Localization/Navigation

6.270
January 2012

Miscellaneous Notes

- Chargers – currently in Bethpage, NY – we have temporary ones you can borrow if needed
- Motor batteries – if you haven't gotten a motor battery, do it today! (see me)
- Vision position system is almost ready. It's looking like ~17 updates per second – wireless modules and details coming later today
- Drop test – do it earlier – it gets harder to pass as you add more components and make it heavier
- Useful functions: #include <math.h>
  http://www.nongnu.org/avr-libc/user-manual/math_8h.html

Team Check-ins Tomorrow

- 5 minutes
- We'll check your progress and answer any questions
- Bring your robot
- Be on time!

- Office hours afterward if you have more in-depth questions

- Team 1: 1:00
- Team 2: 1:05
- Team 3: 1:10
- Team 4: 1:15
- Team 5: 1:20
- Team 6: 1:25
- Team 7: 1:30
- Team 8: 1:35
- Team 9: 1:40
- Team 10: 1:45

- Team 11: 2:00
- Team 12: 2:05
- Team 13: 2:10
- Team 14: 2:15
- Team 15: 2:20
- Team 16: 2:25
- Team 17: 2:30
- Team 18: 2:35
- Team 19: 2:40
- Team 20: 2:45

- Team 21: 3:00
- Team 22: 3:05
- Team 23: 3:10
- Team 24: 3:15

Notes: Mock Competition Monday

- Starts at 7pm
- Format:
  - 1 robot at a time
  - On floor in lab – 8x8ft square area
  - Computer sends goal coordinate
  - When robot reaches goal (~2in tolerance), gets a new goal point, and so on
  - Score: # of goals reached in 2 minutes
- Compete as many times as you want before 10pm
- Highest score of the night wins
- Small prizes for top 3 teams

Putting things together

- Yesterday, we saw how to drive straight or turn to a direction
- In order to drive somewhere specific, 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

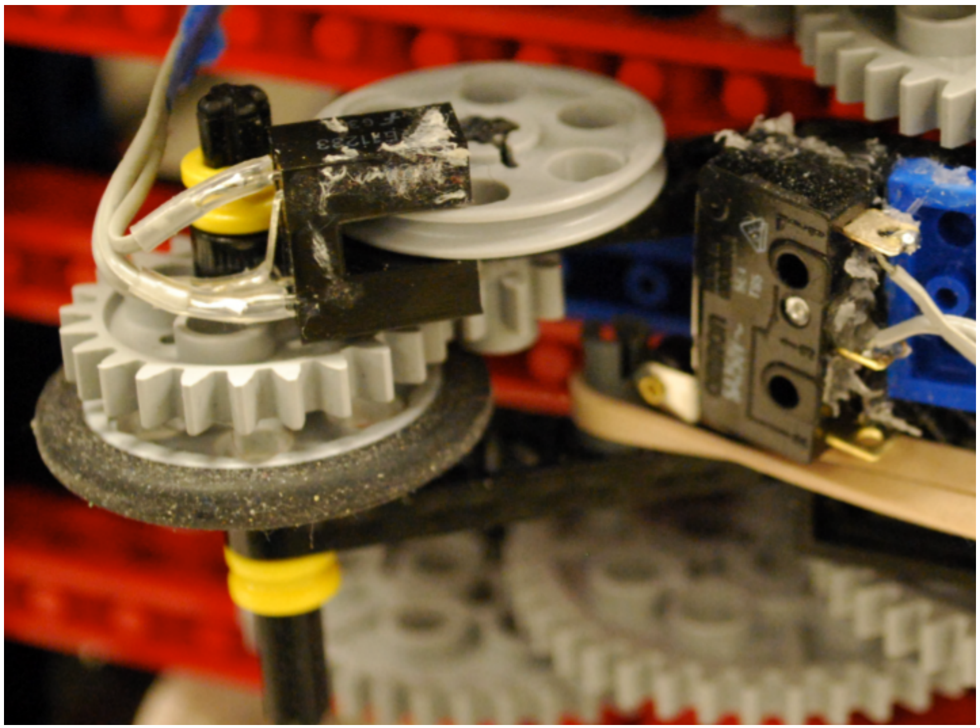
- Difficult to navigate unless you know where you are at all times
- Tough problem:
  - Sensors noisy
  - Small errors can lead to large problems:
    - A few degrees of error can lead to 1ft of inaccuracy if you drive across the board

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() *mod* 360 //update cur heading



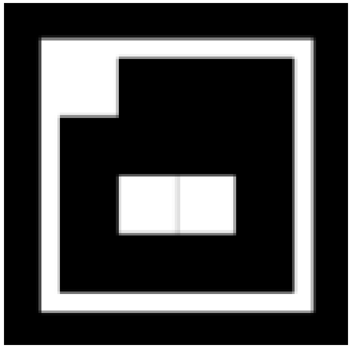
- Sidenote:
  - Encoders on drive wheels measure how far motors spin, not actual distance travelled
  - Drive wheels may slip!
  - Consider a free-wheel encoder – only measures **actual** distance

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...

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

  (this is pseudocode – actual data structure will differ)

- This would mean VPS data is 100% trusted, since it overwrites our estimated position (x,y)

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?

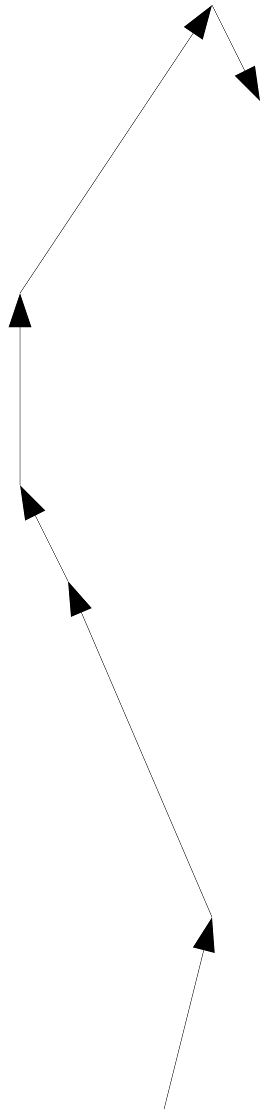
Dealing with latency

- 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

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