6.270 Lecture 4

Localization/Navigation

Scott Bezek
January 2011

Corrections/Notes

- “8-hole pulley” → “6-hole pulley”
- Dual differential: should have mentioned that it's weaker than a standard differential drive since single motor provides all of the torque driving forward or backward rather than two motors

Putting things together

- Yesterday, we saw how to drive in a certain direction
- In order to drive somewhere, must know where we are first (localization)
- Also want high level control of robot: should be able to say moveToPoint(x,y) (navigation system)

Localization

- Difficult to navigate unless you know where you are at all times
- Tough problem:
  - Sensors noisy
  - Small errors can lead to large problems:
    - A few degrees of error can lead to 1ft of inaccuracy if you drive across the board

A peek at localization...

- Dead reckoning: Estimate your own position based on previous estimated position and amount of change
- How?
  - Encoder – distance
  - Gyro – direction
  - Distance sensors?
  - Accelerometer?
- Why?
  - VPS updates infrequently
  - VPS updates are old (latency)
  - VPS heading isn't extremely accurate

A peek at localization...

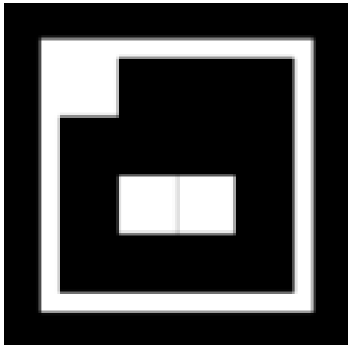
- We want to update our estimated position: x and y
- At each time step: (pseudocode)
  - dist = encoder_read(ENC_PORT) * CONV_FACTOR
  - encoder_reset(ENC_PORT)
  - x = x + dist*cos(theta) //use old heading
  - y = y + dist*sin(theta)
  - theta = gyro_get_degrees() % 360 //update cur heading

Better localization possible?

- It doesn't make sense to just ignore the VPS
- Best of both worlds?
- Dead reckoning:
  - Accurate short-term; fast updates
  - Relative changes
  - Reliable, smooth data (but drifts)
- VPS:
  - Accurate long-term (no drifting)
  - Absolute positioning
  - Potential outages, dropped packets, jitter

How does VPS work?

- Fiducial pattern on top of your robot
- Camera mounted above playing field that tracks these patterns
- Wirelessly transmits your location to your robot



Use VPS data too...

- Let's add some code to handle the VPS too
- When a VPS update arrives:
  - x = vps_data.x
  - y = vps_data.y

- This would mean VPS data is 100% trusted, since it overwrites our dead reckoning estimated position...

Merge VPS data w/ dead reckoning

- One idea: weight VPS data and combine with existing dead-reckoning data
- When a VPS update arrives:
  - //calculate a confidence weight
  - confidence = (255 – abs(motor_vel)) / 255.0
  - x = confidence*vps_data.x + (1-confidence)*x
  - y = confidence*vps_data.y + (1-confidence)*y

- Better, but what about latency?

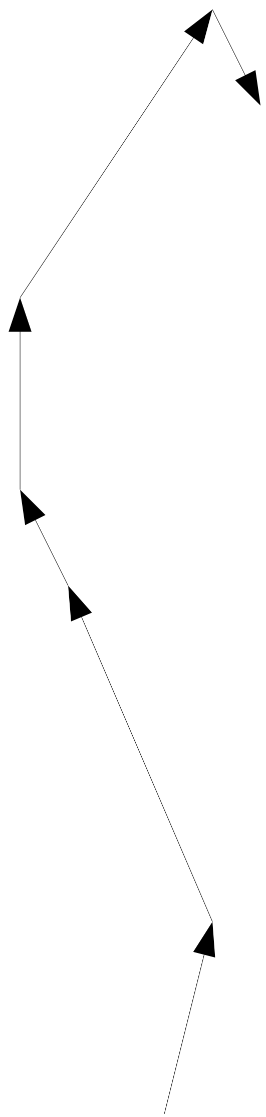
Dealing with latency

- VPS data is inherently old – when it says “you are at (x,y)” think of it as actually saying “*300ms ago* you were at (x,y)”
- If we store history of distance travelled and rotation amount (from dead-reckoning), can reconstruct path taken since VPS snapshot
- Apply this path to the VPS snapshot data to get an accurate estimate of where we are now

Keeping path history

- Store a history of dead-reckoning updates (ring buffer)
- At each time step:
  - dist = encoder_read(ENC_PORT)*CONV_FACTOR
  - encoder_reset(ENC_PORT)
  - x = x + dist*cos(theta)
  - y = y + dist*sin(theta)
  - newTheta = gyro_get_degrees() % 360
  - dTheta = newTheta – theta
  - theta = newTheta
  - add_to_history(dist, dTheta, current_time())

Path History Example



dist	dTheta	time
4	30	1000
7	0	1051
2	-12	1103
4	-12	1157
6	-110	1202

Applying path history

- Given the VPS x,y,theta, apply path history to get a more accurate estimate of current location
- Pseudocode:
  - Let data_time = time that the VPS snapshot represents = vps_data.timestamp - 300ms
  - Look in path history to find first entry newer than data_time
  - Apply distance and dTheta to current location estimate
  - Repeat previous step until at end of history

A peek at localization...

- When a VPS update arrives:
  - //calculate a confidence “weight”
  - confidence = (255 – abs(motor_vel)) / 255.0
  - data_time = vps_data.timestamp – 300 //300ms latency
  - dx_since_data = get_total_dx_since(data_time)
  - dy_since_data = get_total_dy_since(data_time)
  - vps_x = vps_data.x + dx_since_data
  - vps_y = vps_data.y + dy_since_data
  - x = confidence*vps_x + (1-confidence)*x
  - y = confidence*vps_y + (1-confidence)*y

Basic Localization

- Just created basic sensor fusion localization code!
- Could get more advanced (e.g. Kalman filters)

- Now that we know where we are, let's go somewhere!

Let's build a nav subsystem!

- Goal: package navigation/locomotion into self-contained system
- Navigation should run in the background (use threading) so that high level code doesn't need to worry about PID updates or dead-reckoning at all
- Abstraction!

What should it do?

- High-level functions to drive around:
  - moveToPoint(x, y , fwd_speed, tolerance)
  - turnToHeading(heading, ang_speed, tolerance)
  - turnToPoint(x, y, ang_speed, tolerance)
  - moveStraight(fwd_speed)
  - StopMoving()
  - isMoving()
- Keep track of state of navigation system:
  - MOVING_TO_POINT
  - TURNING_TO_HEADING
  - MOVING STRAIGHT
  - STOPPED

Why is this nice?

- Clean, easy-to-read code – drive in a square:
  - moveToPoint(0,0, VEL, TOL)
  - while (isMoving()); //loop until stopped
  - moveToPoint(100,0, VEL, TOL)
  - while (isMoving());
  - moveToPoint(100,100, VEL, TOL)
  - while (isMoving());
  - moveToPoint(0, 100, VEL, TOL)
  - while (isMoving());
  - moveToPoint(0,0, VEL, TOL)

Start from the bottom

- At the lowest level, we need to set left/right motor velocities
- We would rather set forward/angular velocities – then we can have a rotation PID controller and a proportional forward velocity controller
- For moveToPoint(), we'll use both rotationPID and forward controller simultaneously
- For turnToPoint(), we'll only use rotationPID

Setting up a nav system

- Imagine we have some “global” nav system state:
  - Float goalX
  - Float goalY
  - Float goalTheta
  - Int goalFVel
  - Int goalAVel
  - Int state = STOPPED

Setting up a nav system

- Then high-level functions are simple – just need to set state variables for background navigation system to read
- Void moveToPoint(x, y, fVel, tolerance)
  - GoalX = x
  - GoalY = y
  - GoalVel = fVel
  - GoalTolerance = tolerance
  - State = MOVING_TO_POINT
- Void turnToHeading(heading, aVel, tolerance)
  - GoalTheta = heading
  - GoalVel = aVel
  - GoalTolerance = tolerance
  - State = TURNING_TO_HEADING
- Void turnToPoint(x, y, aVel, tolerance)
  - heading = atan2(currentY - y, currentX – x)
  - turnToHeading(heading, aVel, tolerance)

The Navigation Process

- Main navigation loop (runs in background):
  - while(true){
    - getLocation() //dead-reckoning and VPS
    - If (state == TURN_TO_HEADING)
      - desiredHeading = goalHeading
    - else
      - desiredHeading = … //use trigonometry based on goalX, goalY...
    - setRotationPIDGoal(desiredHeading);
    - UpdateRotationPID(); //sets desiredAVel
    - If (state == MOVE_TO_POINT)
      - DesiredFVel = … //proportional to distance to goalX,goalY
    - Else
      - DesiredFVel = 0
    - LeftVel = desiredFVel + desiredAVel
    - RightVel = desiredFVel – desiredAVel
    - motor_set_vel(0, LeftVel)
    - motor_set_vel(1, RightVel)
    - If (state == MOVE_TO_POINT && distToGoal() < GoalTolerance)
      - State == STOPPED
    - If (state == TURN_TO_HEADING && headingError() < GoalTolerance)
      - State == STOPPED
  - }

Minor details

- Add locks to avoid race conditions
- If heading error too large, perhaps limit forward velocity until pointed in the right direction

Upcoming Events

- No big events today - work on your robots!
- Lecture tomorrow: Designing for Failure – 11am
- Control Systems workshop tomorrow at 3pm – have your robot ready to drive
- HappyBoards are just about ready