Kevin Chen

12/17/17

AI for Robotics Notes

**Localization Overview**

Localization

* A robot is lost in an environment and would like to know where it is.
* Can be solved via satellite (GPS), but not very accurate. Not good enough for self-driving cars.
* Self-driving cars record images of road surfaces and use a lot of mathematical techniques to know where exactly it is.
* Localization assumes you have a map of where you are.

Total Probability

* We have a distribution mapping a location to the probability the robot is in that location.
  + Uniform distribution is maximum confusion
* If there are certain identical features we detect and we know where those features should be, we can increase the probability that we are near one of those features, meanwhile decreasing the probability of all the other positions.
* Posterior belief: the belief after a measurement has been taken.
* Convolution: the robot moves in a certain direction and shifts the belief distribution in the x-direction (i.e. the distance axis) by that amount. However, the peaks are shallower because robot movement is uncertain.
* Assuming we get the same measurement after we move, we can multiply the original belief distribution with the new belief distribution to get the new belief distribution of where we are.

Probability After Sense, Compute Sum, Normalize Distribution

* If you sense that you are in front of a red wall, then multiply each position by a number – that number should be significantly larger for positions with a red wall compared to the other positions.
  + The numbers we are multiplying the probability distribution by is called the measurement probability, which is the probability that the sensor is right or wrong.
  + Then normalize the probabilities so that they add up to 1.
* Let be the probability distribution of positions given our observation Z.
* Exact motion: robot moves perfectly. Inaccurate robot motion: there is a chance the robot moves to the incorrect location.

Limit Distribution

* The Limit Distribution is the distribution that the robot converges to when the robot makes an infinite number of moves.
  + Usually is the uniform distribution.

Sense and Move

* Measurements and motion are the routines of “sense” and “move”
* Localization is the iteration of “sense” and “move” given an initial belief.
* Every time the robot moves, it loses information. Every time it senses, it gains information.
* Entropy is the measure of information. It is calculated by:

Sample Sense and Move Code:

p=[0.2, 0.2, 0.2, 0.2, 0.2]

world=['green', 'red', 'red', 'green', 'green']

measurements = ['red', 'red']

motions = [1,1]

pHit = 0.6

pMiss = 0.2

pExact = 0.8

pOvershoot = 0.1

pUndershoot = 0.1

def sense(p, Z):

q=[]

for i in range(len(p)):

hit = (Z == world[i])

q.append(p[i] \* (hit \* pHit + (1-hit) \* pMiss))

s = sum(q)

for i in range(len(q)):

q[i] = q[i] / s

return q

def move(p, U):

q = []

for i in range(len(p)):

s = pExact \* p[(i-U) % len(p)]

s = s + pOvershoot \* p[(i-U-1) % len(p)]

s = s + pUndershoot \* p[(i-U+1) % len(p)]

q.append(s)

return q

for k in range(len(measurements)):

p = sense(p, measurements[k])

p = move(p, motions[k])

print p

Localization Summary

* Belief = probability distribution
* Sense = product of the probability distribution followed by normalization
* Move = convolution, the expected probability of getting somewhere.
* The grid-based localization method or the histogram method involves finding your location on a grid. The problem scales exponentially based on the number of dimensions.
* Histogram filters are approximate filters.

Bayes’ Rule

* Let X be the probability distribution of grid cells and Z be a given measurement.
* Bayes Rule states: .
  + is our measurement probability, is our prior, and is our normalizing constant.
  + .

The Theorem of Total Probability

* Let be the probability of being at cell “i” at time “t”. Then:
  + This statement is known as the theorem of total probability
  + The operation of a weighted sum over all the other variables is known as a convolution.
  + This is used to calculate the new probability distribution after moving.

**Kalman Filters**

Tracking Intro

* We need to track the position and velocity of other cars, bikes, pedestrians, etc. to avoid colliding with them.
* Kalman Filter is a popular technique for estimating the state of the system.
* Unlike Monte Carlo localization (which uses discrete states), Kalman Filters uses continuous states.
* Kalman filters are unimodal distributions, whereas Monte Carlo can use multimodal distributions.
  + There can be more than one local maxima in Marte Carlo, but only one maxima is Kalman filters.
* Basically predicts where the car will be at the next time interval given data from previous time intervals

Gaussian Intro

* Kalman Filters use Gausian distributions (i.e. normal distribution) – a continuous distribution in which the area under the curve sums up to 1.
  + 1-D Gaussian: Consists of the mean and the variance .
* Probability density function:
* is used for normalizing the function.

Variance Comparison

* The less steep the Gaussian curve, the larger the variance (and standard deviation)
  + Variance is a measure of uncertainty.
  + You prefer a curve with less variance.
  + The maximum value of a Gaussian curve is when

Measurement and Motion

* Kalman filters iterate between measurement updates and motion/prediction updates.
  + Measurement updates uses Bayes Rule, which is just a product
  + Motion/prediction uses Total Probability, which is a convolution.

Shifting the Mean, Parameter Update

* When you combine a new Gaussian curve with a new measurement, the new mean will be somewhere between the mean of the new and prior curves.
  + The more certain you are in the new measurement, the more you pull the mean over to the new measurement’s mean.
* The resulting Gaussian is more certain than the two component Gaussians because you get more information. It’s variance is less than the original two.
* Given the prior curve with mean and variance combined with a new curve with mean and variance , the new mean and variance are:

Gaussian Motion

* When you move a distance, you shift the mean by that amount. The variance is higher due to imperfect motion.
  + The amount that you move needs to also be expressed by a Gaussian curve.
* When you sum two Gaussian curves, the new mean is equal to the sum of the two means, and the new variance is the sum of the two variances.

Kalman Filter Code

* If we don’t know anything about the prior, we should initialize the prior with a very large variance so that the measurements have much more of an effect than the prior.

Kalman Prediction

* Able to infer the future position of an object.
* Implicitly finds the velocity to predict the future position of the object.

Sample code (single dimension Kalman filter):

# You move or change by a certain Gaussian distribution.

def update(mean1, var1, mean2, var2):

new\_mean = float(var2 \* mean1 + var1 \* mean2) / (var1 + var2)

new\_var = 1./(1./var1 + 1./var2)

return [new\_mean, new\_var]

def predict(mean1, var1, mean2, var2):

new\_mean = mean1 + mean2

new\_var = var1 + var2

return [new\_mean, new\_var]

measurements = [5., 6., 7., 9., 10.]

motion = [1., 1., 2., 1., 1.]

measurement\_sig = 4.

motion\_sig = 2.

mu = 0.

sig = 10000.

for i in range(len(measurements)):

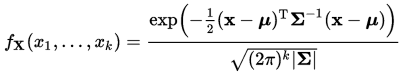
mu, sig = update(mu, sig, measurements[i], measurement\_sig)

mu, sig = predict(mu, sig, motion[i], motion\_sig)

print [mu, sig]

Kalman Filter Land

* Multidimensional Gaussians are Multivariate Gaussians.
  + For a Gaussian with D dimensions, the mean is a vector with D elements and the covariance (replaces the variance) is a D x D matrix.
* This is the formula (though don’t need to memorize it):



* For 2D Gaussians:
  + If the contour diagram circles are small, the uncertainty is small.
  + If the contour diagram is oval shaped, you are relatively uncertain about one variable compared to the other
  + If the contour diagram is oval shaped in a particular direction, the uncertainty is correlated between the two variables.

Kalman Filter Prediction

* Kalman Filter states consist of observable variables and hidden variables.
  + We iterate between taking an observation and then predicting the hidden variable.
  + Taking an observation results in a Gaussian curve that has low variance around the measured variable and high variance around the hidden variables.
  + Predicting (i.e. moving) results in a Gaussian curve that gets correlated between the two variables, starting at the observed value.
  + Then take another observation, and multiply the new Gaussian curve with the prior (from the previous step).
  + Keep iterating until terminating condition.

Kalman Filter Design

* You need two things:
  + State transition function for predicting
  + Measurement function for observing
* You can express the functions via a matrix:
  + State transition function for two variables ( is the observable and is the unobservable) : .
  + Measurement function for two variables ( is the observable, is the unobservable, and Z is the measurement):
  + Notation: in state transition function is called F, and in measurement function is called H.
* Update prediction (after movement):
  + x’ = Fx + u, where x is the mean of the original estimate, F is the state transition matrix, and u is the motion vector.
  + , where P is the original covariance matrix, F is the state transition matrix, and is the transpose of the state transition matrix.
* Update (after) measurement:
  + Error: , where z is the measurement and H is the measurement function.
  + , where R is the measurement noise.
  + Common gain:
  + Now we update the measurement and uncertainty:

* If variables are not correlated, P and r should be initialized to zeroes except for the diagonal entry, which is the initial variance each variable has.
  + Can be off-diagonal elements if variables are correlated.

Sample Code (multidimension Kalman filter):

def kalman\_filter(x, P):

for n in range(len(measurements)):

# measurement update

y = matrix([[measurements[n]]]) - H \* x

S = H \* P \* H.transpose() + R

K = P \* H.transpose() \* S.inverse()

x = x + (K \* y)

P = (I - K \* H) \* P

# prediction

x = F \* x + u

P = F \* P \* F.transpose()

return x,P

Summary/Other Notes

* Vector U is used for adding extra variables into the system.
* Kalman filters are unimodal
* As number of dimensions increase for Kalman filters, the space complexity gets quadratically larger.
* It is an approximate filter only for linear systems.

**Particle Filters**

Overview of Class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **State Space** | **Belief** | **Efficiency** | **In Robotics** |
| **Histogram Filters** | Discrete | Multimodal | Exponential | Approximate |
| **Kalman Filters** | Continuous | Unimodal | Quadratic | Approximate |
| **Particle Filters** | Continuous | Multimodal | (Depends) | Approximate |

* Particle Filters are easy to program.
* Consists of a set of “dots” or “particles” that represent a guess as to where the robot is and where it is headed. The dots comprise an approximate representation for the posterior of the robot.
* The dots that are more consistent with the measurements are more likely to survive.
* Places of high probability will collect more particles and are thus more representative of the robot’s posterior belief.

Creating Particles, Importance Weight, Resampling

* First we initialize a large number of particles randomly.
* Then we take a move and take a measurement, and then weight a particle based on how likely that particle is true.
* The weight of each particle is proportional to the chance it will survive when we resample N particles from the weighted particle list with replacement.
  + You need a function for each particle that takes in the robot measurement and computes the likelihood that the robot is at this particle.
* Of course, you need to normalize the weights in order for you to get the actual probability that they will survive. (The sum of all weights needs to add up to 1.)
* A particular particle might be never sampled (though this often is quite unlikely for a large N and large probability)

Resampling Wheel

* This is a technique used for approximate weighted resampling.
* Pseudocode (for sampling N elements with weights ):

index = random integer between 1 to N

= 0

for i = 1 to N:

# is the largest weight out of

= + random real number between 0 and 2 \*

while w[index] < :

= - w[index]

index = index + 1

select p[index]

Orientation

* You should run the particle filter several iterations to see the results improve over time
  + Measurements of how good the prediction is should be based on the error: the amount of difference between predicted and actual values
* You might get high error if there was no particle initialized to near where the robot is.

Theory of Particle Filters

* Measurement updates:
  + were represented by the particles
  + were represented by the importance weights and resampling
* Motion updates:
  + were represented by the particles
  + is represented by the transition function
  + Summation was represented by moving the particles.

Sample Code:

* Look at the move(), sense() functions in the supplemental homework exercise in the file “ParticleFiltering.py”

Additional Notes

* This class talked a bit about bicycle motion (i.e. car-like motion), where the robot (i.e. car) can only turn its front wheels.
* The supplement homework exercise code (in the file “ParticleFiltering.py”) contains the implementation of how much the x and y coordinates change by after the robot turns and moves by a specified amount.
* The homework exercise code also factors in noisy sensor and movement, which is based on a Gaussian.

**Search**

Motion Planning

* Planning from the start location to the end (goal) location can be seen in the Google Car:
  + Navigate the city streets (like a GPS)
  + Getting onto the correct road. (Maybe not enough room to lane switch, so may require detour)
* Given the map, starting location, end location, and cost of each action:
  + Find the minimum cost path

First Search Program

* Open list: the list of nodes to explore
* g-Value: the number of steps to get to a certain node. Keep a copy of the g-Value for each node in the open list (incrementing it by 1 when going to the successors).
* Closed list: the list of nodes we already explored
* Always expand the node with the lowest g-Value (breaking ties arbitrarily)

A\* Search and heuristics are omitted because it was covered in CS 3600.

Dynamic Programming

* Input: map and goal
* Output: best path from anywhere
* Stochastic environments: actions are non-deterministic due to changing layouts (e.g. cars moving)
  + Thus, we want an action from every grid cell.

Computing Value, Optimal Policy

* The value function is a function that takes in a node and returns the length of the shortest path to the goal.
* The optimal policy tells you what the best action to take (i.e. the action that would result in the shortest path) from each node.

Stochastic Actions

* Certain actions might not always succeed, so we assign a reward to each state and compute the optimal policy based on these rewards.
* Details are omitted because this topic was covered in CS 3600.

**PID Control**

Robot Motion

* Goals of generating smooth paths and control (i.e. PID control)
* We want to convert the paths with sudden right-angle turns into a smooth path that a car would actually be able to go over.

Smoothing Algorithm

* We have a path in a grid that we want to smoothen
* Denote each point in the path as
* Smoothing Algorithm:
  + Set
  + We then want to minimize
* Minimizing ensures that we stay close to the original path.
* Minimizing ensures that we make the points as close as possible to each other.
* We assign a weight to the terms:
  + The larger alpha is, the smoother the path is. The smaller the alpha, the closer to the original path.
* We make sure that the start and goal cannot be changed (i.e. and )

Path Smoothing

* Use gradient descent to minimize
  + Minimize by modifying : . Note that is different than the above . Set this to 0.5
  + Minimize by modifying :
* We can combine the above two steps into one instruction:
  + We will set to 0.1

Sample code:

from copy import deepcopy

def printpaths(path,newpath):

for old,new in zip(path,newpath):

print '['+ ', '.join('%.3f'%x for x in old) + \

'] -> ['+ ', '.join('%.3f'%x for x in new) +']'

path = [[0, 0], [0, 1], [0, 2], [1, 2], [2, 2], [3, 2], [4, 2], [4, 3], [4, 4]]

def smooth(path, weight\_data = 0.5, weight\_smooth = 0.1, tolerance = 0.000001):

newpath = deepcopy(path)

change = 2 \* tolerance

while change > tolerance:

change = 0

for i in range(1, len(path) - 1):

for j in range(len(path[0])):

original = newpath[i][j]

newpath[i][j] += weight\_data \* (path[i][j] - newpath[i][j]) + weight\_smooth \* (newpath[i-1][j] + newpath[i+1][j] - 2.0 \* newpath[i][j])

change += abs(newpath[i][j] - original)

return newpath

printpaths(path,smooth(path))

PID Control

* You want your car to go along a certain trajectory (called the reference trajectory). How much should you steer?
* The crosstrack error is the lateral distance between the vehicle and the reference trajectory.
* The amount you steer should be proportional to the crosstrack error.

Proportional Control

* The robot will overshoot if you just steer proportional to the crosstrack error. (The constant is called .) The robot will never be able to actually get on the reference trajectory (instead oscillating on both sides of the reference trajectory):
  + , where is the turning angle.



* This is called marginally stable, or stable for short.
* Increasing makes it oscillate faster.

PD Control

* Goal of eliminating the oscillating.
  + Need to select correct constants for proportional gain and differential gain

Systematic Bias

* The car “drifts” at a certain angle when the wheel is completely straight due to mechanical issues, defects, etc.
* Adding a certain drift will cause the robot to converge onto a new parallel axis.

PID Implementation

* PID tries to eliminate this bias
  + After a while, if there is a lot of CTE’s accumulated, the robot will try to correct that.

Twiddle

* Aka Coordinate Ascent
* Used for finding good control gains (i.e. , , and )
  + We measure this based on the average crosstrack error.
* Algorithm (where run is the function that returns the average CTE when running):

p = [0, 0, 0] # These are our three parameters

dp = [1, 1, 1]

best\_err = run(p)

while sum(dp) > small\_threshold:

for i in range(len(p)):

p[i] += dp[i]

err = run(p)

if err < best\_err:

best\_err = err

dp \*= 1.1 # make the numbers in dp a little bit larger

else:

# Try the other way.

#We added dp[i] before, so we subtract 2 \* dp[i]

p[i] -= 2 \* dp[i]

if err < best\_err:

best\_err = err

dp \*= 1.1

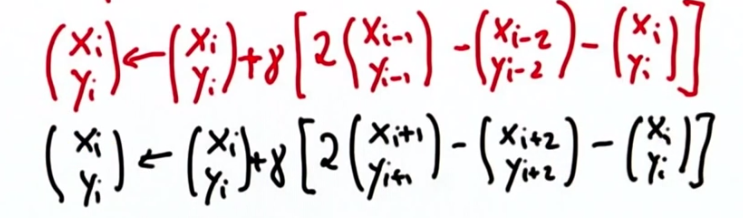
else:

p[i] += dp[i]

dp[i] \*= 0.9

Advanced Smoothing

* If you want the smoothing algorithm to factor in two adjacent points in each direction for each point, then append the following algorithm:



* Update code:

newpath[i][j] += weight\_smooth \* (newpath[(i-1)%len(path)][j] + newpath[(i+1)%len(path)][j] - \  
 2.0 \* newpath[i][j]) + \  
 (weight\_smooth / 2.0) \* (2.0 \* newpath[(i-1)%len(path)][j] - \  
 newpath[(i-2)%len(path)][j] - newpath[i][j]) + \  
 (weight\_smooth / 2.0) \* (2.0 \* newpath[(i+1)%len(path)][j] - \  
 newpath[(i+2)%len(path)][j] - newpath[i][j])

Gradient Descent

* A gradient is a vector pointing in the direction of greatest increase. It is perpendicular to the level curve/level surface. Its magnitude depends on how fast the increase is (the faster the increase, the greater the magnitude)
  + The new value (b) is equal to the original value (a) after taking a step in the direction of greatest decrease () by a weighting factor
* Our goal in this course was to minimize
  + We minimize this with gradient descent
  + By using the equation with , we get:

**SLAM**

Introduction

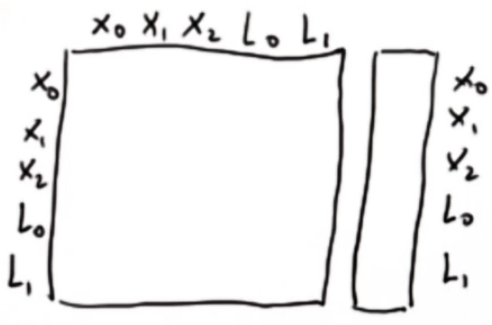
* Simultaneous Localization and Mapping
* Used if you don’t know where you are (and need to localize) and you don’t know the map (need mapping)
* Graph SLAM is an easy and effective method.

Graph SLAM

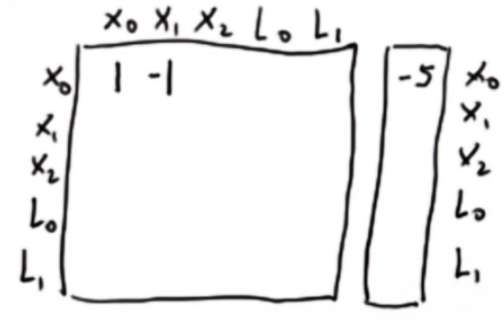
* Factors in:
  + Initial Location Constraints (where you start)
  + Relative Motion Constraints (the motion you make relative to the previous motion)
  + Relative Measurement Constraints (the sensor data you gather)
* You can think of these constraints as “rubber bands” between each position, with the distance constraints being inversely proportional to the tightness of the rubber bands.

Implementing Constraints, Adding Landmarks

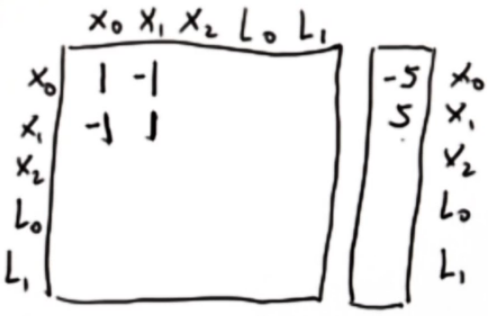
* Create a matrix and a vector placed side by side as shown below.
* Label each row and column of the matrix with poses (x) and landmarks (L). Do the same for each row of the vector. Example:



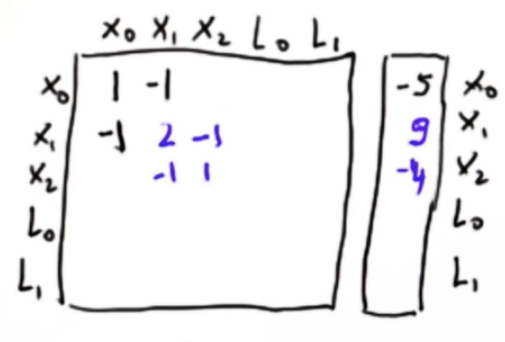
* You assume all elements are zero.
* Now update the matrix for each motion and landmark sensed as described below.
* Landmarks (example):
  + You move from to by moving right 5 units. In other words,
  + We can write this as , so we add 1 to the coefficient of , subtract 1 from the coefficient of , and subtract 5 in the entry of the vector:



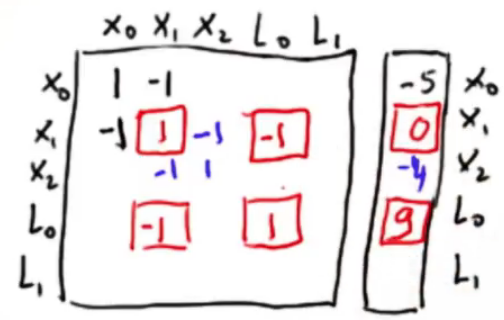
* + We can also write it as , resulting in this update:



* + You now move from to by moving left 4 units. In other words,
  + , so we add 1 in the entry of the row, subtract 1 in the entry of the row, and add 4 in the row of the vector. This results in 2, -1, 9.
  + , so we add 1 in the entry of the row, subtract 1 in the entry of the row, and subtract 4 in the row of the vector. This results in -1, 1, -4



* Landmarks are handled basically the same way
  + Example: If we sensed 9 units away from , then :



Matrix Modification

* Define where the original robot position is. Example: . If that’s the case, we update the entry of the row to be 1, and the entry of the vector to be 0.

Omega and Xi

* The matrix is called and the vector is called
* Let
* is the vector that describes the best estimate for all the robotic locations and landmark locations.

Introducing Noise

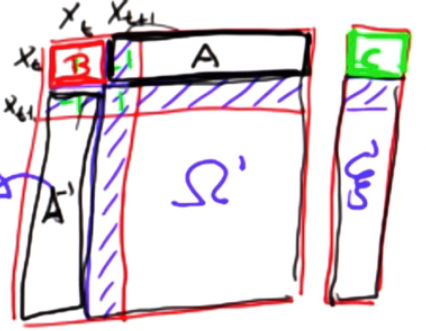
* If you give some inconsistent results (e.g. move right by 1 decreases the measured distance to a landmark by 2), this algorithm will readjust the distances between each position and landmark based on the tightness of the “rubber bands”

Confidence Measurements

* If you are more confident about a measurement, then the amount you change the values in the matrix should all be multiplied by a certain constant (greater than 1)
* The strength factor (the constant you multiply) should be , or the inverse of the standard deviation
* Take a look at “GraphSLAM” python file for example code.

Online SLAM

* Graph SLAM may be inefficient as it is for N steps (but the map defined by certain landmarks is a fixed size)
* Online SLAM only requires you to remember the last step.
* The Omega matrix and Xi vector look like this now (for 1 dimension):
* When moving from to , we expand the matrix and vector. Then we apply the normal increments as we do for Graph SLAM:
* Now we need to convert this matrix back to the original dimensions. We have three matrices B, A, and C and original matrices and . We want to calculate the new and



* These are the equations for and :
* Take a look at “OnlineSLAM” Python file for example code.