Processes

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 precalculate what the best action is for each state

$$V^*(s) = \max_{\pi} E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) \tag{1}$$

$$V^*(s) = \max_a \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')\right) \tag{2}$$

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

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

```
initialize V(s) arbitrarily
loop until policy is good enough
      loop for all states s
            loop for all actions a
                  $Q(s, a) = R(s, a) + \gamma \sum_{s' \in S} (T(s, a, s')V(s'))$
            V(s) = max Q(s, a) over all actions a
      end loop
end loop
```

The basic concept is to relax the values in the graph until they start to converge

- Good and bad rewards will propagate backwards through chains of actions

It turns out you can approximate the central equation by take the max instead of the sum term

- Requires lots of running time
- Need to sample from T so you don't always take the same transition
- Basis for model-free methods (unknown reward/transition structure)

Making Decisions: Reinforcement Learning/Q-Learning

What if you don't know the reward structure beforehand?

- You can move from state to state, but you don't know what states/actions provide rewards
- The state transition probablities may also be unknown

The model-free situation is the case for a robot trying to learn a task

- The robot explores, gets rewards (positive or negative) and tries to figure out what it did right when

Let $Q^*(s, a)$ be the expected discounted reinforcement of taking action a in state s, then continuing on by choosing optimal actions in each subsequent state.

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \max_{a'} Q^*(s', a') \tag{3}$$

The update rule for learning Q* is

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{4}$$

The update rule can be executed for each expeience tuple $\langle s, a, r, s' \rangle$, which is a state, an action, a reward, and a resulting next state. The constant α is a learning constant that decays over time.

Q specifies the optimal action in state s by taking the max Q over all actions available
- The maximum Q(s, a) in a state is also the V*(s) policy in the MDP model

When Q has converged to Q*, the robot can act greedily at each time step
- But if Q has not converged, the robot needs to explore
- Exploration is not something that is theoretically well-defined
- Heuristic exploration methods are currently used
  - e.g. Simulated annealing: pick non-optimal actions with some probability

Issues with Q-learning
- Generalizing over large state and/or action spaces (i.e. continuous ones)
- Slow convergence
- Sparse reward structures

Q-learning works best with smaller state/action spaces and dense reward functions
- Not a GA approach
- More related to gradient descent methods

Variations on Q-learning: Dyna architecture

The basic Q-learning algorithm only updates the statistics for the current $\langle s, a, r, s' \rangle$ tuple
- What if we also updated some other tuples in the Q table at each step?
- What if we also want to explicitly keep track of T and R?

Process
- Given an experience tuple, update the model for T(s, a, s’) and R(s, a)
- Update the policy (Q) at state s using the Q learning rule and the latest T and R values
- Select an additional k action pairs (s, a) at random and update their Q values

$$Q(s_k, a_k) = R(s_k, a_k) + \gamma \sum_{s' \in S} T(s_k, a_k, s') \max_{a'} (s', a') \tag{5}$$

- Choose an action a’ to perform in s’ based on the Q values modified by an exploration strategy

The obvious improvement to Dyna is to prioritize the updates
- Keep track of predecessor states for each state in the Q table
- When a Q value is updated, the amount of update prioritizes its predecessors

Reinforcement Learning Example: Learning by observation

One of the basic problems with Q learning is reaching a reward state in a complex environment

- Training a robot to wall follow
- Training a robot to achieve a goal in the face of obstacles

Robot can explore a long, long time without ever finding a positive reward

What if the robot watches while something else guides the robot?

- Hand-tuned controller
- Person with a joystick

Another problem with Q learning is that the system has to be tabular

- Problematic in a continuous world

What if we train a function to approximate the Q table?

- Can use a neural network (recurrent or not)
- Can use an interpolation function

It turns out that so long as the approximation function does not extrapolate, you're ok

- Locally weighted averaging is OK
- Locally weighted regression (fitting lines to the local data) is OK
- So store a table, but use it as a starting point to interpolate continuous data

Smart & Kalbling strategy: Training has 2 phases

Phase 1:

- Use a joystick or a hand-tuned program to control the robot on a task
- Let the Q-learning system watch and execute updates

Phase 2:

- Give control to the Q-learning system
- Execute the task many more times
- Continue to update the Q table

Findings:

- System initially gets worse in phase 2, then improves beyond the performance in phase 1
- Without phase 1, the system never learns (took 2 hours just to reach the goal once)

E28/CS82 Lecture #36 S05

Overall Topic: Purposive Robots

Reinforcement Learning: Policy Gradient Methods

Sometimes learning takes the form of trying to estimate parameters for a control system

- PID constants for a simple controller
- Parameters for a kinematic system (e.g. and Aibo walking)

There are a number of ways to learn parameters

- Can train an ANN as a controller replacement
- GAs offer a method for stochastic search in the parameter space
- PBIL offers a more gradient-based search in the parameter space than GAs
    - Keep statistics on each bit for the population
    - Each generation use the statistics to generate solutions to test
    - Update the statistics with the top K solutions
- Explicity gradient descent (policy gradient) also seems to work with a good initial guess

Policy gradient concept

- Given a starting parameter vector $V = (p_0, p_1, \ldots, p_N)$
- Repeat for K iterations
    - Generate a new parameter vector $V' = (p_0 + \varepsilon_0, p_1 + \varepsilon_1, \ldots, p_N + \varepsilon_N)$
        - ε_i can take on a small positive, 0, or small negative value, randomly selected
        - Magnitude of the values is consistent
    - Evaluate the controller on V'
- From the K iterations, calculate the partial derivatives for each parameter
    - Identify the slope for each parameter, or whether the 0 case is a local maximum
- Normalize the vector of partial derivatives (slopes)
- Update V using the partial derivative for each parameter that improves performance
    - Use a learning constant to manage the rate of learning

Findings

- Requires a good initial parameter set (likely hand-tuned)
- A poor intial parameter set causes the system to do bad things
- If started in a good location, can find good optimal parameters

Possible improvements

- Simulated annealing approach
    - Move in the optimal direction with some increasing probability over time
- Momentum terms
    - Current motion is, in part, affected by previous motion
    - Generally provides faster initial learning

Making Decisions: Partially Observable Markov Processes

What if we don't actually know what state we're in?

- Transitions between states are probabilistic
- States generate observations
- Observations are probabilistic

A POMDP is

- S: a finite set of states of the world
- A: a finite set of actions
- T: S x A -> PI(S): a state-transition function T(s, a, s')
- R: S x A -> R: reward function R(s, a)
- Ω: a set of observations
- O: S x A -> PI(Ω): observation function O(s', a, o)
    - Probability of an agent seeing o after taking action a and ending up in state s'

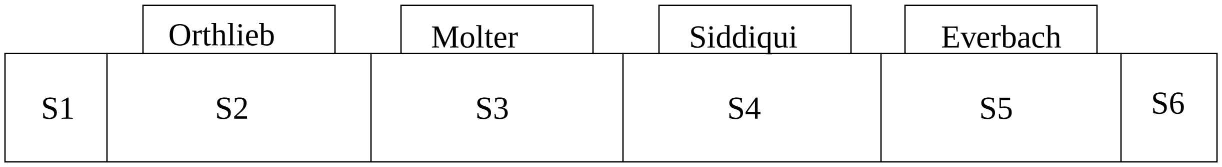
The problem is that we don't know the exact state of the world, but we still need to act

So we keep track of a belief state b

- Probability of being in each state
- Action policy is now a function of the belief state, not a function of the actual state

Example:

Consider a linear problem with a robot trying to find a particular office



There are six states, one for each office and two end corridor states.

There are five possible observations in each state: Orthlieb, Molter, Siddiqui, Everbach, NULL

There are three possible actions to take in each state: read sign, LEFT, and RIGHT

The OCR system on the robot returns the correct symbol with 60% accuracy. There is a 10% probability of returning each of the other symbols, and a 10% probability of returning the NULL symbol. In a state without a sign, the robot returns the NULL symbol with 60% probability, with a 10% probability for each of the non-NULL symbols.

If the robot goes LEFT it does so with 90% probability. There is a 10% probability of staying in the same state. The same likelihoods of success/failure hold for going RIGHT. If the robot just observes, it remains in the same state with 100% probability.

Goal: assume the robot is looking for Siddiqui's office, and has an equal probability of starting in any of the non-goal states. So the belief vector is:

$$b = (0.2, 0.2, 0.2, 0.0, 0.2) \quad (1)$$

Assume our first action is to read a sign, and we observe the symbol Molter. This modifies the belief vector according to the observation and transition probabilities

$$b'(s') = P(s'|o, a, b) \tag{2}$$

$$b'(s') = \frac{O(s', a, o) \sum_{s \in S} T(s, a, s') b(s)}{P(o|a, b)} \tag{3}$$

As with many situations, the denominator in the term normalizes b(s) to one

$$b = (0.1, 0.1, 0.6, 0.0, 0.1, 0.1) \tag{4}$$

Now consider we move RIGHT and read a sign again. This time we again see the symbol Molter.

$$b = (0.01, 0.06, 0.51, 0.31, 0.01, 0.11) \tag{5}$$

Notice that we are still most likely to be in the Molter state, but we may also be sitting in front of Siddiqui's door and just reading the door incorrectly. If we move RIGHT again and observe Siddiqui, then we get:

$$b = (0.0, 0.01, 0.03, 0.85, 0.08, 0.03) \tag{6}$$

So 85% might meet our threshold for deciding to knock on the door. If it's not good enough, we can either try sign reading again, or take more actions to try and improve our confidence.

Optimal Policies for POMDPs

Since the belief state is a sufficient statistic for making decisions (we need no prior information), the optimal policy is the solution of a continuous space "belief MDP"

- B, the set of belief states (continuous space)
- A, the set of actions (same as the POMDP)
- $\tau(b, a, b')$ is the state transition function
- $\rho(b, a) = \sum b(s) R(s, a)$ is the reward function of belief states

So then you just need to solve for the policy of the belief MDP to get the appropriate policy for the POMDP with an infinite discounted horizon.

This is hard to do...

A few observations about the belief space:

- It turns out that the belief vector is a weighting on various policy strategies
  - Each possible action you can take is the root of a policy tree
  - For each state, the nonstationary value of the state is V(s) (optimal policy)
  - The value of any given policy tree is constant
- The value of the current state is the belief for each state multiplied by the state's value
- So the belief state is a linear weighting of the optimal policy tree in each state

The result is that the V(b) function is a convex piecewise hyperplane

- The less entropy in your belief state, the easier it is to choose the optimal policy
- You get less entropy when you are on an extremum of the belief space

For each hyperplane region, the policy to follow is constant over the region

So one possible solution is to generate all possible policy trees and then prune out the ones that do not contribute to the optimal policy surface

- Exponentially hard

So you want to only keep track of the policy trees actually on the optimal policy surface, V(b)

- Dynamic programming task: build up the V(b) hypersurface incrementally

Value iteration

- Start with a horizon of 1, then iterate back in time
- At each time step, figure out the spanning set of policy trees (estimate of V(b) so far
- Continue until the estimate for V(b) doesn't change

One method is to use the "witness" algorithm

- At each iteration, consider each possible action for time t
- Update the existing policy trees by one time step
- Check to see if the current estimate of V(b) is violated by any new policy tree
    - If it is, add the new policy tree to V(b)
- After considering all possible actions in each state, prune the set of policy trees, if possible

In practice, for fairly small t (on the order of 100) the policy converges.

The really cool thing is that once you have a stationary policy, you can actually generate a finite state automata from the POMDP policy (although it's not guaranteed to be small).

- Basically converting a probabilistic method into a deterministic policy to follow
- Like generating guidelines based on experience

The big issue is that POMDPs are hard so solve for numbers of states over 100

- Thrun recently proposed a sampling-based method of identifying V(b)
- Using intelligent sampling of the belief space they demonstrated over 800 states
- Thrun thinks 8000 are possible with better numerical techniques

E28/CS82 Lecture #37 S05

Overall Topic: Social Robots

Robot Emotions

The argument has been made that emotions are actually necessary for intellect
- Sloman and Croucher, IJCAI 1981
- The intellect is driven by multiple overlapping motivations and goals
- The motivations and goals interact with obstacles and other entities in the environment
- Motivations and goals are constantly changing and evolving over time
- Emotions are one method of moving goals up and down the priority list

Robot emotions are also a useful vocabulary to use when characterizing robot behavior
- Can a robot have an emotion without being self-aware?
- People can certainly ascribe emotion to a robot based on apparent behavior
- Robots can use personification by humans to achieve their goals

Explicit modeling of emotions in robotics is useful in a number of contexts
- Social robotics
  - Goal is to improve communication and enhance the human-robot experience
  - Attempting to modify human emotion, positively, by using apparent robot emotion
- Game-play
  - Primary goal is to achieve more realism by letting agents show appropriate emotions
  - Secondary goal is to negatively affect human emotions by using apparent emotions

General Emotion Mechanism (from Maxwell’s 2000 robotics notes)

Set up the robot’s task as a goal, multiple goals, or series of goals
- Give the robot a state
- Give the robot a way to calculate if it has reached its goal
- Give the robot a way to estimate distance to the goal from a state
- Goals can change over the course of a robot’s "performance"

Set up the robot’s inner character
- Set of parameters that may have semantic labels
- These parameters define an emotional state space

Set up a set of attitudes (behaviors)
- Attitudes act as partitions or subsets of the emotional state space
- Access to actions is stochastically determined by the current attitude

Set up a set of actions
- Set up probability of each action given an attitude

Set up an action selection mechanism
- Greedy mechansim: always take the action that gets you closest to the goal
- Stochastic greedy mechanism: add some randomness to the greedy algorithm
- Alpha-beta search: use game-playing techniques to look at predicting responses

Set up the rules for moving through the emotional space

- Inputs to the function are:
    - Current emotional state
    - Current physical state (distance to goal)
    - Latest external input
- Some rules may be gradual
- Some rules may ignore some inputs
- Some rules may drastically alter the emotional state
- Some rules may be independent of the inputs
    - Decay to a set of "equilibrium" parameters
    - Happy person won’t tend to hold anger very long

Emotional Models for Virtual Agents (Broekens and DeGroot, 2004)

FeelMe framework for computational emotions: series of processing stages

1. Decision Support System
    - Extracts information from the environment that is relevant to emotions
    - Some kind of pattern recognition system that identifies relevant events
2. Appraisal System
    - Evaluates the events from the DSS
    - Converts them to modifiers in an emotional state space
    - There may be multiple banks of appraisers in an interconnected network
    - Use Mehrabians PAD (pleasure, arousal, dominance) space (3 dimensions)
    - Output is a stream of N-dimensional vectors in the emotion space
3. Appraisal Signal Modulator [ASM]
    - Performs transformations on the appraisal-results: dampening, or correlation
4. Emotion Maintenance System [EMS]
    - Continuously integrates new emotion vectors into the emotional state
    - Appraisal-results induce changes in the emotional state
    - In the absence of new inputs, the emotional state can drift towards a “personality” state
5. Behavior Modification System
    - Selects, controls, and expresses the agent’s emotional behavior
    - So far this has been used on a limited reportoire of actions

Robot Theatre: the character behind the voice

Most good dramas have a plot driven by characters who make purposeful actions towards a goal

- Heros: characters whose primary goal is what moves the story along
- Villains: characters whose primary goal either will prevent, or is to prevent the heros from achieving their goals
- Outer obstacles: external factors that prevent the heros from achieving their goals
- Inner obstacles: internal factors that prevent the heros from achieveing their goals
- Given circumstances: factors that influence how a character will achieve their goal

If you are trying to execute a task that involves human interaction, treating it as a dramatic situation will help the robot to act appropriately and with character

Example: Improv architecture (Allison Bruce, WIRE 2000)

Elements of the architecture

- Props and goals: physical entities in the room
- Characters with an internal state and inner obstacles that define their personality
- A set of behaviors and actions available to each characters
    - actions are dependent upon behaviors
- A success function that indicates how well the robot has achieved the current goal

Inner obstacles

- A set of relevant basic emotions
- The inner obstacles determine how a character reacts to external stimuli
    - What states it can move to next
- The inner obstacles also affect how the character reacts to the success function history
- Inner obstacles can change or be affected by external stimuli
- Inner obstacles decay back to an equilibrium state
    - equilibrium state determines standard personality

Behaviors

- Behaviors interact with the goals and inner obstacles to limit the appropriate responses
- Each behavior has a set of prototype inner obstacles that define it

Standard main loop

- Receive input: behavior, action
- Update internal state based on the selected action
- Search the list of behaviors to find the one that most closely matches the current state of the characters (inner obstacles)
- Search the available actions and choose the one that appears to be the most successful towards reaching the goal
- Choose a line from the dialog
- Perform the line

Triggering behaviors

- Some input behaviors are defined as "triggering", which means they can significantly modify the inner obstacles
    - e.g. Distracting someone
- A triggering behavior from the transmitting actor generates a mandatory behavior in the receiving actor--if the receiver is in a receptive state.
- The internal obstacle values are set to the "ideal" values of the mandatory behavior

E28/CS82 Lecture #38 S05

Overall Topic: Robot Teams

Robot Teams: Moving and Planning

Methods for decision-making and planning for robot teams lie along a continuum

- Distributed decision-making
- Centralized decision-making
- Hybrid decision-making

Which method to use depends on a number of factors

- Can the robots communicate?
- Is there a central decision-maker available to the system?
- How coordinated to the robot's actions need to be?

Robot soccer is one domain in which a team of robots needs to work together for a single goal

- Motion planning
- Role selection
- Play selection

Motion Planning for Robot Soccer

This is a nice application of some of the kinematics and methods we've looked at

- One difference is that all calculates are done in a single coordinate frame (world)
- Central decision maker is watching the robots via central overhead camera

Concept is to move the robot to a target point specified by (x^*, y^*, ϕ^*)

- Need to approach the target from the proper direction (to kick the ball)
    - θ = angle from robot to target
    - ϕ = orientation of robot
    - v = desired velocity of robot
    - (v_l, v_r) = left and right wheel velocities
    - (c, d) = (clearance parameter, distance to target)

$$\alpha = \theta - \phi^* \quad (1)$$

$$\theta' = \theta + min\left(\alpha, \tan^{-1}\left(\frac{c}{d}\right)\right) \quad (2)$$

$$\Delta = \theta' - \phi \quad (3)$$

$$(t, r) = (\cos^2\Delta \cdot \mathrm{sgn}(\cos\Delta), \sin^2\Delta \cdot \mathrm{sgn}(\sin\Delta)) \quad (4)$$

$$v_l = v(t - r) \quad (5)$$

$$v_r = v(t + r) \quad (6)$$

Result is a trajectory that circles around the target until it gets close to the goal orientation

Obstacle avoidance is integrated into the system at each time step
- Target point is dynamically altered to bring the robot around the target at a certain distance
- Result is slightly discontinuous movement as it reacquires the original target after clearing the obstacle (see Figure 3 of Bowling and Veloso, 98)

Role Selection: decentralized

In the 1998 system, there was little central decision-making
- Each robot was viewed as an independent agent trying to make decisions
- At each time step, they calculated the value of passing or shooting
- Effectively, they used a 1-step lookahead planning system (mostly reactive)

$$V_{\text{pass}} = \frac{P(\text{pass})P(\text{shoot})}{T(f_{\text{pass}})} \qquad V_{\text{shoot}} = \frac{P(\text{shoot})}{T(f_{\text{shoot}})} \tag{7}$$

Role Selection: centralized and dynamic

In the later system (2004) they used the concept of plays and a playbook

A “play” is a set of roles for the individual robots
- Can specify a series of sequential actions in time

A playbook is a series of plays, as well as a set of weights for each play
- System keeps track of how well a play works
- Weights for each play get adjusted based on its performance
- One important factor is to not reduce the weights of specialty plays
  - If a play is not called a lot because it is rarely appropriate, it should not be penalized
- Using a method of adaptation that follows the “Regret” model
  - We first saw this in the Markov decision process topic

Selecting plays
- Each play has a set of prerequisites for the play to be selectable
- Each play has a weight
- Plays can be aborted, succeed, complete, or fail
- Once called, a play has a small amount of hysteresis to avoid oscillations

Selecting Roles
- Roles are dynamically assigned when a play is selected
- The lead role (usually the ball handler or interceptor) is selected first
- The remaining roles are selected in order
- A function calculating appropriateness for each role is used to make the selection
- Roles are not static throughout the play, but can be reassigned
- Again, hysteresis in the system avoids oscillation in role assignment

Role Selection: decentralized and dynamic

The AIBO team took a different approach to role selection
- New rules permitted wireless communication between robots

Still limitations on communication and processing
- No centralized decision-maker
- Don’t want to be sending a lot of data around (i.e. to the goalie)

Static play selection in the sense of having roles
- One ball attacker
- One offensive rebounder
- One defensive rebounder
- One goalie (fixed selection)

Dyanmic role selection was based on bids
- Each robot would send out a fixed amount of information about its situation
- Each robot would compute a “bit” based on the information from each robot
- The lead role would go to the highest bidder
    - Decentralized auction since all robots execute the same bid function on the same data
    - Not necessarily synchonized, but close
    - Use hysteresis in the role selection system to avoid oscillation

Potential fields guide each robot to its goal location given a role
- Obstacles, opponents, and walls have a high potential
- Ball has a low potential for the attacker

E28/CS82 Lecture #39 S05

Overall Topic: Robot Teams

Distributed Robot Exploration

The task of having a group of robots explore and map a new area is important

- Centralized approach: communicate with a central decision-maker
    - Run an optimizer to minimize total energy (or something) while maximizing coverage
    - Basically a travelling salesman problem, and NP-complete
    - For less than 12 searchpoints there are optimal solutions
    - For more than that, there is always simulated annealing and heuristic search
- Decentralized approach: have each robot follow an individual strategy
    - Each robot looks for the closest frontier area
    - Each robot explores that area and then makes a decision about where to go next
    - Without some kind of information sharing, tend to get leaders and followers
        - The lead robot keeps pushing the frontier away
        - The following robots never actually get to uncovered territory

A hybrid approach to the task is to use a market-based approach

- Each robot is trying to optimize its own actions
- Communication distributes the task among the various robots

The basic idea is that each robot is trying maximize its own profit

- Exploring a new area is worth $$
- Time, distance travelled, or energy, are all expenses (time is a proxy for most)
- Each robot wants to maximize its exploration while minimizing time on task

Example: exploring a fixed size area

Give the task of exploring a region to one robot

Each robot, upon receiving a task, can start an auction for pieces of its task

- Any pieces not auctioned off, the robot must execute itself
- There is a slight premium for the robot that actually executes the task in terms of pricing

The robot responsible for the whole task, starts by dividing up the task and auctioning off pieces

- The robots do not all start in the same place
- Different areas will have different cost/reward structures for each robot
- if the robots have differing capabilities, this will also play into the reward structure
- The result of the auction is that the task is divided up between the robots

Robots can continue to divide up and auction off pieces of their task, and the auctions can be dynamic during the exploration.

- An unkown obstacle, for example, could significantly change the costs of exploration

The end result is a distributed decision making system that results in dynamically changing tasks and reasonable optimization of robot resources (note that designing an optimal exploration strategy in this case is not possible because of unkown obstacle configurations).