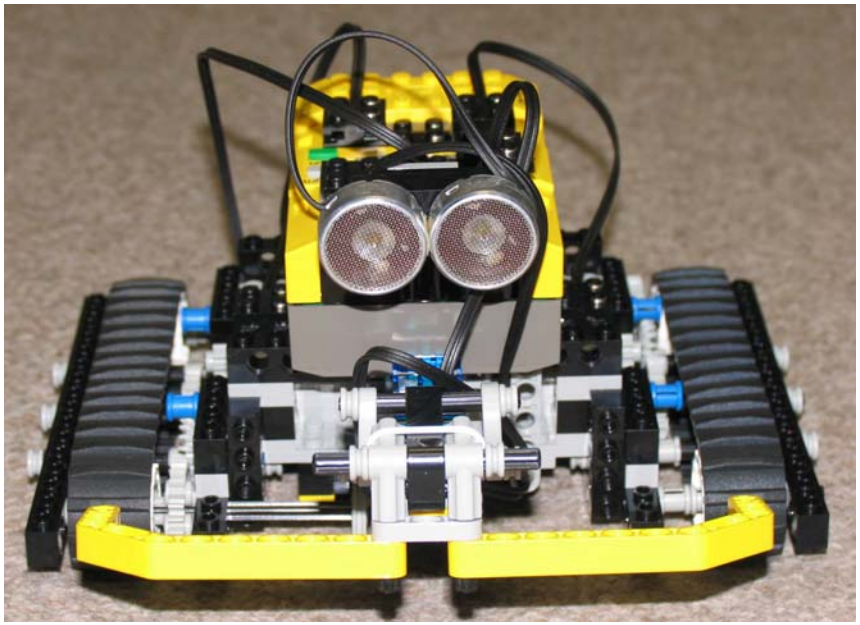
Probabilistic Localization with the RCX

Lloyd Greenwald

(www.cs.drexel.edu/~lgreenwa)



Outline

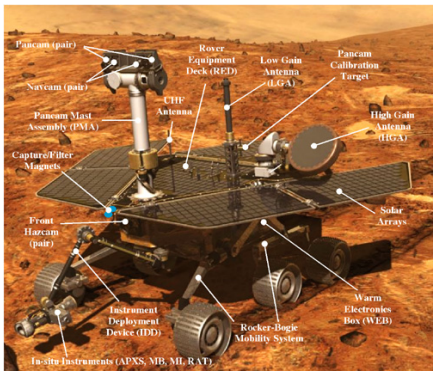
- Overview
  - The localization problem
  - A simplified educational challenge and RCX solution
- Teaching the solution
- Example solution demonstration
- Hands-on Lab

Localization: "Where am I?"

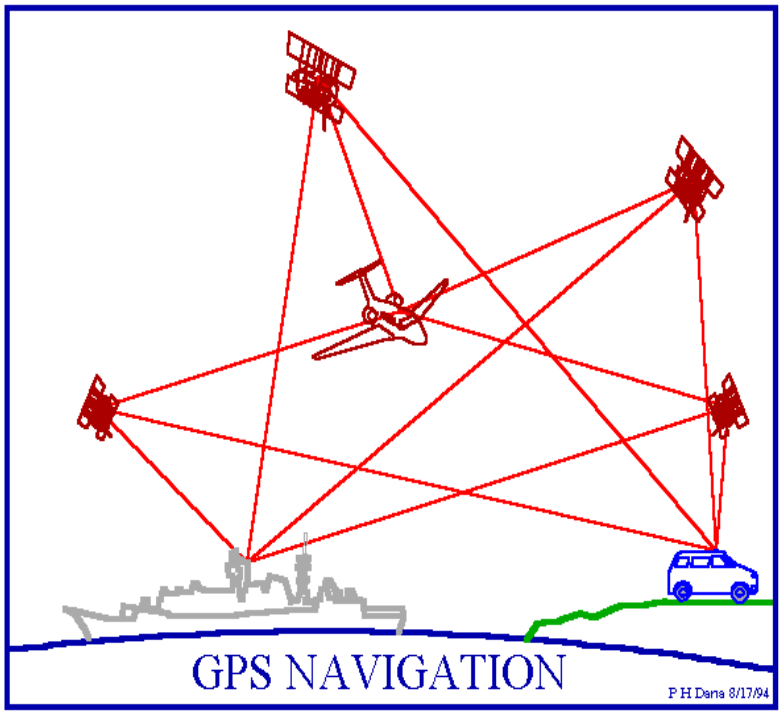
- In many robot applications it is useful for the robot to know where it is relative to features in the world
  - Machine locations in a manufacturing shop
  - A kitchen in a home
  - Known mines in a battlefield
  - Prized flowers on a lawn
  - Battery charging stations

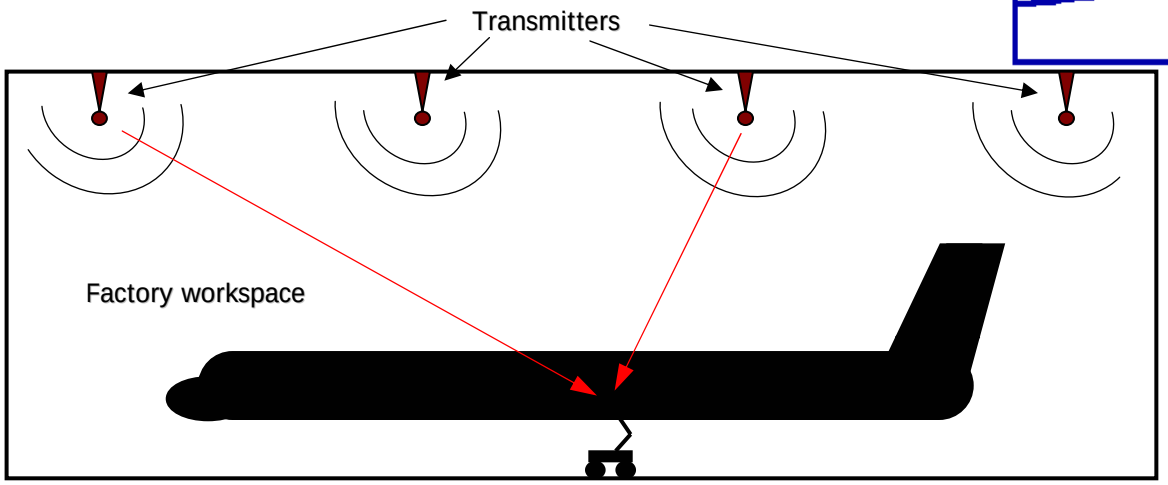
- How does a robot localize in the world?

- Some options:
  - GPS
    (Global Positioning System)
  - Odometry
  - Landmarks/sensors



Global Positioning (GPS)





(from Dana, UTAustin and Frankel, GT)

Varying levels of accuracy, size, cost

Not suitable for small, mobile, low-cost robots

Odometry

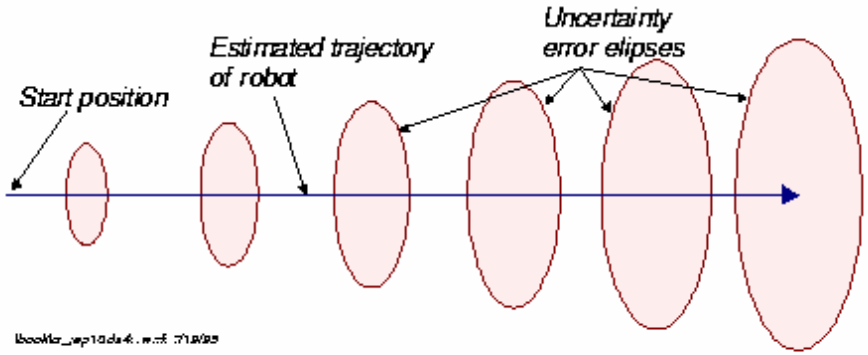


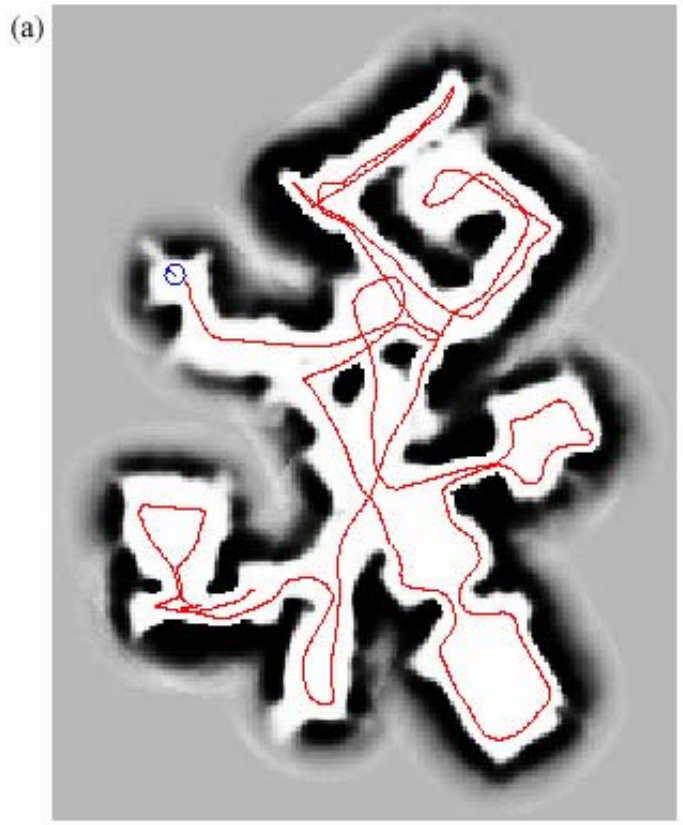
Figure 5.1: Growing "error ellipses" indicate the growing position uncertainty with odometry. (Adapted from [Tonouchi et al., 1994].)

(From Borenstein et. al.)

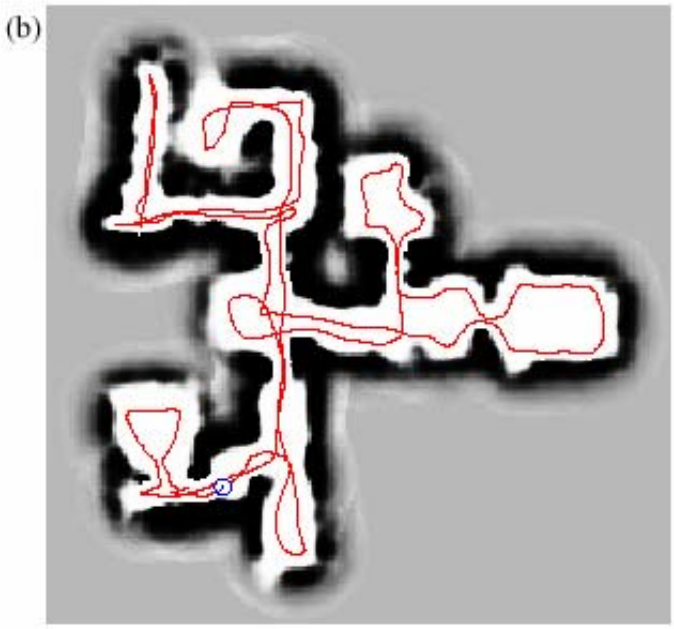
- Given:
  - Starting point
  - Motor commands, sensor values
  - Model of robot drive system and geometry
- Compute:
  - Current point using kinematics

- Problems:
  - Predicted movement varies from observed movement
    - Calibration errors, uncertain robot geometry
    - Friction, wheel slippage
  - Errors grow without bound unless periodic absolute position corrections from other sensors used

Building Maps With Odometry



Bad odometry



Good odometry

(From Thrun et. al.)

Landmarks

- Artificial: distinctive landmarks placed at known locations in environment
- Natural: distinctive features already in environment (known in advance or learned)
- First recognize landmark, then use with map to localize

- Problems:
  - Requires modified environment or unique features in environment
  - Noisy sensors have difficulty finding landmarks
  - Natural landmarks may be ambiguous

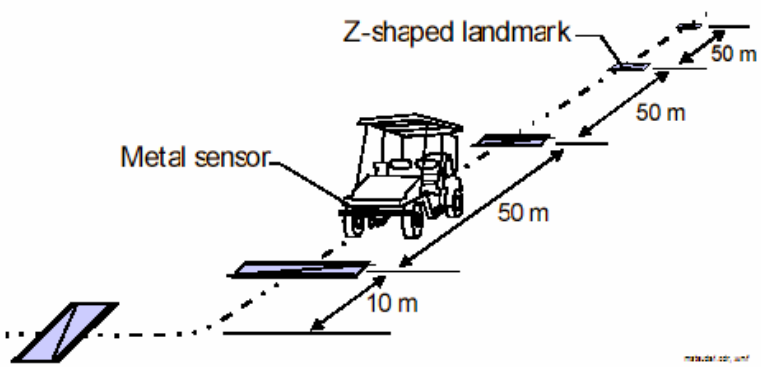


Figure 7.7: Komatsu's Z-shaped landmarks are located at 50-meter (164 ft) intervals along the planned path of the autonomous vehicle. (Courtesy of [Matsuda and Yoshikawa, 1989].)

(From Borenstein et. al.)

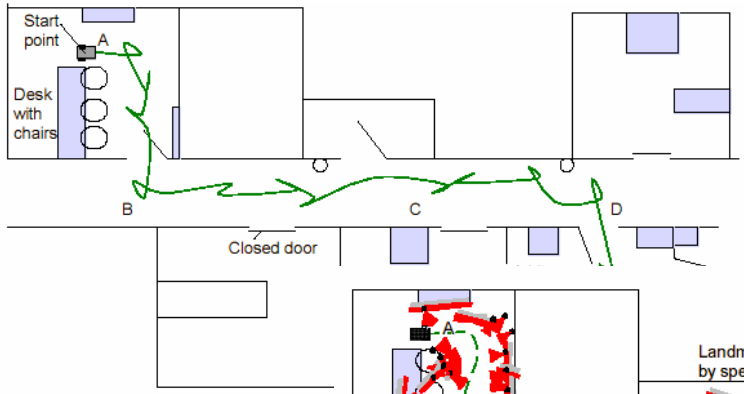


Figure 8.12: Actual office environmer exploratory travel phase. (Courtesy of

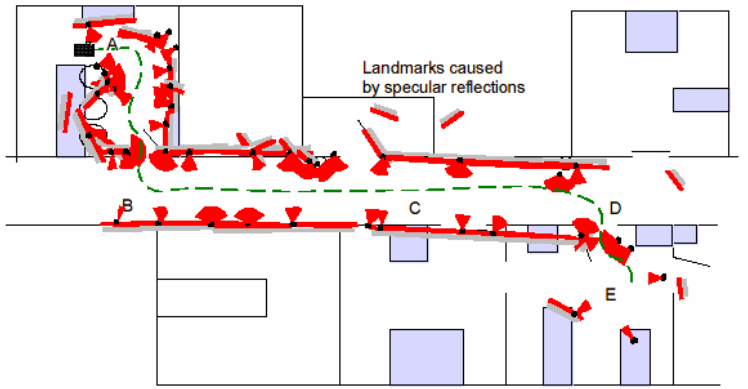
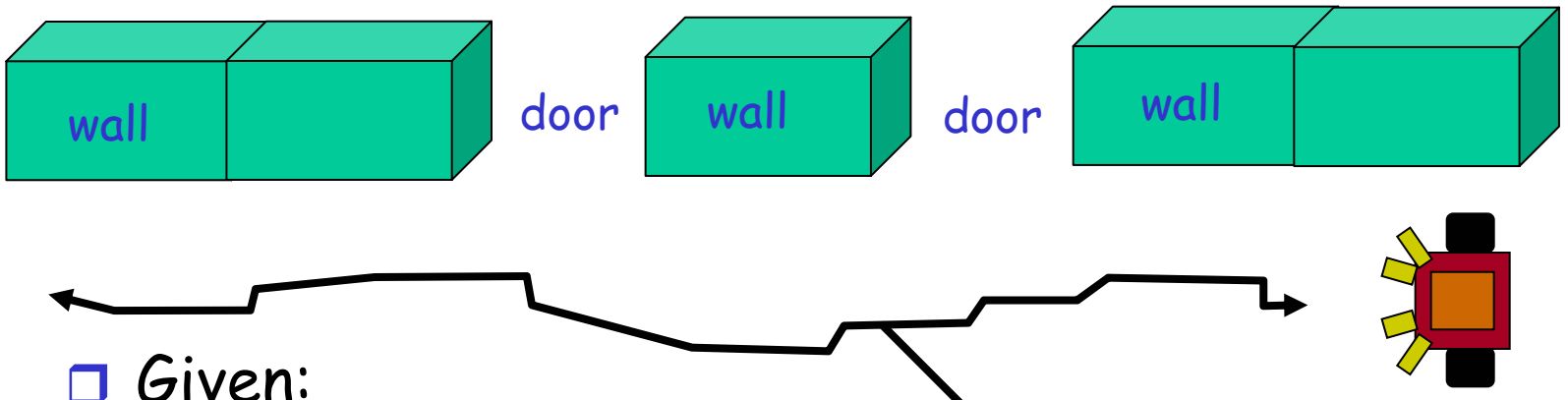


Figure 8.13: Gathered features and robot's return trajectory (Courtesy of [Bauer and Rencken, 1995]).

The Challenge: Simplified Localization

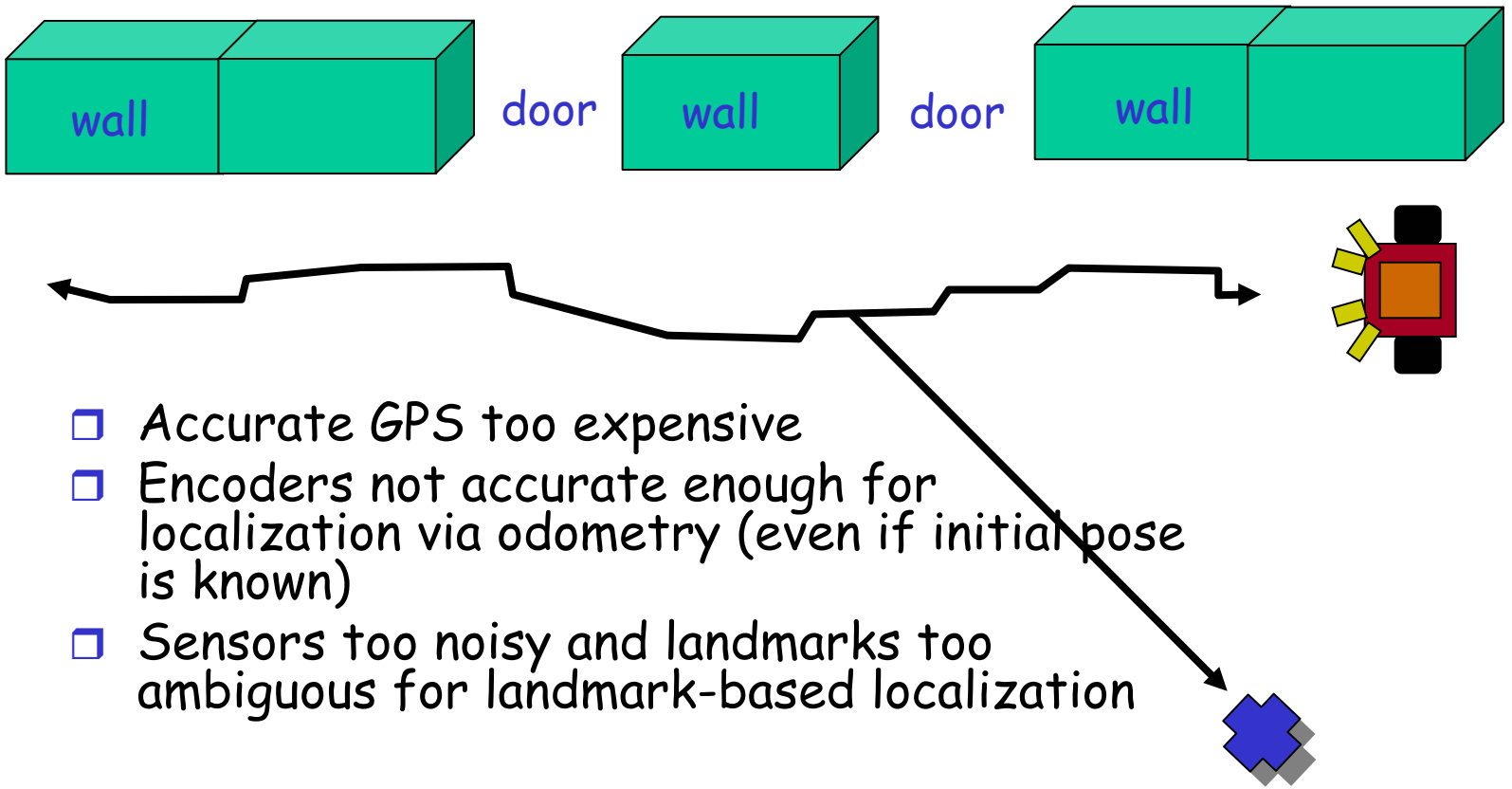


- Given:
  - Known map (with coordinates)
  - (Un)known initial position (x,y) and orientation (θ) - *pose*
- Implement:
  - Find and align with wall
  - Move along wall
  - Determine location
  - Go to known goal point (x,y)

Probabilistic Lo

Adapted from:
"Adapting the sample size in particle filters through KLD-sampling" by Dieter Fox. International Journal of Robotics Research (IJRR), 22(12), p 985-1004, December 2003

Potential Solutions



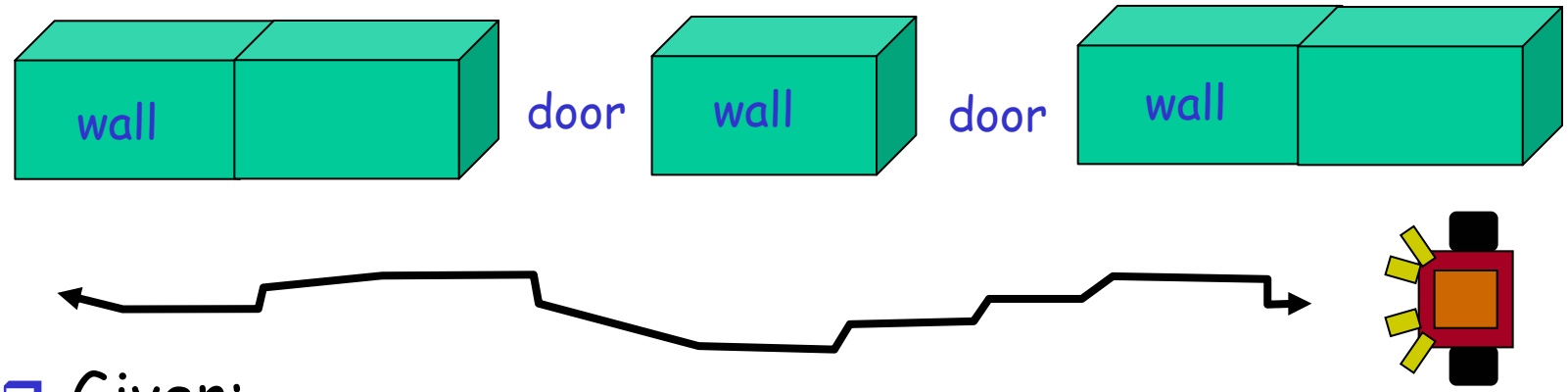
- Accurate GPS too expensive
- Encoders not accurate enough for localization via odometry (even if initial pose is known)
- Sensors too noisy and landmarks too ambiguous for landmark-based localization

- Combine both encoders and sensors with smart algorithms: **probabilistic localization**

Teaching Probabilistic Localization

- Instructions
- Background material
- Implementation details and tips

Probabilistic Localization with Particle Filtering



- Given:
  - Initial pose (optional)
  - Map of environment features/landmarks
  - Sequence of movement actions, over time
  - Sequence of sensor readings, over time
  - Model of movement uncertainty
  - Model of sensor reading-landmark associations
- Compute:
  - Probability distribution over possible current poses (belief state)

Probabilistic Localization with Particle Filtering: Steps

- Implementation details:
  1. Find and align with wall (optional)
  2. Move along wall
  3. Recognize ends of course
  4. Recognize doors/walls
  5. Calculate odometry estimates of movement (using encoders or timing)
  6. Maintain orientation using closed-loop feedback, either sonar or encoders (optional – or use single drivetrain)
  7. Use particle filtering to update probability distribution over locations
- Evaluation:
  - Continuously display most likely position (or display full distribution off-board)
  - Go to known goal point (x,y) (optional)

Teaching Probabilistic Localization

- Instructions
- Background material
  - Review odometry
  - Review sonar
  - Particle filtering for probabilistic localization
- Implementation details and tips

Odometry

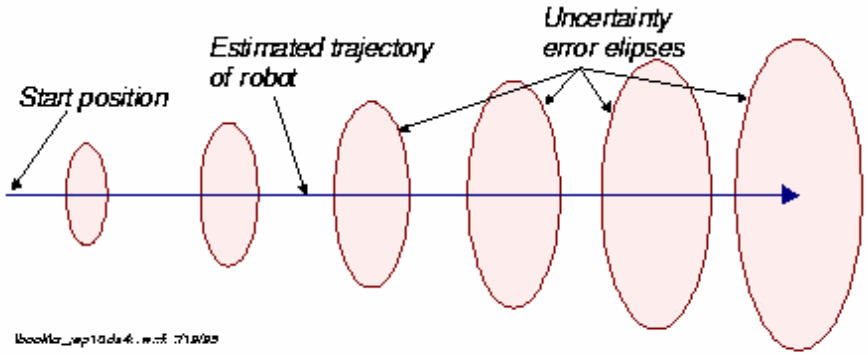


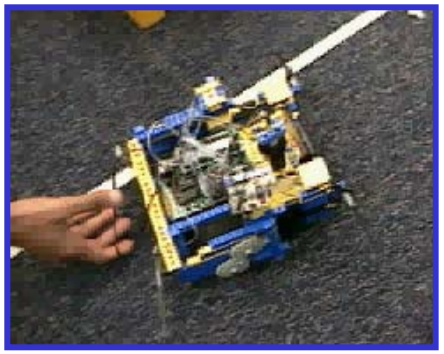
Figure 5.1: Growing "error ellipses" indicate the growing position uncertainty with odometry. (Adapted from [Tonouchi et al., 1994].)

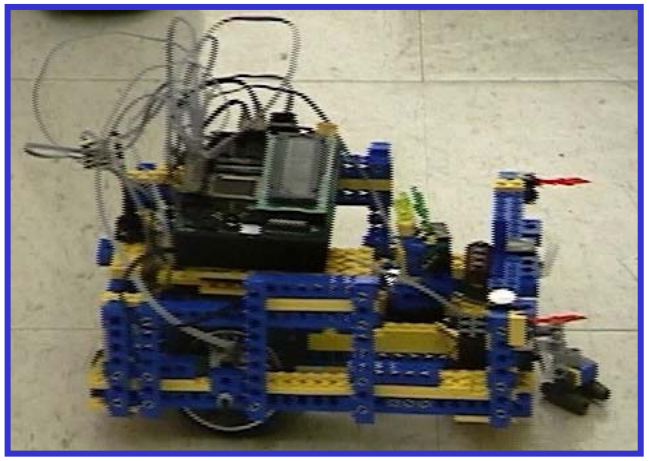
(From Borenstein et. al.)

- Given:
  - Starting point
  - Motor commands, sensor values
  - Model of robot drive system and geometry
- Compute:
  - Current point using kinematics

- Problems:
  - Predicted movement varies from observed movement
    - Calibration errors, uncertain robot geometry
    - Friction, wheel slippage
  - Errors grow without bound unless periodic absolute position corrections from other sensors used

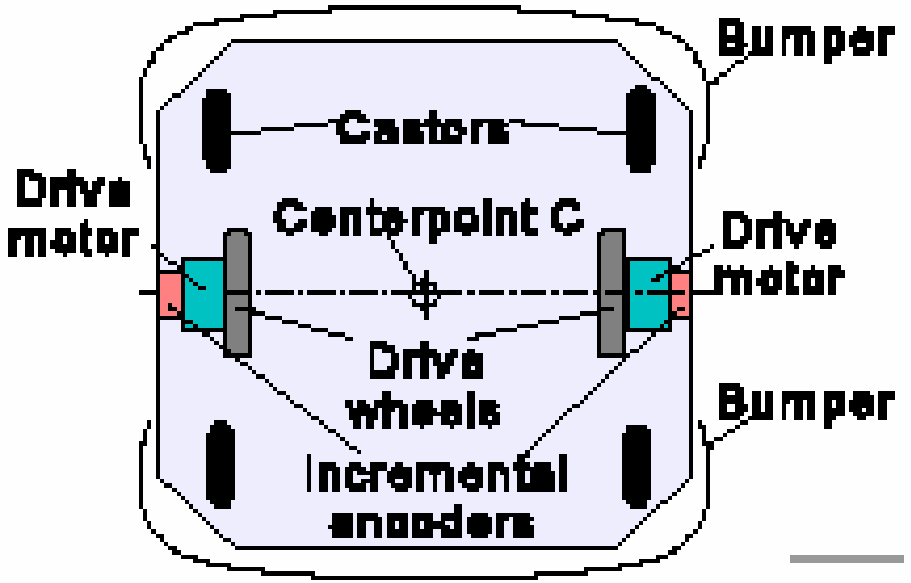
Odometry Examples





- Good line, poor line, squares

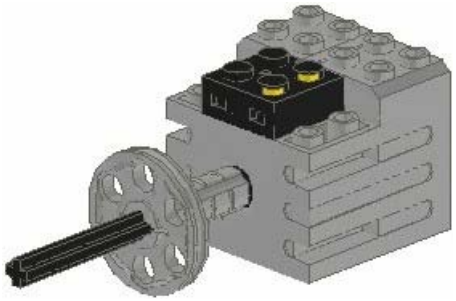
Differential Drive



(From Borenstein et. al.)

- Two motors and driven wheels
- Robot pivots around center point
- Casters support weight at edges
- Odometry tips:
  - Larger wheelbases are less prone to orientation errors
  - Castor wheels that bear significant weight induce slippage
  - Limit speed during turning to reduce slippage
  - Limit accelerations

Calculating Kinematics



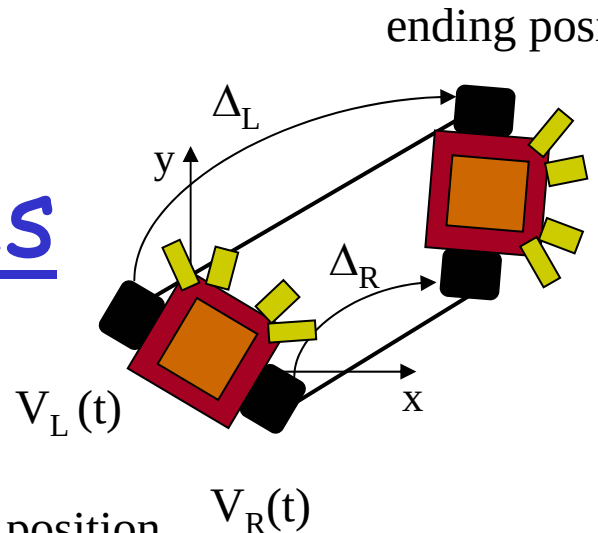
- Assume encoders mounted on drive motors
- Let
  - Cm = encoder count to linear displacement conversion factor
  - Dn = wheel diameter
  - Ce = encoder pulses per revolution
  - N = gear ratio
- $Cm = \pi Dn / N Ce$
- Incremental travel distance for left wheel

  $\Delta L = Cm\ NL$ (NL = encoder counts on left wheel)
- Incremental travel distance for right wheel

  $\Delta R = Cm\ NR$ (NR = encoder counts on right wheel)
- That's all we need for determining horizontal displacement and rotation from encoder counts

16 counts per revolution

Differential Drive Odometry/Kinematics



- ΔL = distance traveled by left wheel
- ΔR = distance traveled by right wheel

Distance D traveled by center point of robot is

$D = (\Delta R + \Delta L)/2$

Change in orientation $\Delta\theta$ is

$\Delta\theta = (\Delta R - \Delta L)/base$

- Given: Starting pose (x,y,θ), D and $\Delta\theta$
- Compute: Ending pose (x',y',θ')

New orientation is now

$\theta' \approx \theta + \Delta\theta$

New position is now

$x' \approx x + D \cos\theta'$

$y' \approx y + D \sin\theta'$

Inverse Kinematics for Waypoint Navigation

- Given (sequence of):
  - Initial pose (x, y, theta)
  - Target pose (x', y', theta')
- Decompose movement:
  1. Rotate robot to face toward (x',y')
  2. Move in straight line to (x',y')
  3. Rotate robot to orientation theta'
- Determine target NR and NL (right and left encoder counts), for each movement:
  - Rotation:
    1. Determine desired angle
    2. Determine encoder counts
  - Straight line movement
    1. Determine desired distance
    2. Determine encoder counts

Closed-Loop Control for Orientation Adjustment

- Given target right and left encoder counts (from inverse kinematics)
- Program: guide robot toward target counts
  - Begin movement with initial motor speeds
  - Monitor feedback from encoders
  - Adjust motor speeds in small increments so that encoder ratios are maintained and the robot achieves the encoder targets with minimal delay

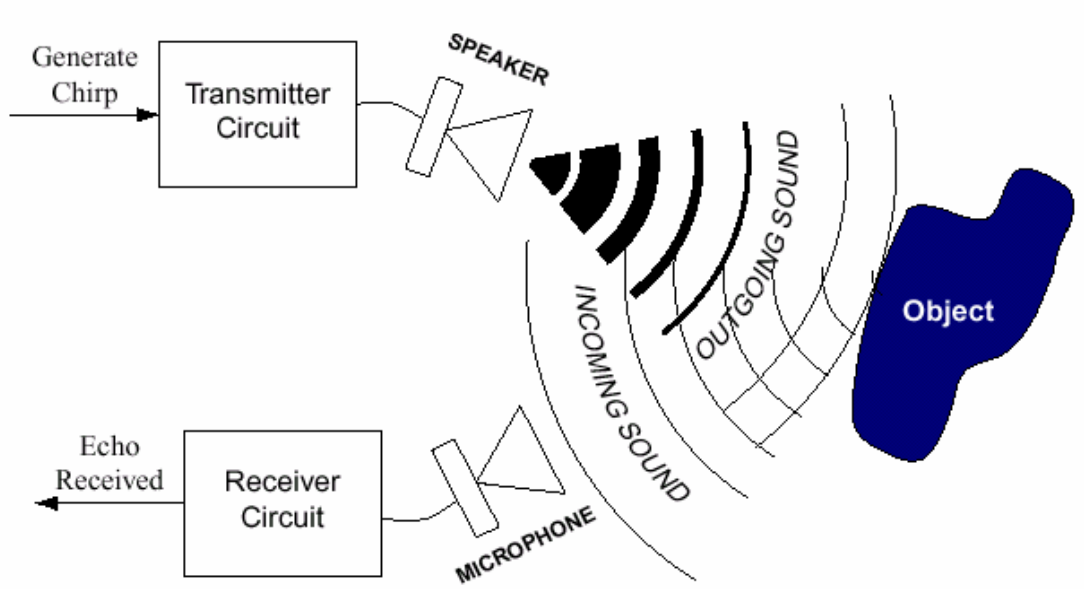
- Loop (while targets not reached)
  - Compute current ratio and set state =
    - Below_target_ratio or above_target_ratio or at_target_ratio (note: target ratio is 1:1 if you are always going straight)
  - Adjust motor velocities to stay in at_target_ratio state (could use proportional derivative control here)

Teaching Probabilistic Localization

- Instructions
- Background material
  - Review odometry
  - Review sonar
  - Particle filtering for probabilistic localization
- Implementation details and tips

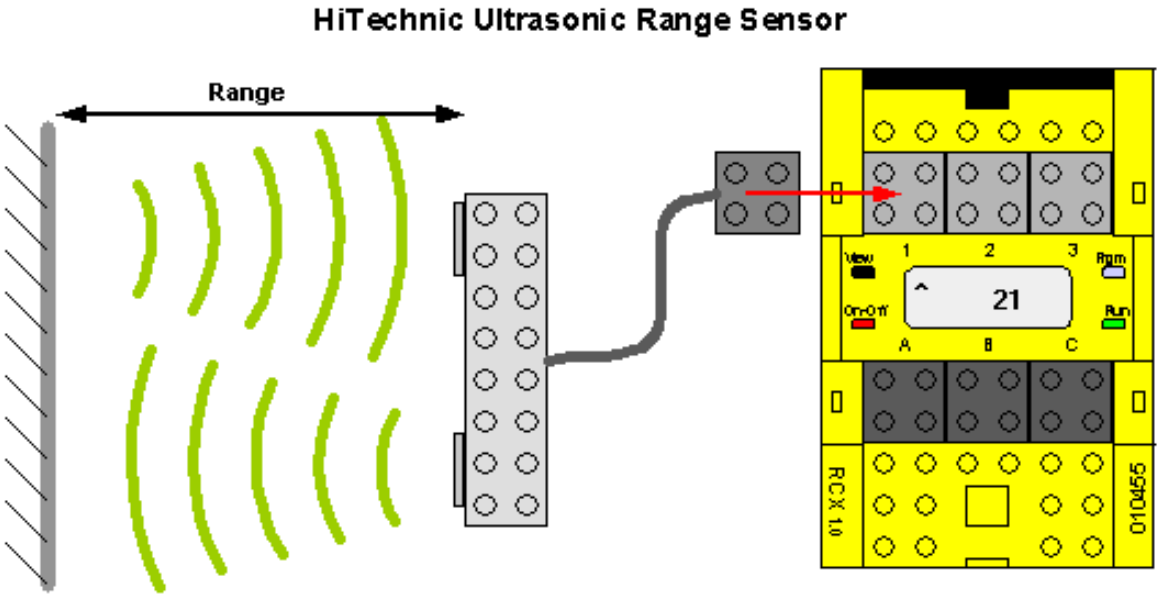
Ultrasonic Distance Sensing (SONAR)

- Ultrasonic burst, or "chirp,"
  - travels out to an object
  - reflected back into a receiver circuit (tuned to detect the specific frequency of sound)
- Measures time-of-flight of "chirp"
- Sound travels about 0.89 ms per foot (1.12 feet per ms) -- 1.78 ms for round trip
- Distance to the target object (in feet) is round trip time (ms) divided by 1.78
- Greater accuracy than with IR
- Bats use form of ultrasonic ranging to navigate



Hitechnic Ultrasonic Sensor

- 40kHz sound bursts
- On-board circuit to time echo and return calculated range to RCX
- Range: 6in-56in in ½ inch units (0-100 as light sensor on RCX)



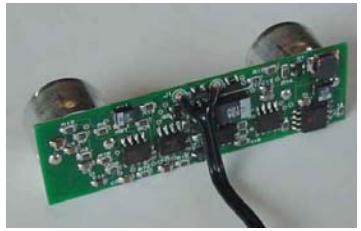
Attach the Ultrasonic Range Sensor as shown. The sensor port should be set to light sensor mode. With the Lego firmware loaded, use the View button to select the port and observe the value displayed. The distance to the target surface is the range as follows;

Range = (Reading / 2) + 6 inches

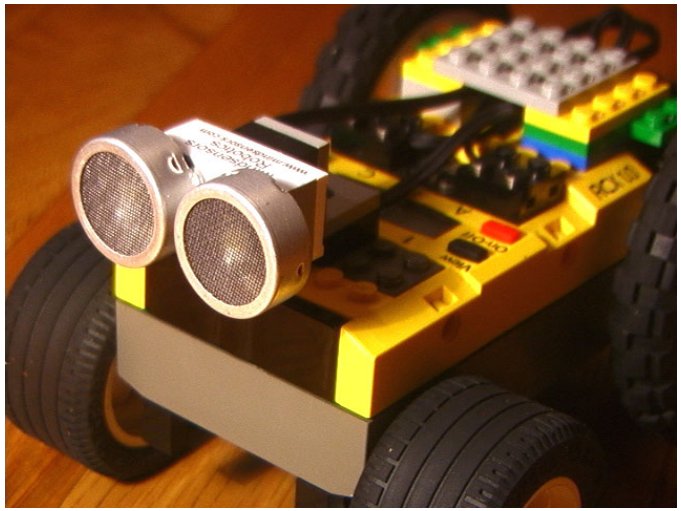
In the example shown the range is 16.5 inches (21 /2 + 6). The sensor actually measures in half inch increments beyond the first six inches.

If you have any questions, email us at info@hitechnic.com or visit www.hitechnic.com

(from hitechnic.com)



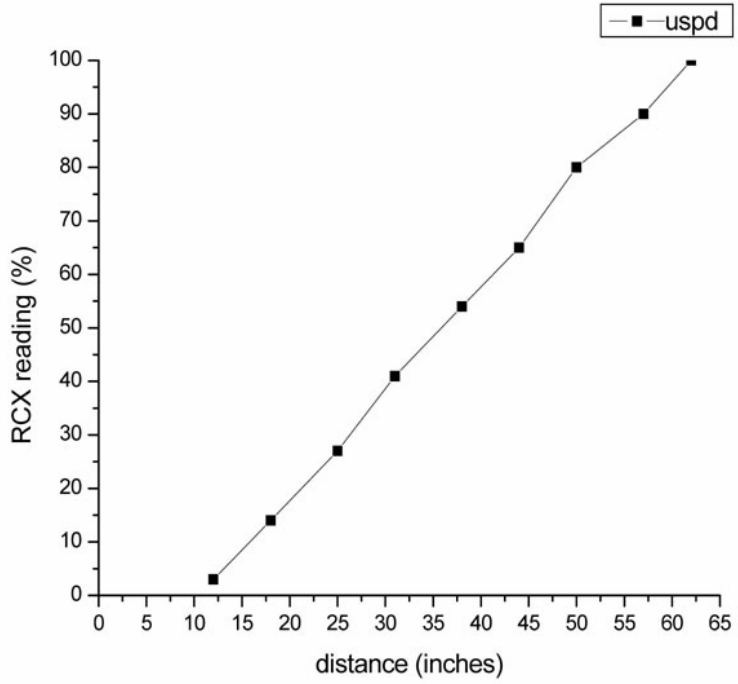
Mindsensors Sonar Details



- Freq: 24kHz
- Range: 30 cm to 1.5meters
- Accuracy: 4-5 cm
- Pinging and timing inside sonar package
- Programming:

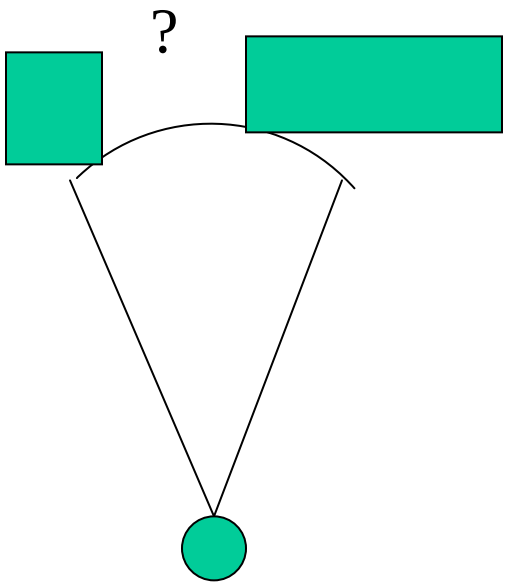
```
int a;
Sensor.S2.setTypeAndMode(3,0x80);
Sensor.S2.activate();
for(;;) {
        a=Sensor.S2.readValue();
        LCD.showNumber(a);
        // add sleep here
}
```

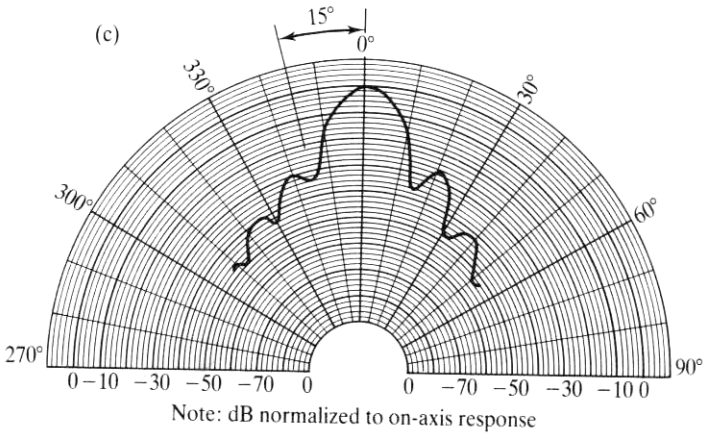
a is number from 1 to 100
Empirical mapping = (12+.53*a) inches
dist = inchesToCm((float)(12f+.53*(float)a))



(from http://www.mindsensors.com/uspd.htm)

Sonar Beam Pattern

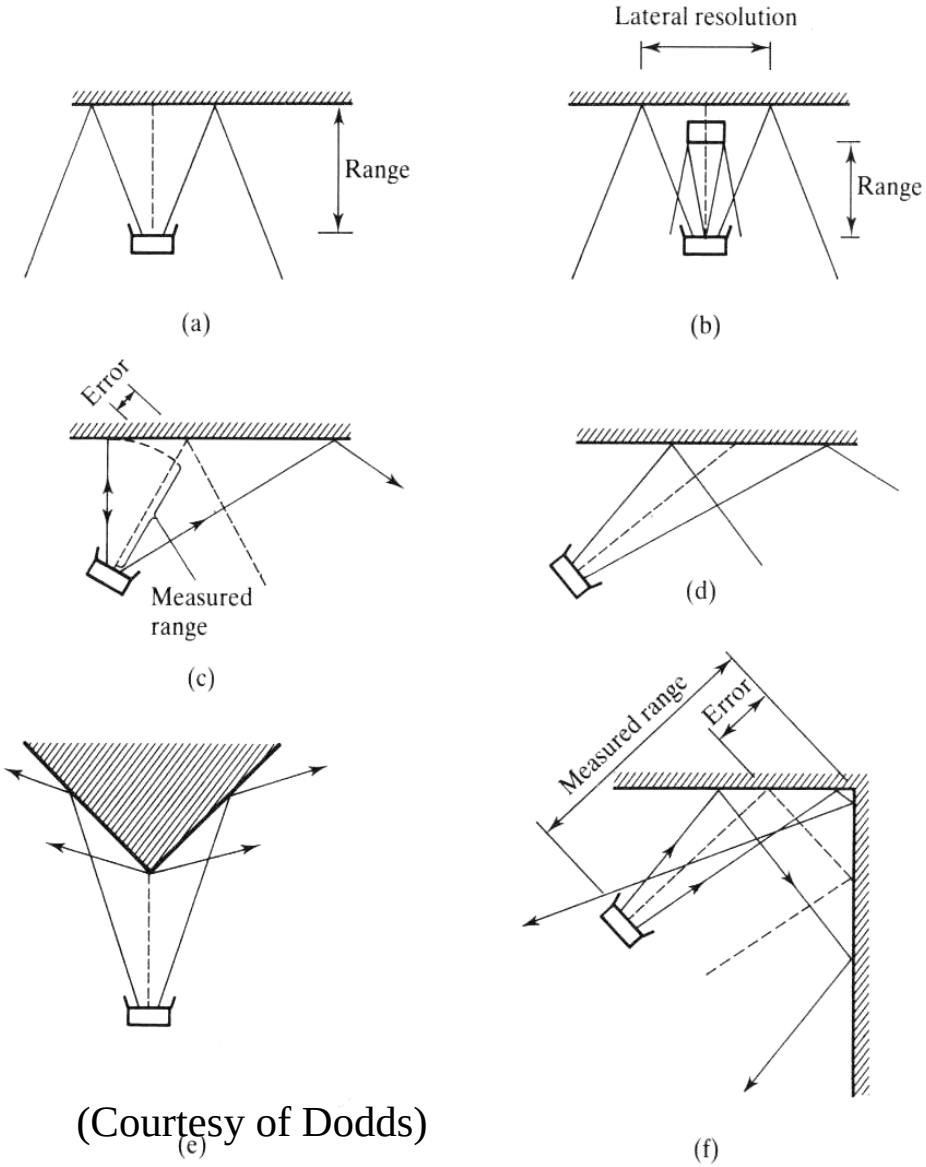




(Courtesy of Dodds)

- Distance is not a point distance
- Sonar beam has angular "spread" (about 30 degree dispersion)
- Closest point of object is somewhere within that arc
- Need multiple readings to disambiguate – but readings take time

Sonar Effects



(Courtesy of Dodds)

(a) Sonar providing an accurate range measurement

(b-c) Lateral resolution is not very precise; the closest object in the beam's cone provides the response

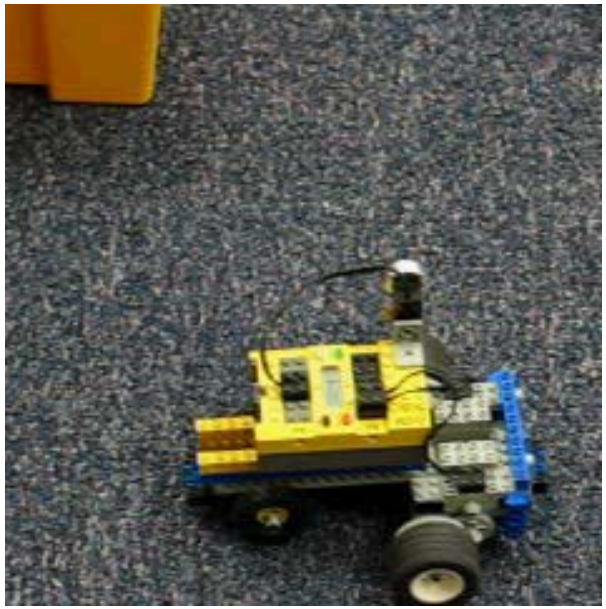
(d) Specular reflections cause walls to disappear

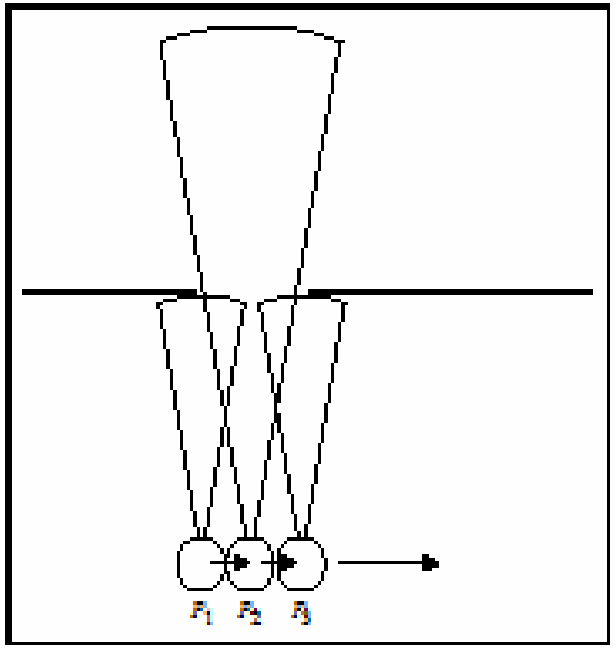
(e) Open corners produce a weak spherical wavefront

(f) Closed corners measure to the corner itself because of multiple reflections

Recognize Doors with Sonar

- Side-facing sonar
  - No need for pivoting or multiple sonar
- Learn door probability model





(from Thrun 2002)

Teaching Probabilistic Localization

- Instructions
- Background material
  - Review odometry
  - Review sonar
  - Particle filtering for probabilistic localization
- Implementation details and tips

Probabilistic Localization with Particle Filtering

- Given:
  - Initial pose (optional)
  - Map of environment features/landmarks
  - Sequence of movement actions, over time
  - Sequence of sensor readings, over time
  - Model of movement uncertainty
  - Model of sensor reading-landmark associations
- Compute:
  - Probability distribution over possible current poses (belief state)

Implement Particle Filtering

- Update location distribution incrementally
- Inputs: movement actions, encoder feedback (optional), and sonar signal

1. **Inputs:** $S_{t-1} = \{\langle x_{t-1}^{(i)}, w_{t-1}^{(i)} \rangle \mid i = 1, \ldots, n\}$ representing belief $Bel(x_{t-1})$, control measurement u_{t-1}, observation z_t
2. $S_t := \emptyset$, $\alpha := 0$ *// Initialize*
3. **for** $i := 1, \ldots, n$ **do** *// Generate n samples*

   // Resampling: Draw state from previous belief
4. Sample an index j from the discrete distribution given by the weights in S_{t-1}

   // Sampling: Predict next state
5. Sample $x_t^{(i)}$ from $p(x_t \mid x_{t-1}, u_{t-1})$ conditioned on $x_{t-1}^{(j)}$ and u_{t-1}
6. $w_t^{(i)} := p(z_t \mid x_t^{(i)})$; *// Compute importance weight*
7. $\alpha := \alpha + w_t^{(i)}$ *// Update normalization factor*
8. $S_t := S_t \cup \{\langle x_t^{(i)}, w_t^{(i)} \rangle\}$ *// Insert sample into sample set*
9. **for** $i := 1, \ldots, n$ **do** *// Normalize importance weights*
10. $w_t^{(i)} := w_t^{(i)} / \alpha$
11. **return** S_t

(from Fox 2003)

General Problem: Tracking Change over Time

The world changes; we need to track and predict it

Diabetes management vs vehicle diagnosis

Basic idea: copy state and evidence variables for each time step

$\mathbf{X}_t$ = set of unobservable state variables at time t
e.g., $BloodSugar_t$, $StomachContents_t$, etc.

$\mathbf{E}_t$ = set of observable evidence variables at time t
e.g., $MeasuredBloodSugar_t$, $PulseRate_t$, $FoodEaten_t$

This assumes **discrete time**; step size depends on problem

Notation: $\mathbf{X}_{a:b} = \mathbf{X}_a, \mathbf{X}_{a+1}, \ldots, \mathbf{X}_{b-1}, \mathbf{X}_b$

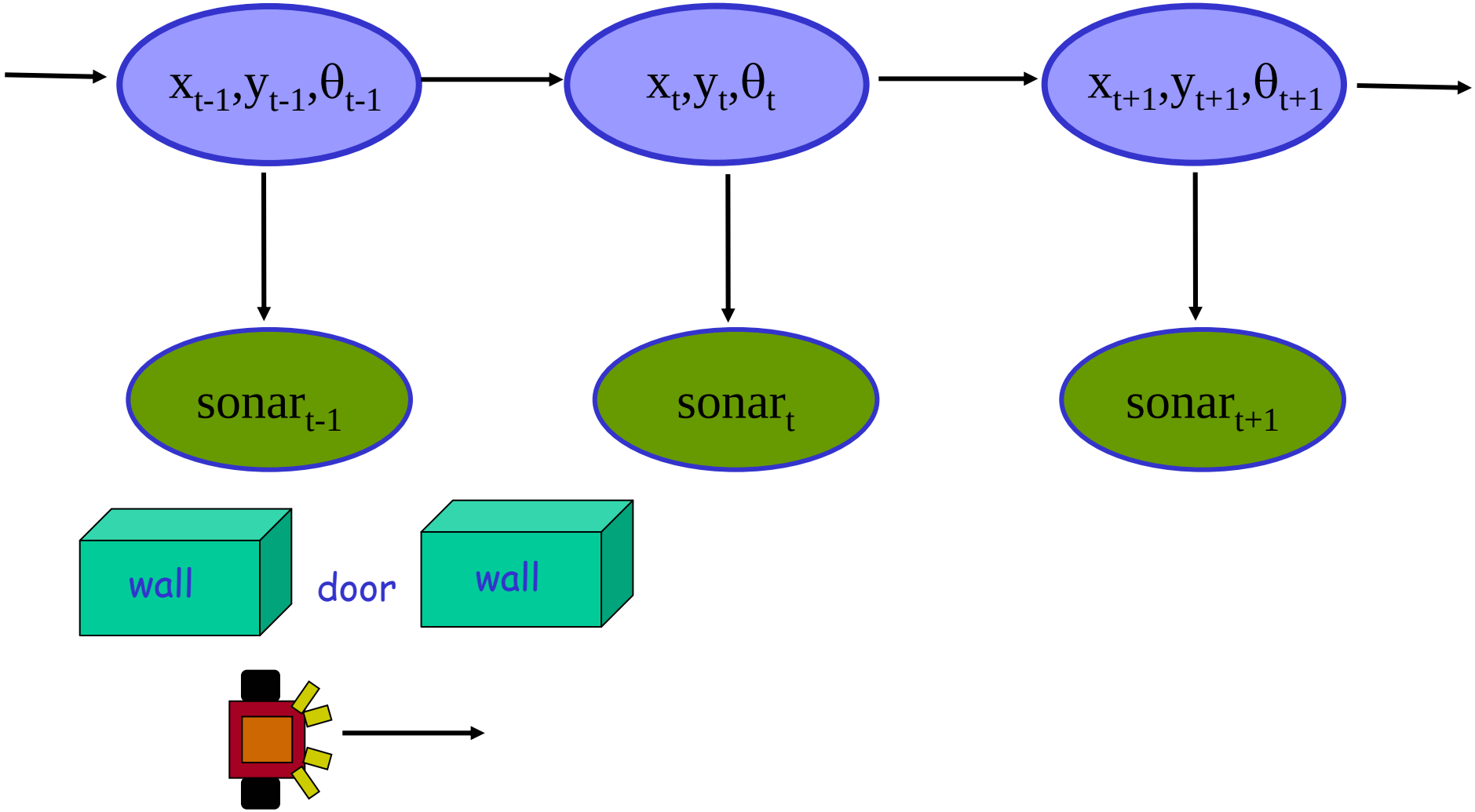
In localization

unobservable state is pose (x, y, θ)

sensor readings and movement actions are evidence

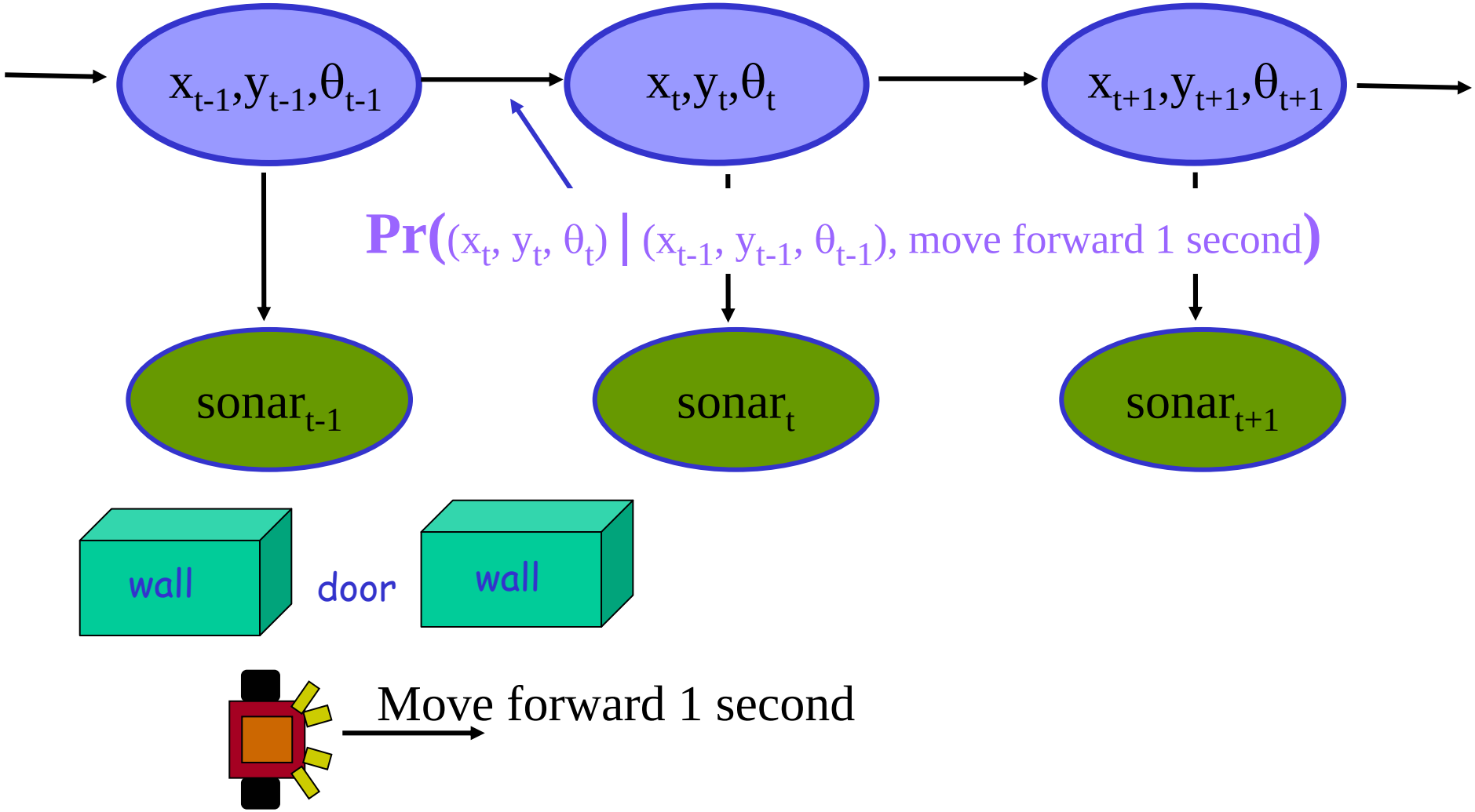
(from Russell and Norvig)

Localization over Time

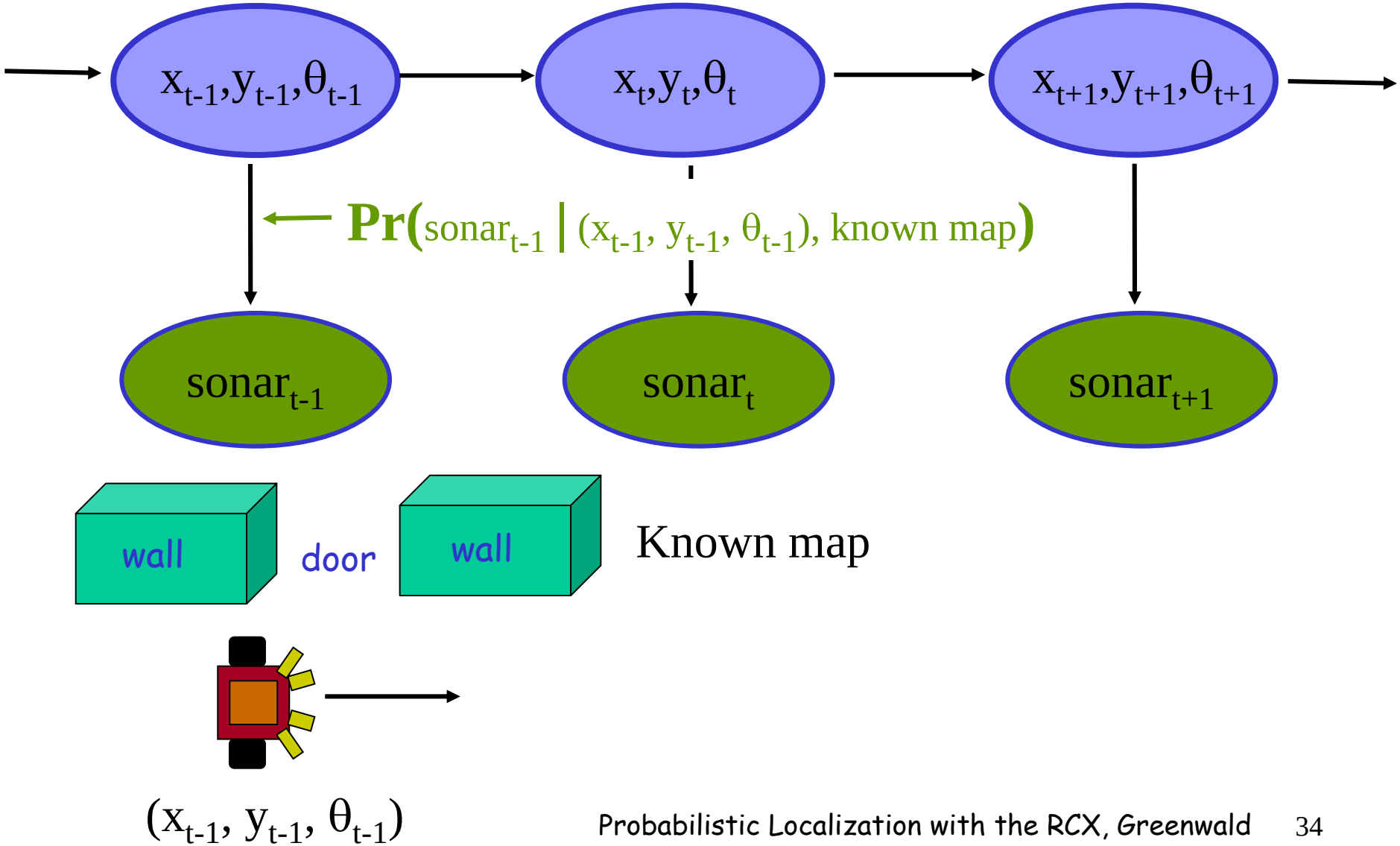


$(x_{t-1}, y_{t-1}, \theta_{t-1})$

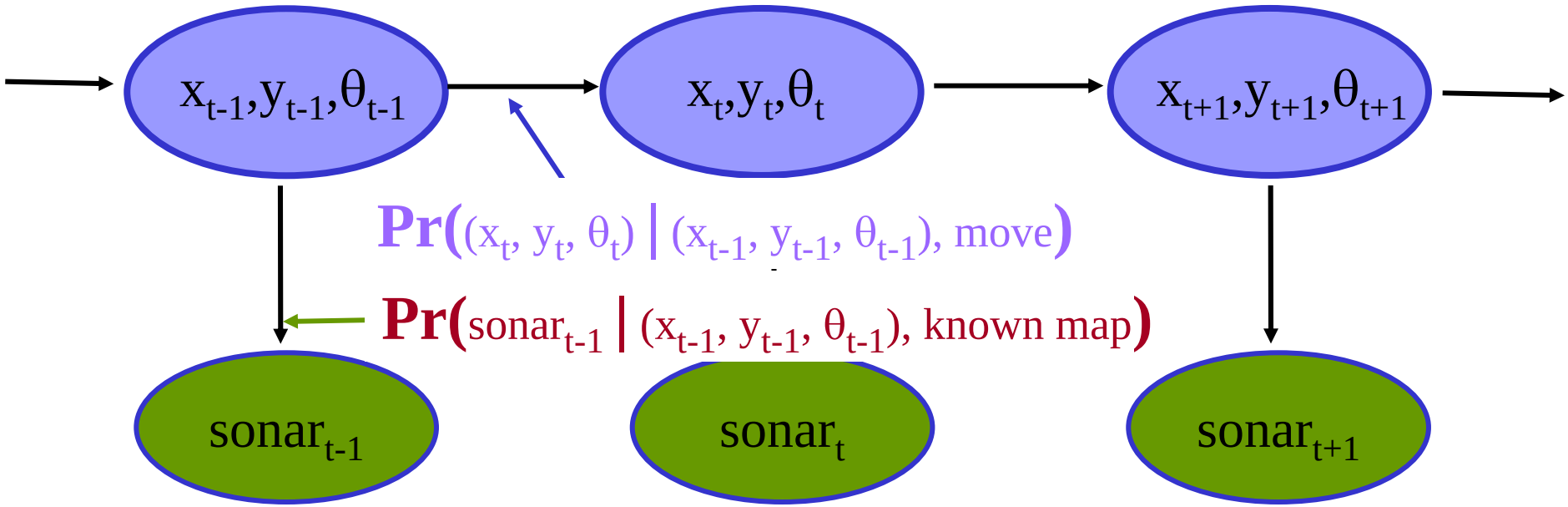
Movement Model



Sensor/Landmark Model



Dynamic Bayesian Network/Hidden Markov Model for Localization



Filtering: Compute current belief state given history of observations/actions

Filtering: Compute Current Belief State

Aim: devise a **recursive** state estimation algorithm:

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t}))$$

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}, \mathbf{e}_{1:t})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})\end{aligned}$$

Dividing the evidence

Bayes' rule $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

Markov property

Update with new evidence (sensor model)

I.e., prediction + estimation. Prediction by summing out $\mathbf{X}_t$:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\Sigma_{\mathbf{x}_t}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t, \mathbf{e}_{1:t})P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\Sigma_{\mathbf{x}_t}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t})\end{aligned}$$

Markov property

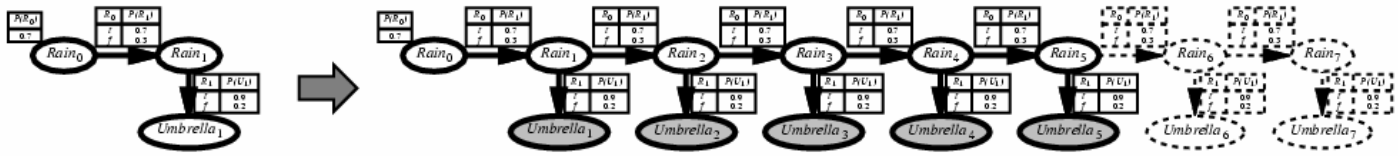
Transition model

Current belief state

Dynamic Bayesian networks ARE Bayesian networks

Exact inference in DBNs

Naive method: unroll the network and run any exact algorithm



Problem: inference cost for each update grows with t

Rollup filtering: add slice $t+1$, "sum out" slice t using variable elimination

with variables in temporal order

Largest factor is $O(d^{n+1})$, update cost $O(d^{n+2})$

This is not good news.
Variable elimination is not efficient for inference because the factors grow to include all state variables

Chapter 15 35

(from Russell and Norvig)

Likelihood weighting

Idea: fix evidence variables, sample only nonevidence variables, and weight each sample by the likelihood it accords the evidence

function LIKELIHOOD-WEIGHTING($X, \mathbf{e}, bn, N$) **returns** an estimate of $P(X|\mathbf{e})$
  local variables: $\mathbf{W}$, a vector of weighted counts over X, initially zero

  for $j = 1$ to N **do**
    $\mathbf{x}, w \leftarrow$ WEIGHTED-SAMPLE(bn)
    $\mathbf{W}[x] \leftarrow \mathbf{W}[x] + w$ where x is the value of X in $\mathbf{x}$
  return NORMALIZE($\mathbf{W}[X]$)

---

function WEIGHTED-SAMPLE($bn, \mathbf{e}$) **returns** an event and a weight

  $\mathbf{x} \leftarrow$ an event with n elements; $w \leftarrow 1$
  for $i = 1$ **to** n **do**
    if X_i has a value x_i in $\mathbf{e}$
      then $w \leftarrow w \times P(X_i = x_i \mid Parents(X_i))$
      else $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid Parents(X_i))$
  return $\mathbf{x}$, w

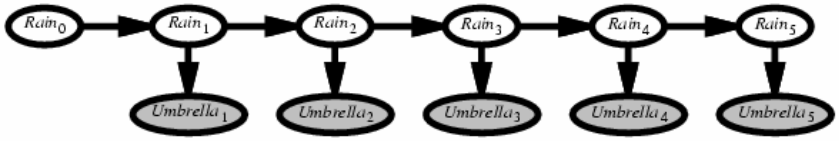
Weighted by conditional probabilities

(from Russell and Norvig)

Resort to approximate inference

Likelihood weighting for DBNs

Set of weighted samples approximates the belief state

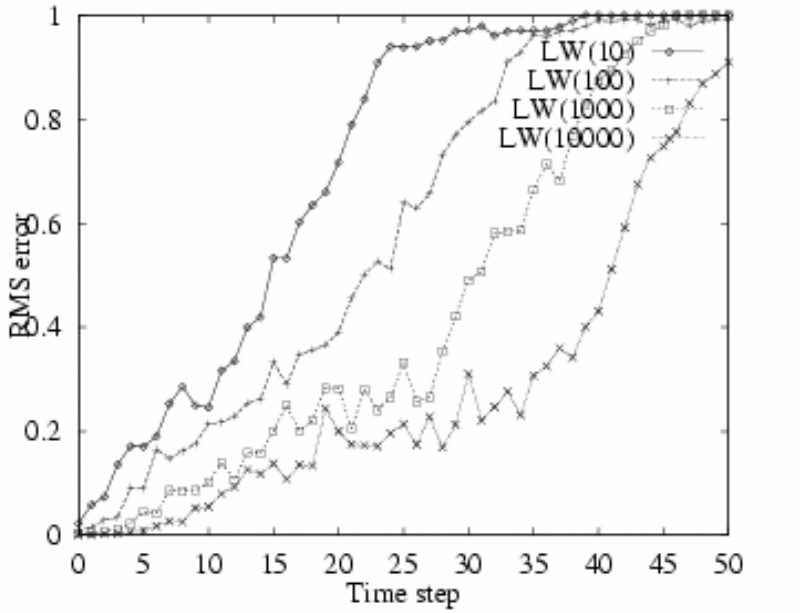


LW samples pay no attention to the evidence!

$\Rightarrow$ fraction "agreeing" falls exponentially with t

$\Rightarrow$ number of samples required grows exponentially with t

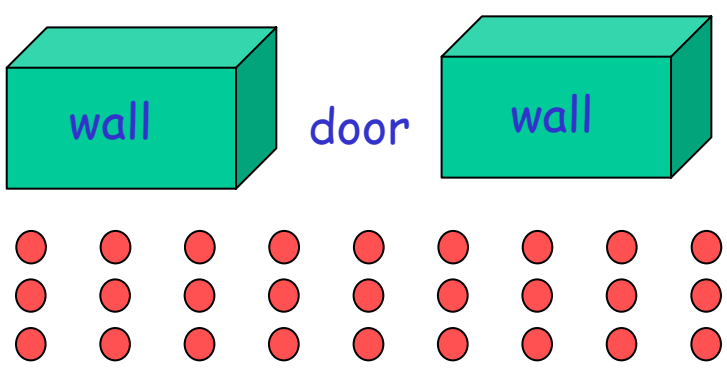
- Samples generated "upstream" from evidence
- Weight of sample depends on evidence but, actual samples have no relation to evidence



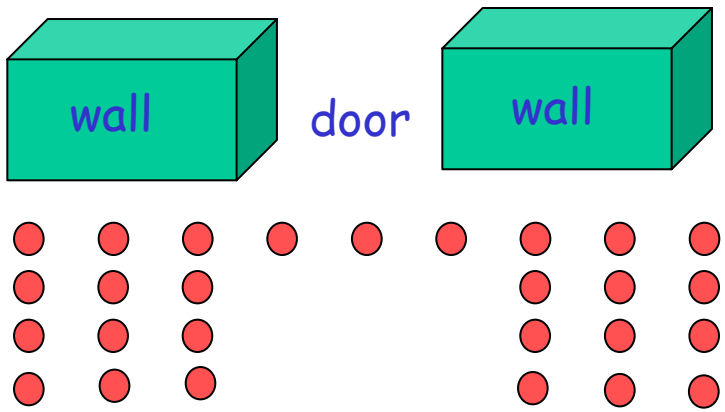
(from Russell and Norvig)

Particle Filtering

- Given: a constant set of "particles" (samples)
- Distribute: particles over possible states at time t
- Re-distribute: particles given new evidence to track belief state – where is the robot?



Could be anywhere

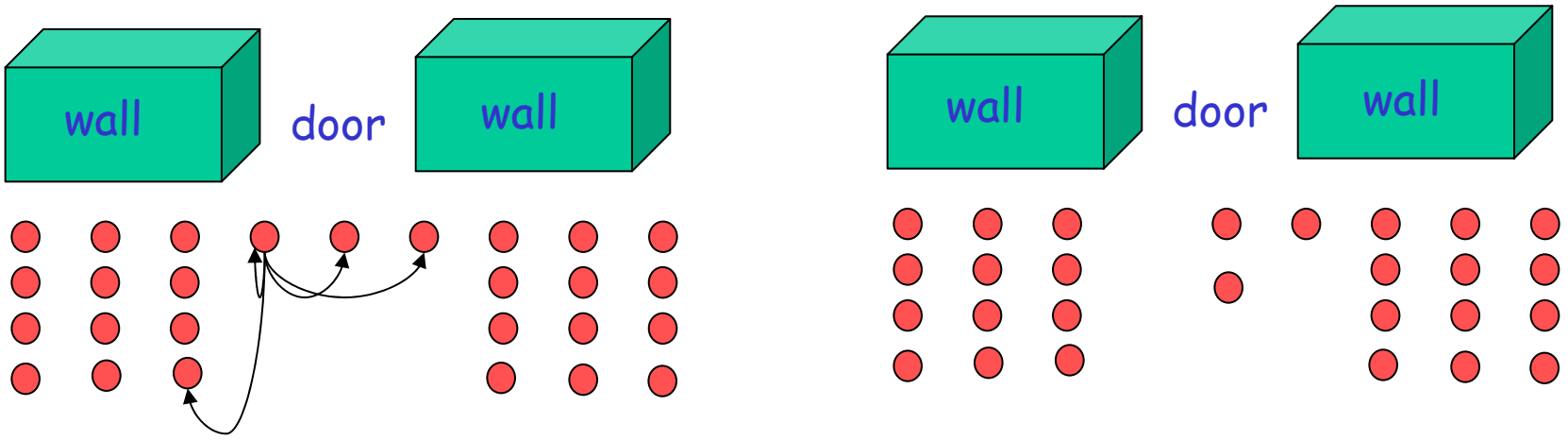


Just saw a wall (maybe)

Particle Filtering Steps

1. Propagate particles (samples) according to movement model
2. Weigh samples according to sensor model (evidence)
3. Resample according to weights

Result: track high probability states and throw away particles with very low weights given evidence

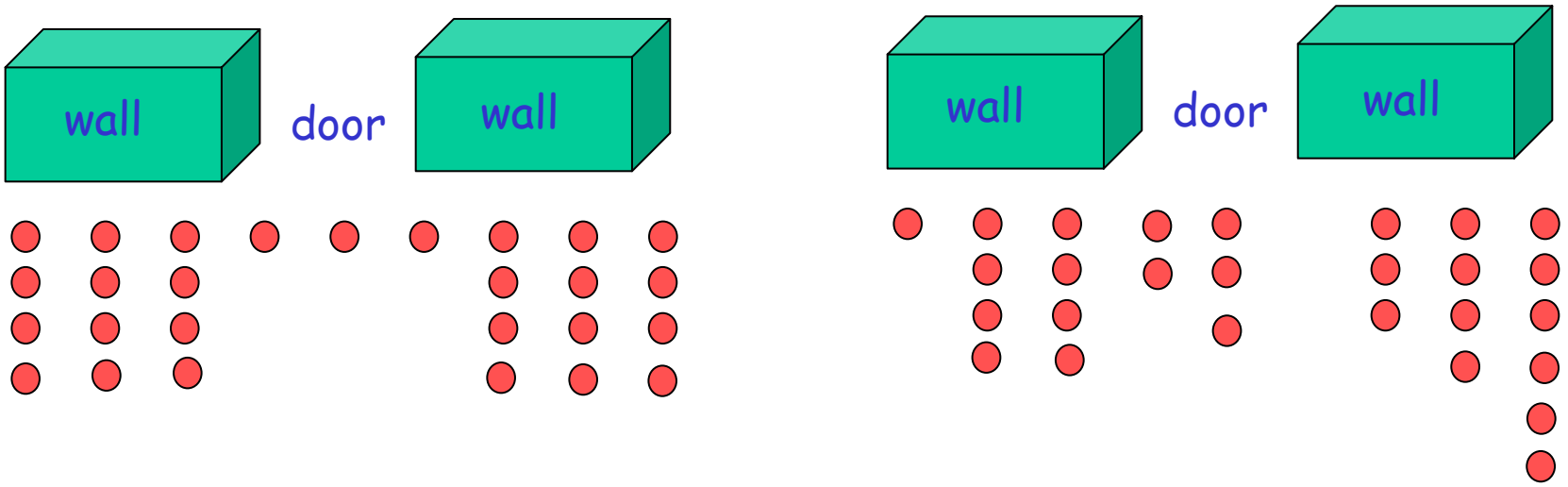


$$\mathbf{Pr}((x_t, y_t, \theta_t) \mid (x_{t-1}, y_{t-1}, \theta_{t-1}), \text{move forward 1 second})$$

Particle Filtering Steps

1. Propagate particles (samples) according to movement model
2. Weigh samples according to sensor model (evidence)
3. Resample according to weights

Result: track high probability states and throw away particles with very low weights given evidence

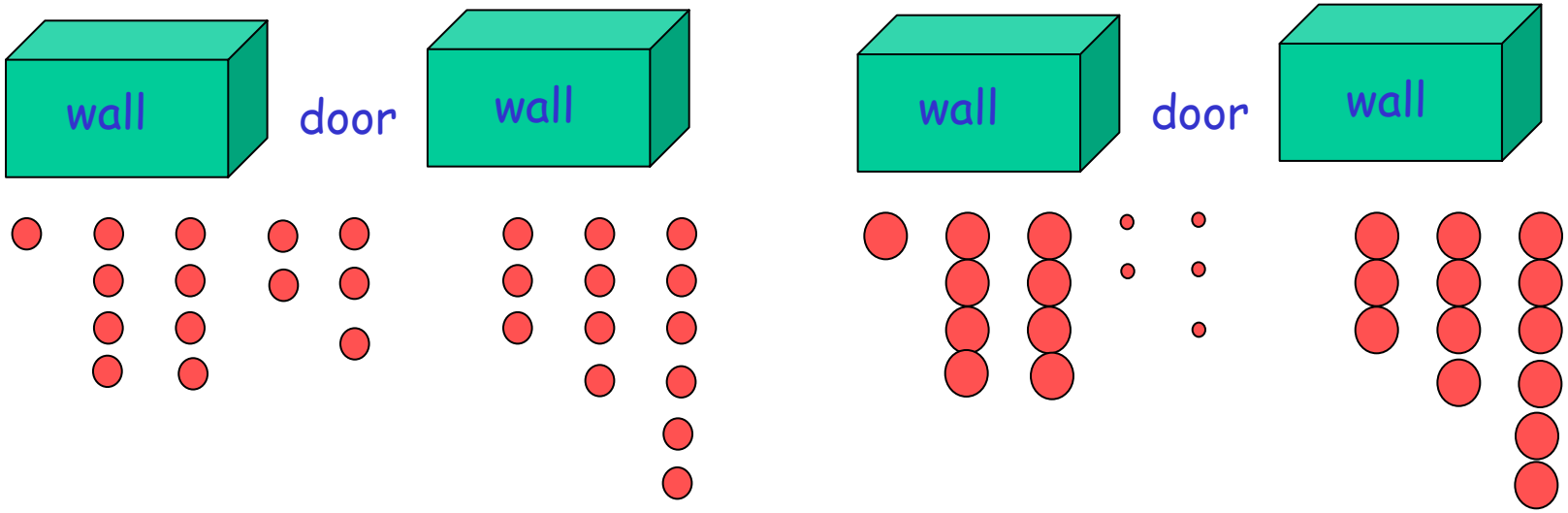


$$\mathbf{Pr}((x_t, y_t, \theta_t) \mid (x_{t-1}, y_{t-1}, \theta_{t-1}), \text{move forward 1 second})$$

Particle Filtering Steps

1. Propagate particles (samples) according to movement model
2. Weigh samples according to sensor model (evidence)
3. Resample according to weights

Result: track high probability states and throw away particles with very low weights given evidence

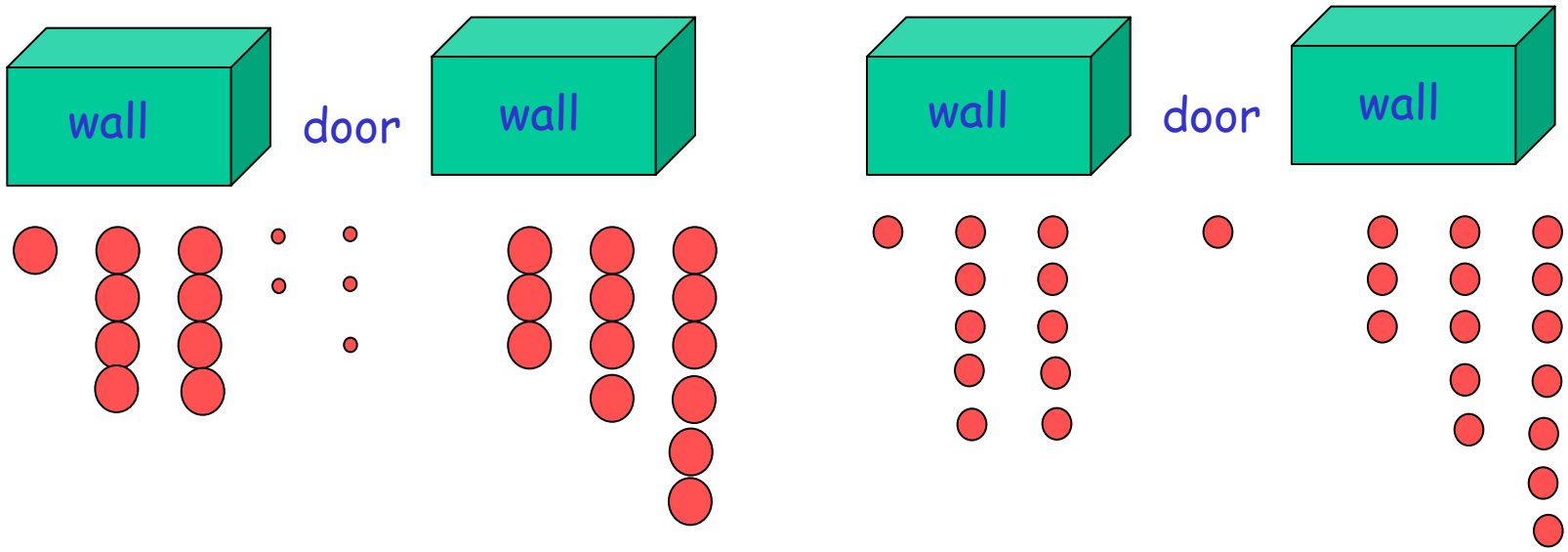


$$\mathbf{Pr}(\text{sonar}_{t-1} = \text{wall} \mid (x_{t-1}, y_{t-1}, \theta_{t-1}), \text{known map})$$

Particle Filtering Steps

1. Propagate particles (samples) according to movement model
2. Weigh samples according to sensor model (evidence)
3. Resample according to weights (weighted random selection)

Result: track high probability states and throw away particles with very low weights given evidence



Particle Filtering Summary

1. Propagate particles (samples) according to movement model
2. Weigh samples according to sensor model (evidence)
3. Resample according to weights (weighted random selection)

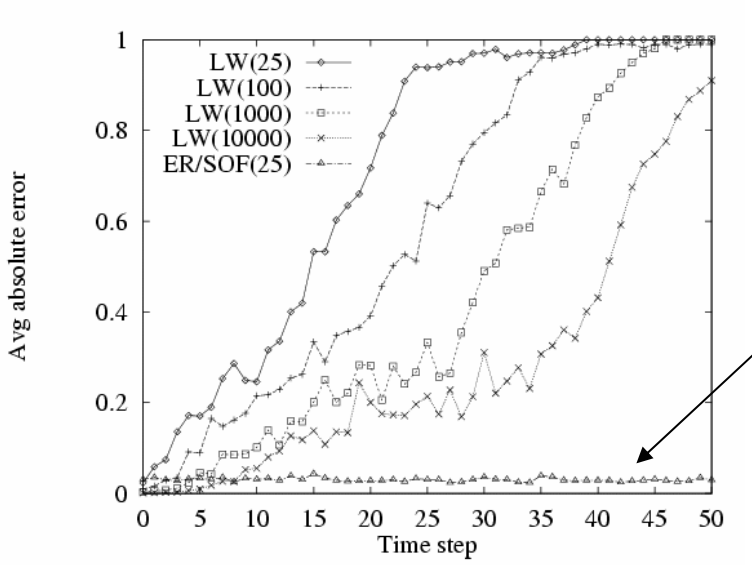
Result: track high probability states and throw away particles with very low weights given evidence

Widely used for tracking nonlinear systems

Can handle high-dimensional state spaces

Simple to implement

Consistent approximations with bounded computation

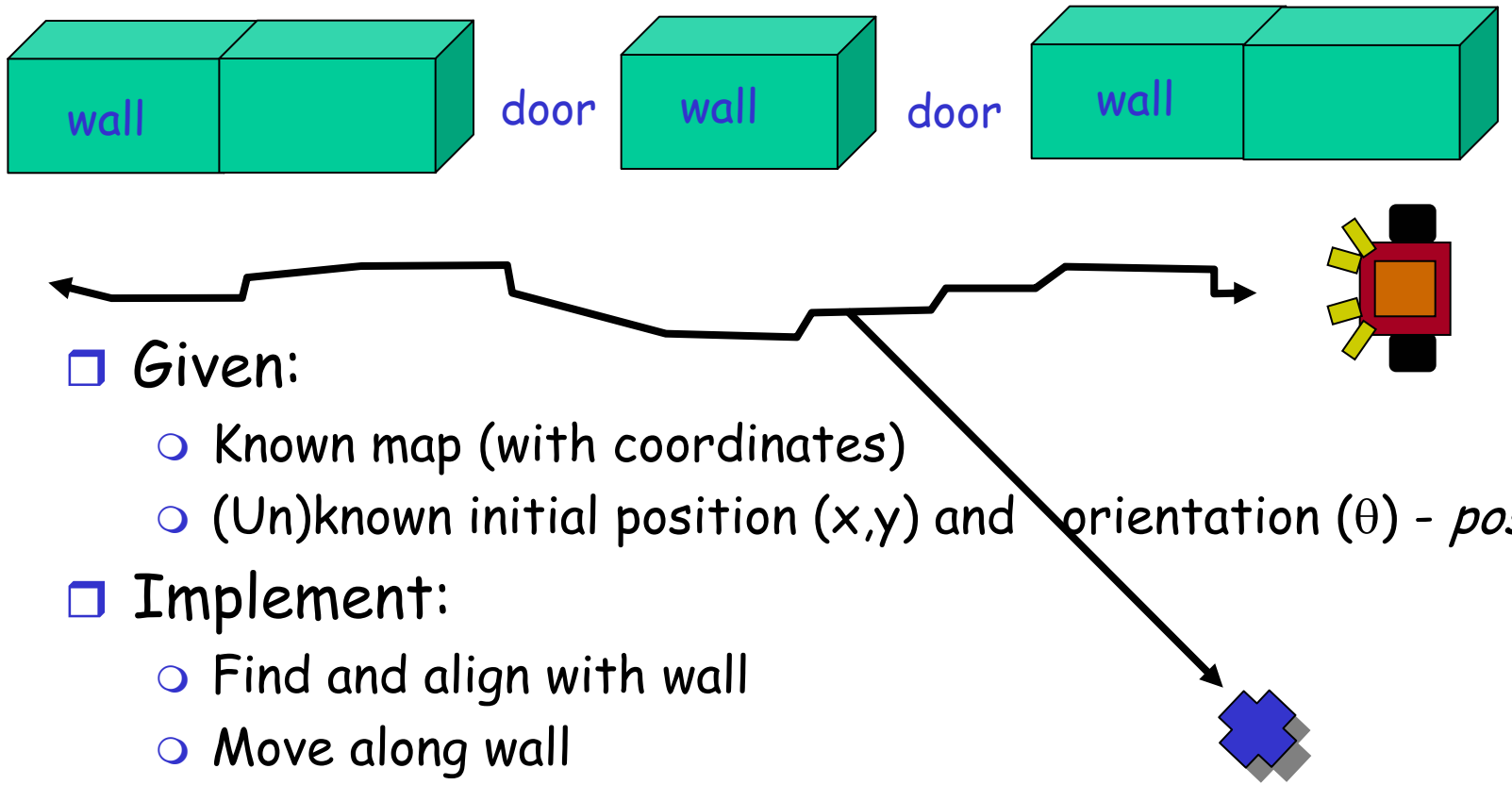


(from Russell and Norvig)

Teaching Probabilistic Localization

- Instructions
- Background material
- Implementation details and tips

The Challenge: Simplified Localization



- Given:
  - Known map (with coordinates)
  - (Un)known initial position (x,y) and orientation (θ) - *pose*
- Implement:
  - Find and align with wall
  - Move along wall
  - Use sensors to determine location
  - Go to known goal point (x,y)

Implement Particle Filtering

- Update location distribution incrementally
- Inputs: movement actions, encoder feedback (optional), and sonar signal

1. **Inputs:** $S_{t-1} = \{\langle x_{t-1}^{(i)}, w_{t-1}^{(i)} \rangle \mid i = 1, \ldots, n\}$ representing belief $Bel(x_{t-1})$, control measurement u_{t-1}, observation z_t
2. $S_t := \emptyset$, $\alpha := 0$ *// Initialize*
3. **for** $i := 1, \ldots, n$ **do** *// Generate n samples*

   // Resampling: Draw state from previous belief
4. Sample an index j from the discrete distribution given by the weights in S_{t-1}

   // Sampling: Predict next state
5. Sample $x_t^{(i)}$ from $p(x_t \mid x_{t-1}, u_{t-1})$ conditioned on $x_{t-1}^{(j)}$ and u_{t-1}
6. $w_t^{(i)} := p(z_t \mid x_t^{(i)})$; *// Compute importance weight*
7. $\alpha := \alpha + w_t^{(i)}$ *// Update normalization factor*
8. $S_t := S_t \cup \{\langle x_t^{(i)}, w_t^{(i)} \rangle\}$ *// Insert sample into sample set*
9. **for** $i := 1, \ldots, n$ **do** *// Normalize importance weights*
10. $w_t^{(i)} := w_t^{(i)} / \alpha$
11. **return** S_t

(from Fox 2003)

Recall:

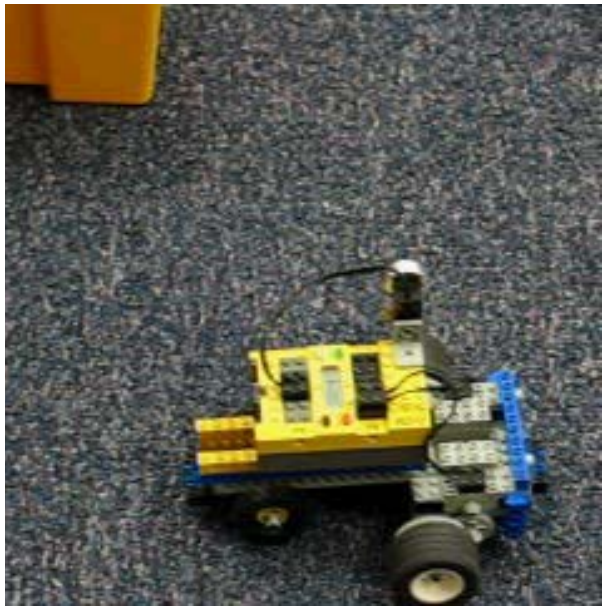
- Propagate particles (samples) according to movement model
- Weigh samples according to sensor model (evidence)
- Resample according to weights (weighted random selection)

Probabilistic Localization Implementation Details

1. Find and align with wall (optional)
2. Move along wall
3. Recognize ends of course
4. Recognize doors (sensor model – map location probability, side-facing sonar, parallel movement)
5. Calculate odometry estimates of movement (movement model - using encoders or timing)
6. Maintain orientation using closed-loop feedback, either sonar or encoders (wall following - optional – or use single drivetrain)
7. Use particle filtering to update probability distribution over locations
8. Continuously display most likely position (or display full distribution off-board)
9. Go to known goal point (x,y) (optional)

Movement

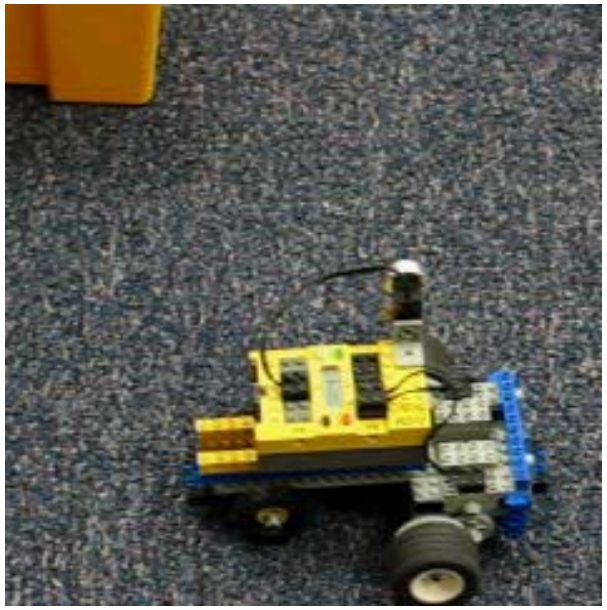
- Build two-wheel differential drive robot with rotation sensors
  - Simplification 1: one drive train
  - Simplification 2: no rotation sensors (i.e. timing-based movement)
- Implement odometry
  - LeJOS: built-in class RotationNavigator handles all calculations
  - Methods: backward, forward, getAngle, getX, getY, gotoAngle, gotoPoint, rotate, stop, travel
- Build movement model



$$\mathbf{Pr}((x_t, y_t, \theta_t) \mid (x_{t-1}, y_{t-1}, \theta_{t-1}), \text{move forward 1 second})$$

Movement

- Build two-wheel differential drive robot with rotation sensors
  - Simplification 1: one drive train
  - Simplification 2: no rotation sensors (i.e. timing-based movement)
- Implement odometry
  - LeJOS: built-in class RotationNavigator handles all calculations
  - Methods: backward, forward, getAngle, getX, getY, gotoAngle, gotoPoint, rotate, stop, travel
- Build movement model



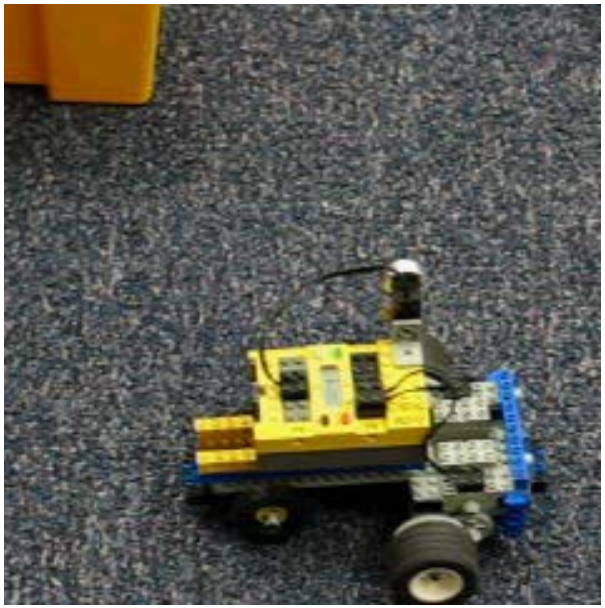
4. *// Resampling: Draw state from previous belief*
   Sample an index j from the discrete distribution given by the weights in S_{t-1}
   // Sampling: Predict next state
5. Sample $x_t^{(i)}$ from $p(x_t \mid x_{t-1}, u_{t-1})$ conditioned on $x_{t-1}^{(j)}$ and u_{t-1}
6. $w_t^{(i)} := p(z_t \mid x_t^{(i)})$; *// Compute importance weight*

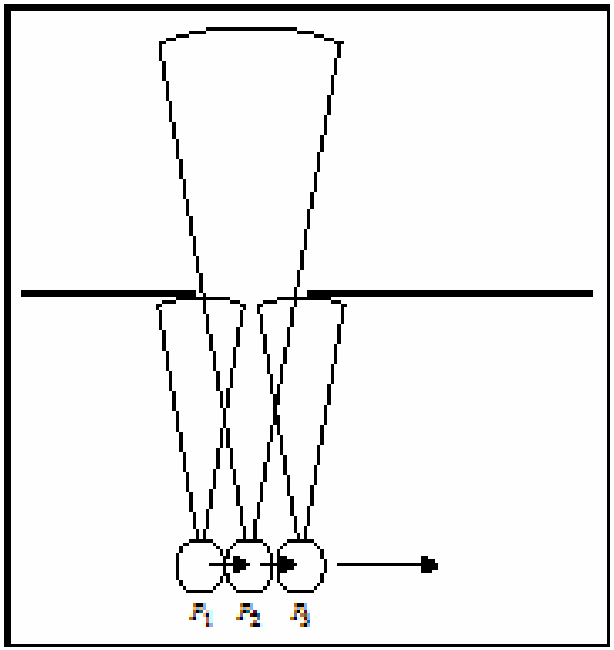
Sensing

- Side-facing sonar
  - No need for pivoting or multiple sonar
  - Sonar readings in LeJOS: setTypeAndMode (3, 0x80), activate, readValue
- Build map location probability model (sensor model)
  - Inputs: sonar reading, distance to wall (optional), orientation to wall (optional), known map

$$\mathbf{Pr}(\text{sonar}_{t-1} = \text{wall} \mid (x_{t-1}, y_{t-1}, \theta_{t-1}), \text{known map})$$

- Maintain parallel orientation and constant distance (optional)

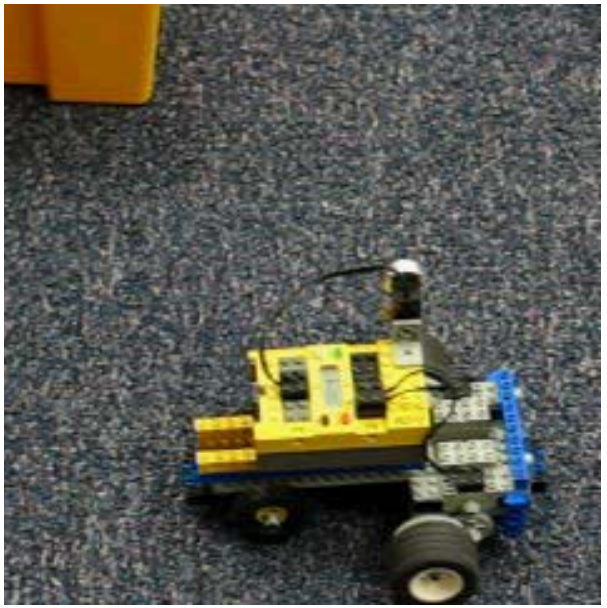


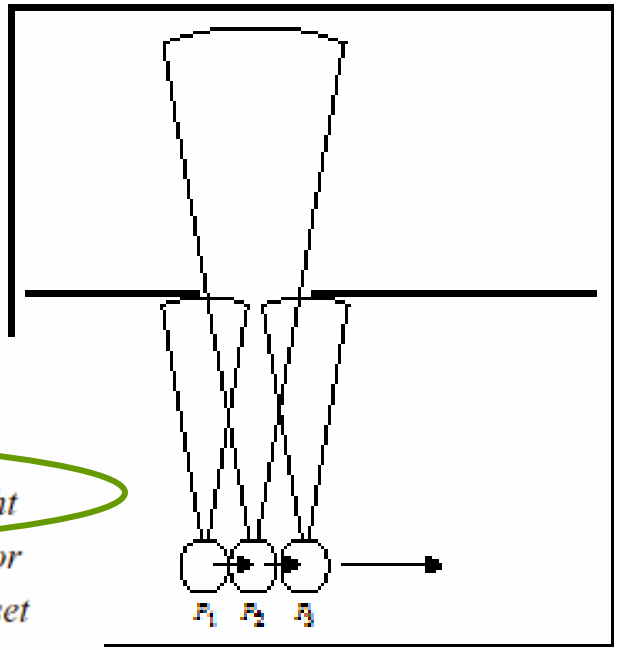


(from Thrun 2002)

Sensing

- Side-facing sonar
  - No need for pivoting or multiple sonar
  - Sonar readings in LeJOS: setTypeAndMode (3, 0x80), activate, readValue
- Build map location probability model (sensor model)
  - Inputs: sonar reading, distance to wall (optional), orientation to wall (optional), known map
- Maintain parallel orientation and constant distance (optional)



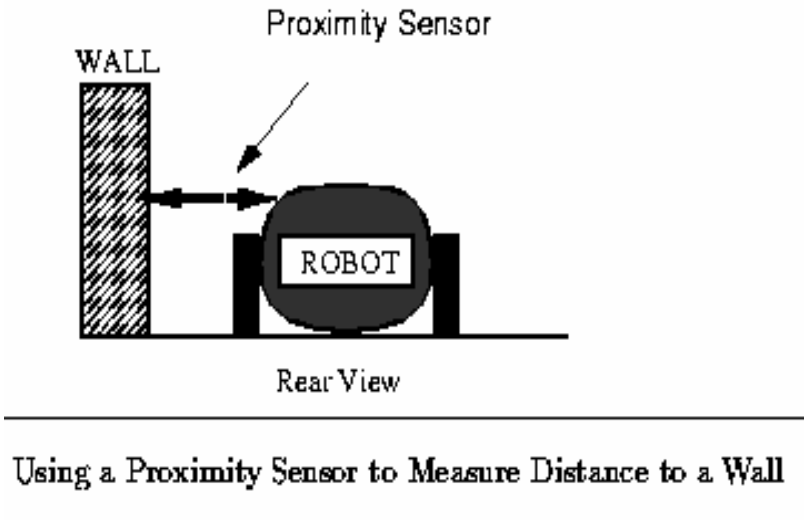


(from Thrun 2002)

// Sampling: Predict next state

5. Sample $x_t^{(i)}$ from $p(x_t \mid x_{t-1}, u_{t-1})$ conditioned on $x_{t-1}^{(j)}$ and u_{t-1}
6. $w_t^{(i)} := p(z_t \mid x_t^{(i)})$; *// Compute importance weight*
7. $\alpha := \alpha + w_t^{(i)}$ *// Update normalization factor*
8. $S_t := S_t \cup \{\langle x_t^{(i)}, w_t^{(i)} \rangle\}$ *// Insert sample into sample set*

Wall Following to Maintain Orientation and Distance



Using a Proximity Sensor to Measure Distance to a Wall

(Courtesy of Bennet)

- Drive parallel to wall
- Feedback from proximity sensors (e.g. bump, IR, sonar)
- Feedback loop, continuous monitoring and correction of motors -- adjusting distance to wall to maintain goal distance

Tips and Hints

- Sonar:
  - Sonar is a problem in general; maintaining parallel movement helps
  - RCX sonar very sensitive to non-perpendicular orientation
  - Useful range: 12-30inches

- Odometry:
  - LeJOS localization has significant error
  - Localization library does not indicate whether or not robot is moving
  - Distance traveled (as read) and distance commanded do not generally match
  - Some momentum effects
  - Using wheels as casters causes too much slippage
  - Rotation sensors lose counts if geared up to high rpms; very very slow gearing might also be a problem
  - Built-in odometry problems provide good pedagogical motivation for particle filtering
  - Rotation sensors cannot be multiplexed with other sensors

- General:
  - Need to sense ends of course – but don't use this extra information in algorithm
  - Wall-following to stay aligned with course helps

Tips and Hints (Continued)

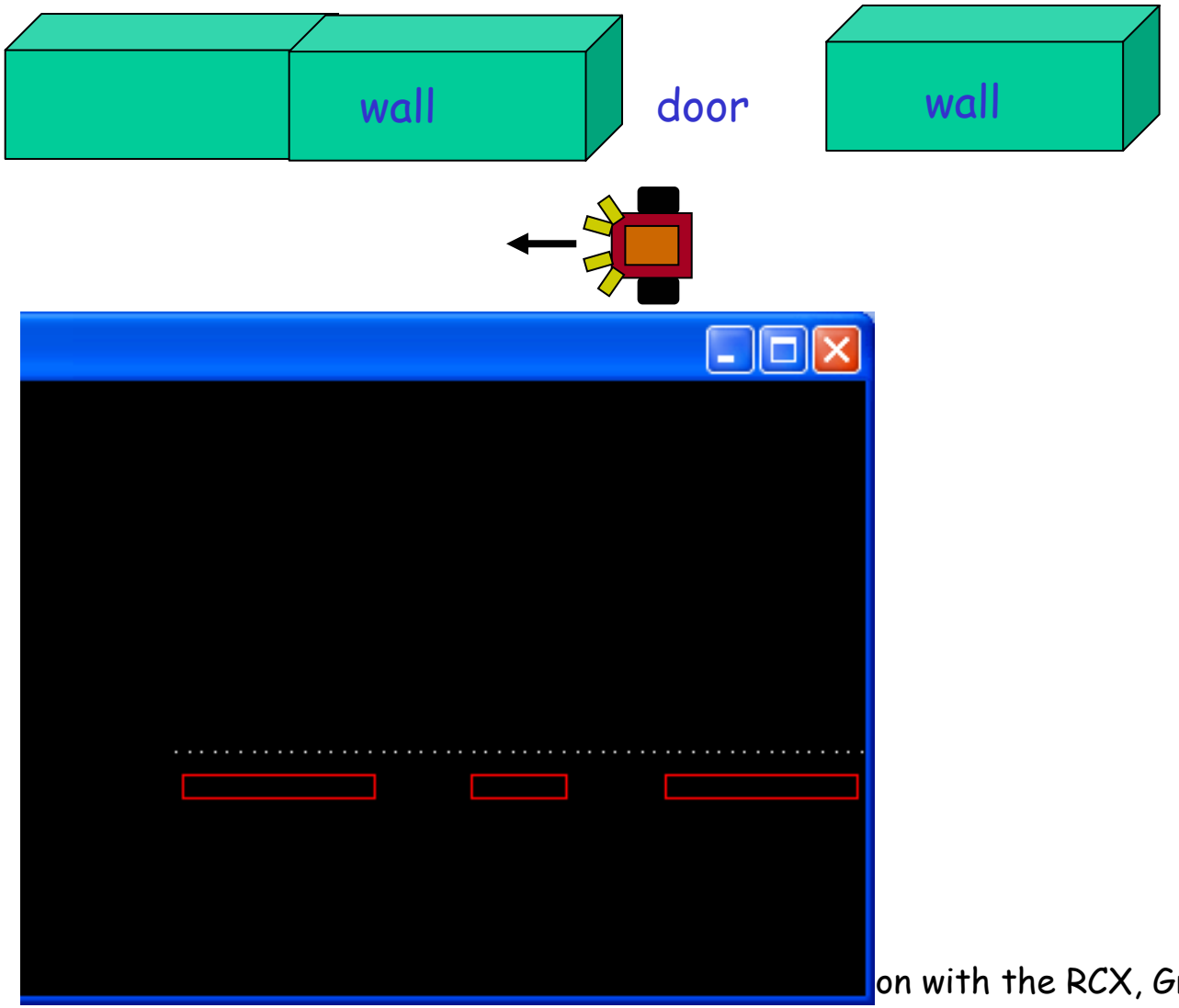
- Particle filtering algorithm:
  - May converge quickly to wrong answer, especially if too few particles
  - Need to add some noise (random samples) to keep from converging incorrectly
  - Need to map fractional movements to integer locations for efficient memory usage
  - RCX memory can be used up quickly – 25-100 samples good starting point
  - Need measure of "location certainty" before moving to goal

- RCX environment issues:
  - Test algorithm off-line first to work out bugs (hard to debug RCX on-line)
    - E.g. use array of sample sonar readings along length of course
  - BrickCC makes a poor IDE – poor error messages, poor comm with tower, difficult configuration
  - Alternative IDEs used: Eclipse (leJOS plug-in), IntelliJ
  - RCXTools (i.e. RCXDownload) works well for compilation and downloading
  - Slow code downloading cycles using IR tower
  - With leJOS behavior arbiter, each behavior needs stop method so that arbiter can gain control
  - leJOS arbiter uses threads
  - leJOS does not have garbage collection
  - Use persistent memory for calibration values
  - Collection classes are available in leJOS and more useful than arrays

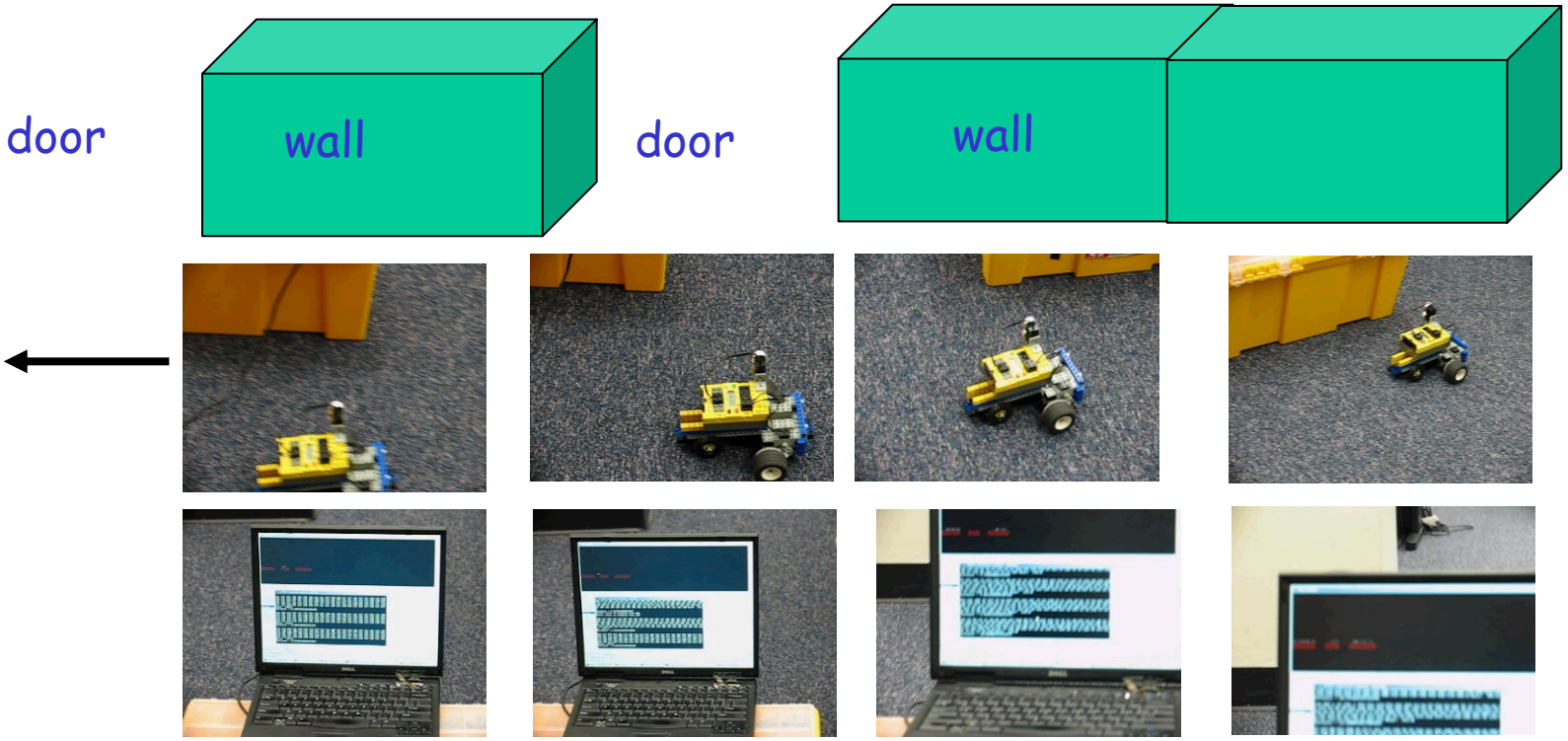
Outline

- Overview
  - The localization problem
  - A simplified educational challenge and RCX solution
- Teaching the solution
- Example solution demonstration
- Hands-on Lab

Demo I (Particle Filtering with RCX and LeJOS): Long or Short Wall?



Demo I (Particle Filtering with RCX and LeJOS): Localization



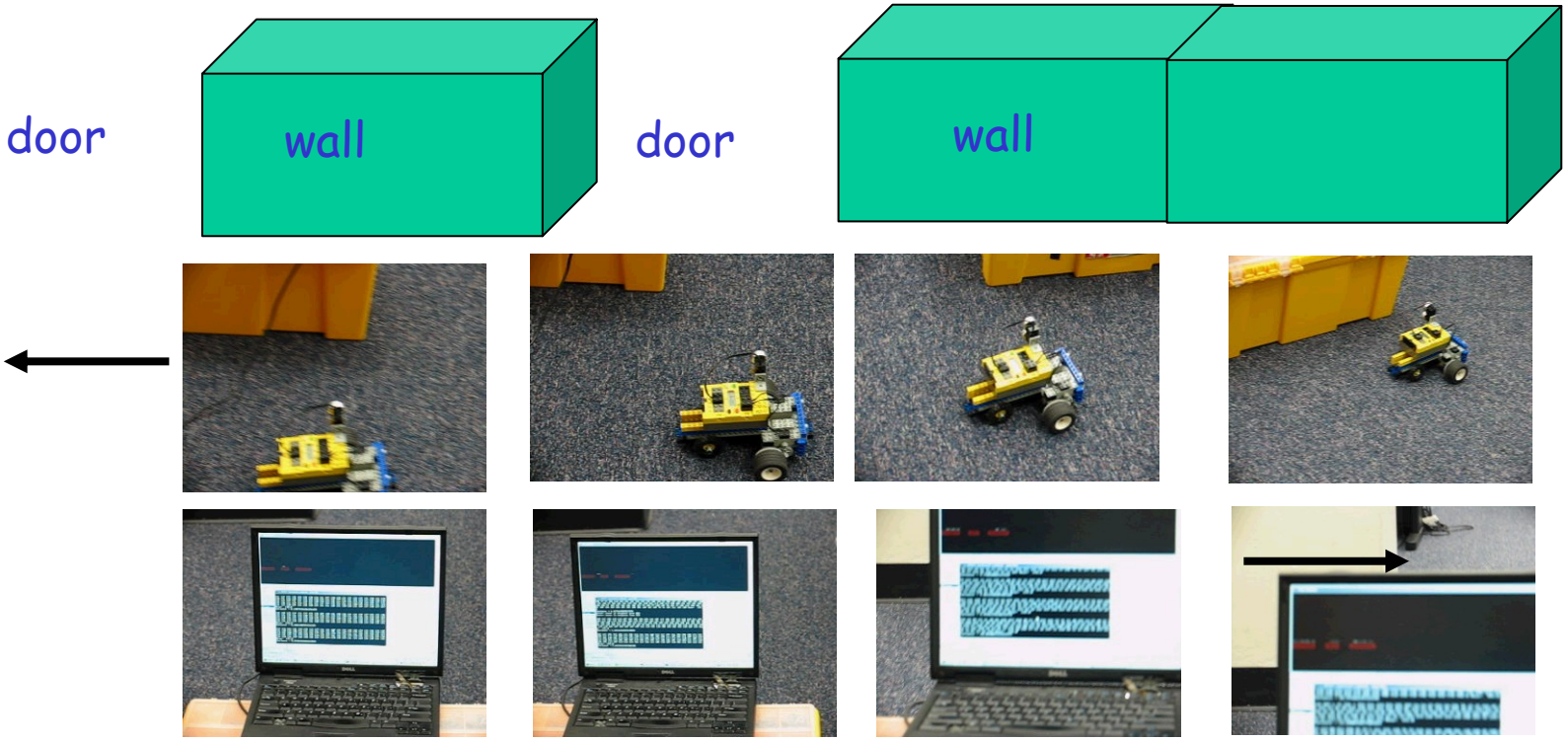
Must Be at Middle Wall

Must Be First Door (after Longer Wall)

Wall But NOT Short Wall Segment

Initialize Samples Uniformly

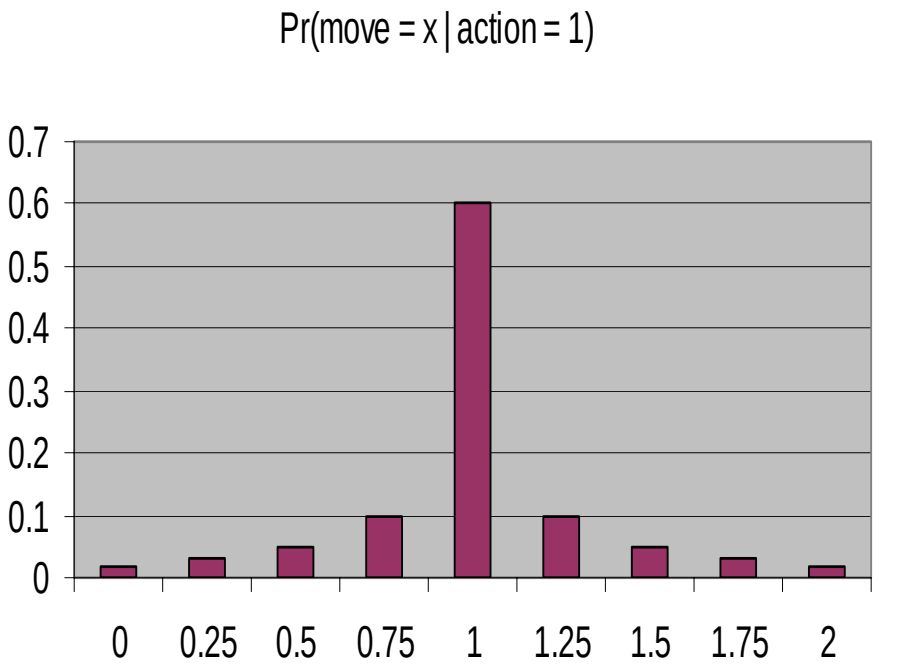
Demo Solution



- Particle Filtering with RCX and LeJOS (link to video)
- Simplifications in demo:
  - no axle rotation sensing – just timing estimates for odometry
  - two motors connected to one port; ignoring orientation errors

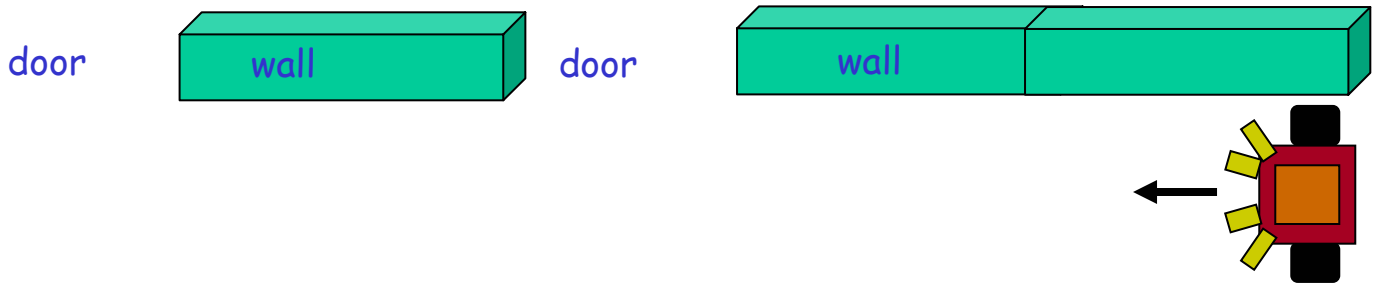
Demo Algorithm Sketch

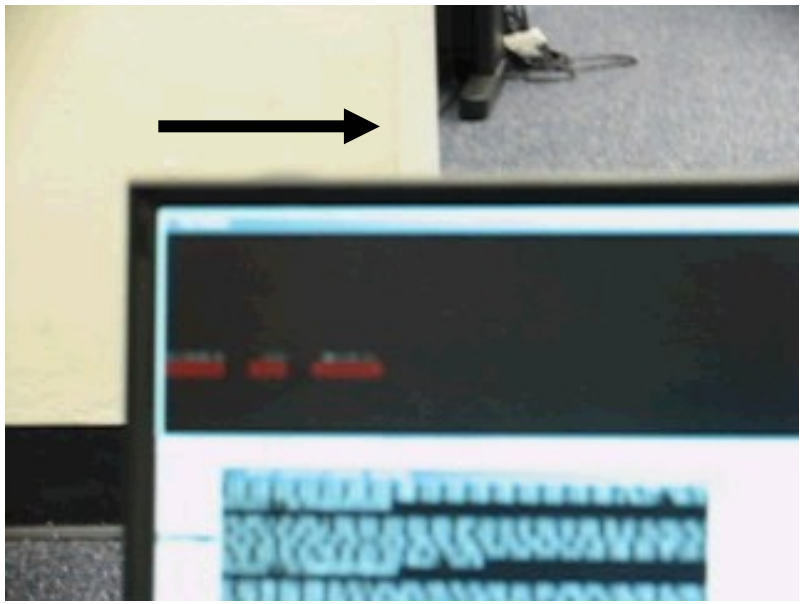
- 25 samples, uniformly distributed
- Loop:
  - Move one inch forward
  - Read sonar
  - Adjust sample weights
  - Resample
  - End if variance in sample location is small
- Adjust sample weights
  - Initialize to 1
  - Compare sonar to map
  - If both say "wall"
    - then weight * 0.6
    - else weight * 0.4



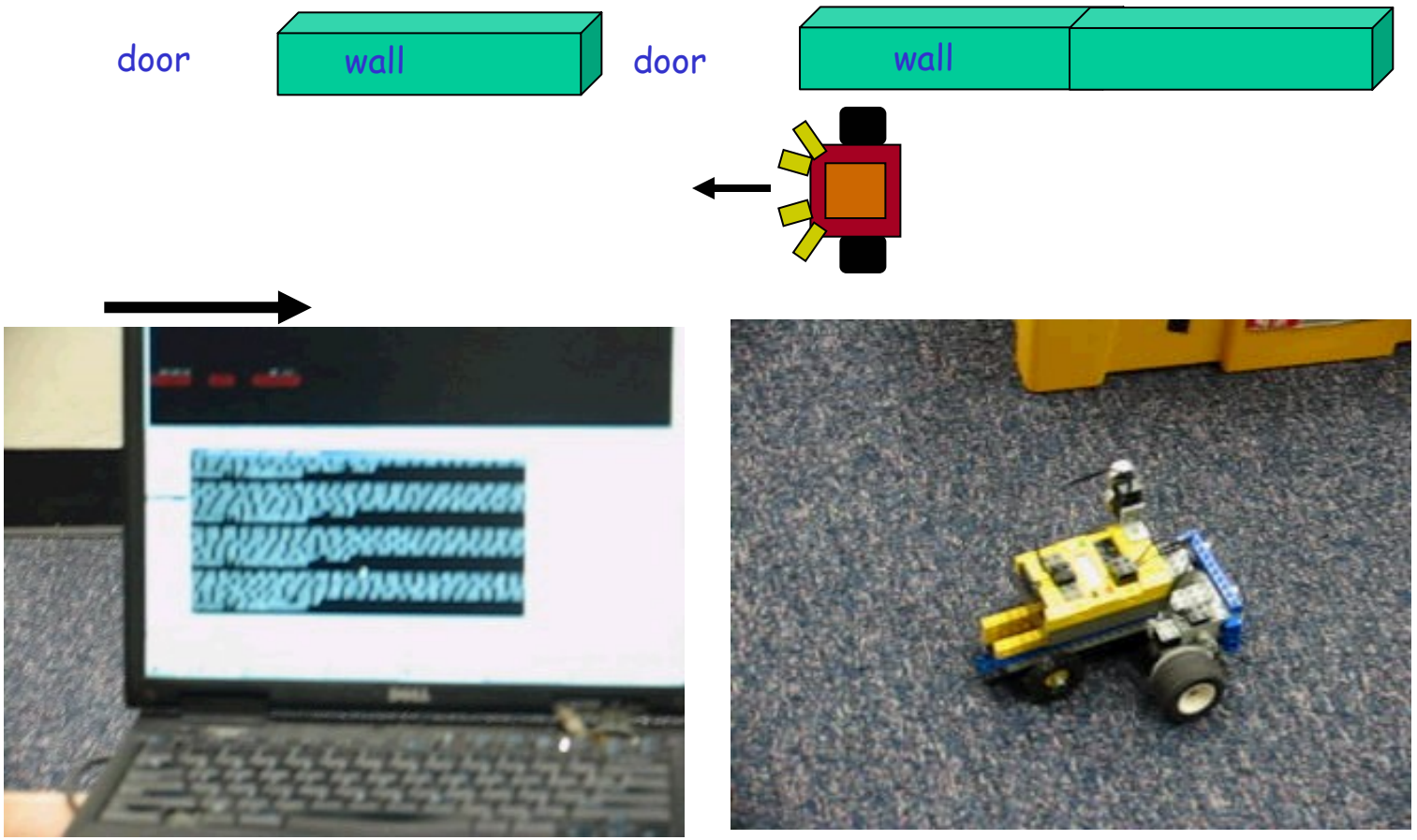
Movement uncertainty model

Initialize Samples Uniformly

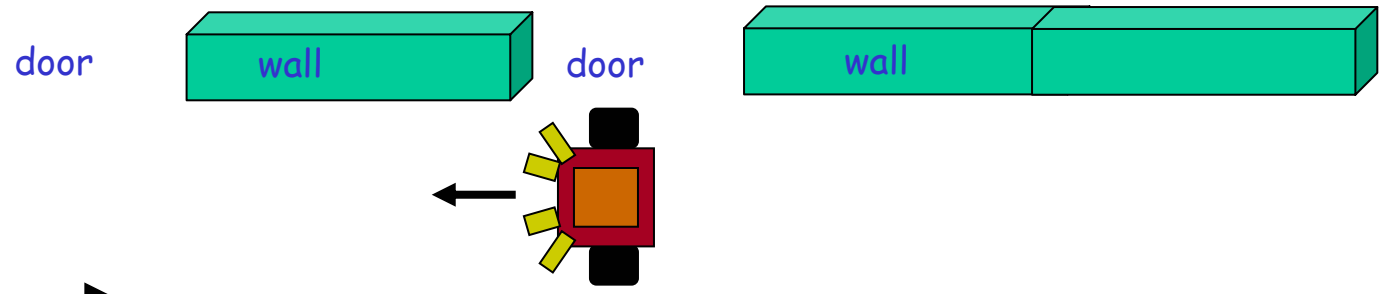


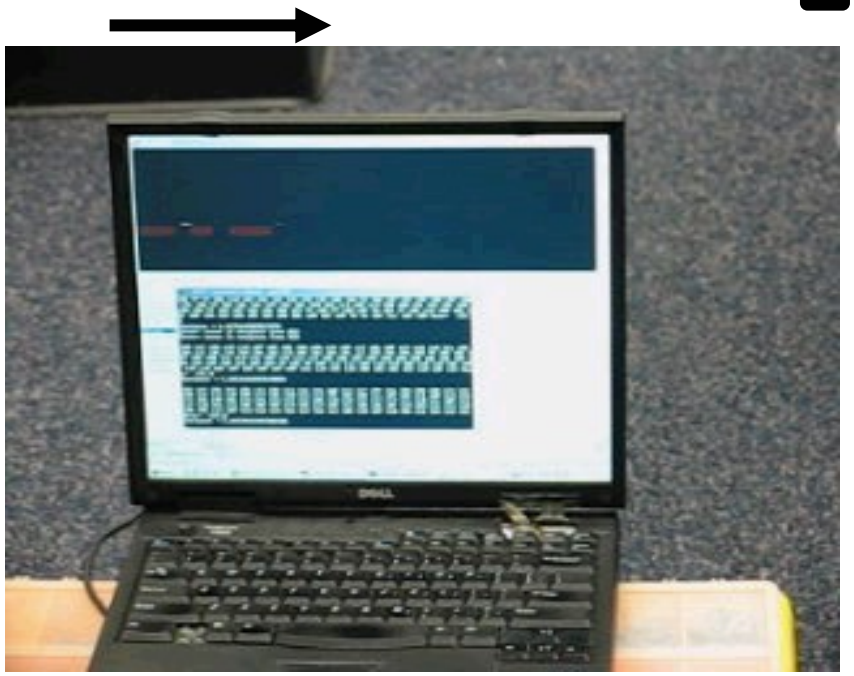


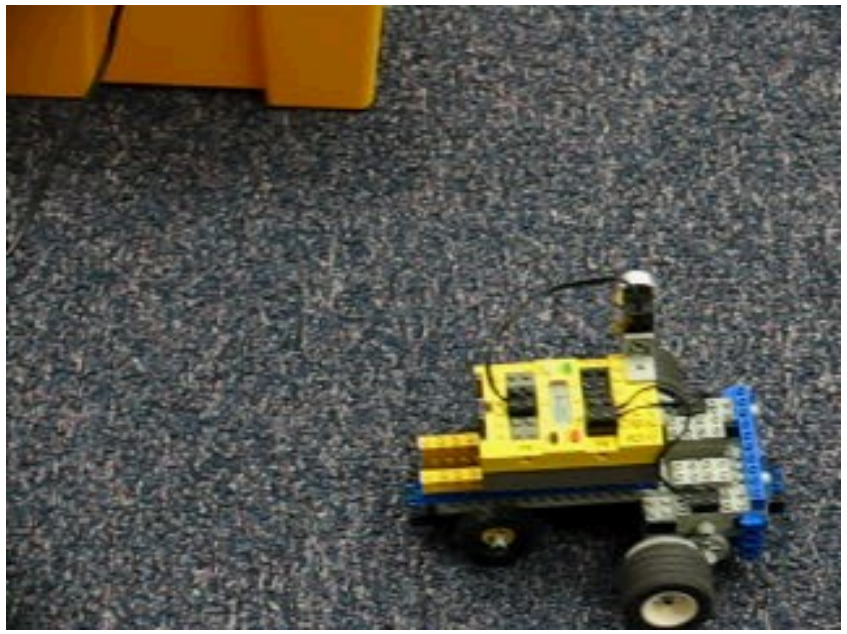
Wall But NOT Short Wall Segment



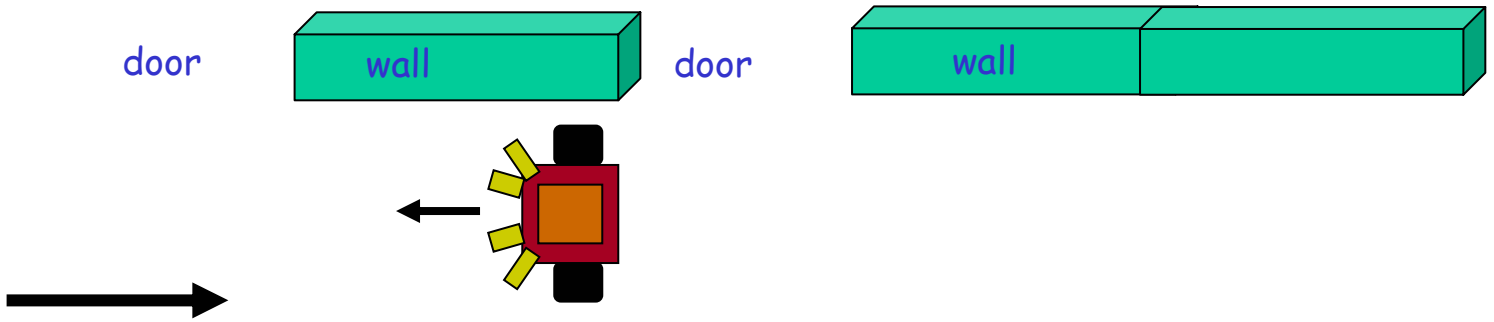
Must Be First Door (after Longer Wall)

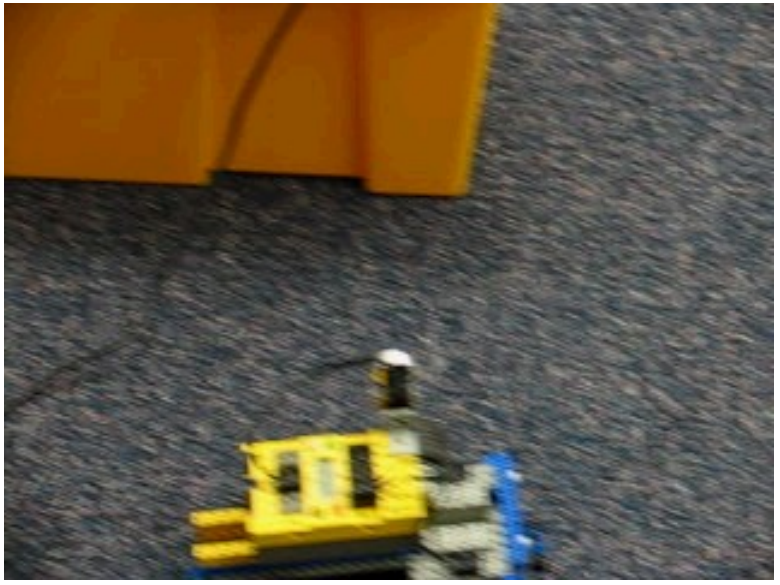






Must Be at Middle Wall





Outline

- Overview
  - The localization problem
  - A simplified educational challenge and RCX solution
- Teaching the solution
- Example solution demonstration
- Hands-on Lab

Hands-on Lab: Materials

- MonteCarloLocalization.zip -- Sample code and lab instructions – contents:
  - MCL.pdf/MCL.doc – instructions (including all code)
  - Testcode directory – RCX Java code to test sonar and navigation
  - MCL_PC directory
    - MCL_PC.java – PC Java code to monitor localization progress – to be compiled and run on PC
    - Irtower.dll – PC libraries needed to run PC code – should be in same directory as MCL_PC.java
  - MCL_RCX directory
    - MCL.java – RCX particle filtering code – to be compiled and run on RCX
    - samples.java – sample transition model
- RCX, Robot base with one or two drivetrains, sonar sensor, zero or two rotation sensors, LEJOS, RCXTools, 2 or more obstacles/walls
- These slides

Hands-on Lab: Building

- Build robot
  - Option 1: two rotation sensors and differential drive (attach sensors to ports 1 and 3)
  - Option 2: zero rotation sensors and single drive train
  - Mount sonar to face one side of robot (attach sensor to port 2)
  - Record the following robot parameters
    - Option 1: Wheel diameter, drive length, gear ratio (in consistent units)
    - Option 2: Time to rotate one revolution (if robot turns), Time to move one meter (in fractional seconds)
- Build linear environment of walls and doors
  - First obstacle should be a wall
  - Record sizes of walls and doors in inches
  - Start with something simple and small
  - Max length = 250 inches

Hands-on Lab: Programming

- PC-side code is already written
  - Displays particles in real-time using IR tower (USB)
  - Load MCL_PC.java onto PC
  - Put irtower.dll in same folder as MCL_PC.java
  - Compile (make sure classpath is correct)
- Experiment with sonar code
- Experiment with transition model

Hands-on Lab: Experiment with Sonar

```
import josx.platform.rcx.*;
import java.io.*;
import josx.rcxcomm.*;
import josx.robotics.*;

public class testsonar
{
   public static void main(String[] args) throws
     Exception
     {
        Sensor.S2.setTypeAndMode(3,0x80);
       Sensor.S2.activate();
      for (;;)
      {
       LCD.showNumber(Sensor.S2.readValue());
       Button.VIEW.waitForPressAndRelease();
      }
    }
}
```

- Attach sonar to port 2
- Place sonar about 12 inches from wall
- Press view button repeatedly to view sonar readings
- Readings are percent
  - 100 percent = wall very far (around 60 inches or more)
  - 0 percent = wall very close (around 12 inches)
  - Somewhere in between gives relative distance of wall
- Pick value that captures maximum reading you get when wall is near – anything greater than this will be considered a door
- Use this value in mcl_rcx.java as WALL
- The program compares the sensor value (below WALL = wall) with the map expectations for a position and uses that to adjust the weight based on the accuracy assumptions of the sonar sensor

Hands-on Lab: Experiment with Navigator Movement Commands

```
import josx.platform.rcx.*;
import josx.robotics.*;

public class testnavigator
{
   public static float wheelDiameter=4.96f, driveLength=9.5f,
      gearRatio=3f; //in cms
      // driveLength in {8.45,10.17} for wide wheels

   public static void main(String[] args) throws Exception
      {
         RotationNavigator nav = new
      RotationNavigator(wheelDiameter,driveLength,gearRatio,
      Motor.A,Motor.C,Sensor.S1,Sensor.S3);

         nav.travel(51); // 51 cm = roughly 20 inches

      //uncomment these lines to try out rotation and
      // moving in a point-to-point square
      //          nav.rotate(360);
      //          nav.gotoPoint(40,0);
      //          nav.gotoPoint(40,40);
      //          nav.gotoPoint(0,40);
      //          nav.gotoPoint(0,0);
      }
}
```

- Record dimensions in inches or centimeters
- Drivelength sensitive to flooring
- Run this program a few times to try to determine the accuracy of traveling 2 inches
- Use to modify samples.java (transition model)
- Can also use to experiment with lejOS odometry

(Materials developed with Babak Shir)

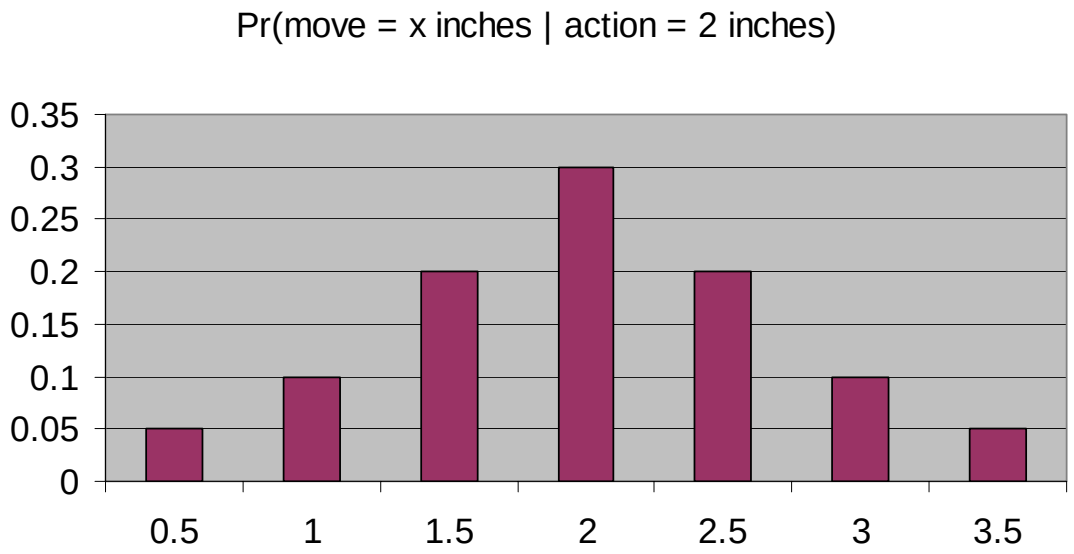
Class RotationNavigator

RotationNavigator(float wheelDiameter, float driveLength, float ratio)
Method Summary
void backward() : Moves the RCX robot backward until stop() is called.
void forward() : Moves the RCX robot forward until stop() is called.
float getAngle(): Returns the current angle the RCX robot is facing.
float getX(): Returns the current x coordinate of the RCX.
float getY(): Returns the current y coordinate of the RCX.
void gotoAngle(float angle): Rotates the RCX robot to point in a certain direction.
void gotoPoint(float x, float y): Rotates the RCX robot towards the target point and moves the required distance.
void rotate(float angle): Rotates the RCX robot a specific number of degrees in a direction (+ or -).
void stop(): Halts the RCX robot and calculates new x, y coordinates.
void travel(int dist): Moves the RCX robot a specific distance.

(Materials developed with Babak Shir)

Hands-on Lab: Experiment with Transition model (samples.java)

- Given a command to travel 2 inches, where might the robot end up?




```
if(r<0.05)
        this.x=(short) (x+1*action);
//robot moves one grid or half an inch
    else if(r<0.15)
        this.x=(short) (x+2*action);
//robot moves 2 grids or one inch
    else if(r<0.35)
        this.x=(short) (x+3*action);
//robot moves 3 grids or one inch and half
    else if(r<0.65)
        this.x=(short) (x+4*action);
//robot moves 4 grids or two inches
    else if(r<0.85)
        this.x=(short) (x+5*action);
//robot moves 5 grids or two inches and half
    else if(r<0.95)
        this.x=(short) (x+6*action);
//robot moves 6 grids or three inches
    else
        this.x=(short) (x+7*action);
//robot moves 7 grids or three inches and half
```

Hands-on Lab: Particle Filtering Execution

- Start MCL_PC.java on PC-side
- Position IR tower near RCX
- Compile, download and start MCL.java on RCX
  - Robot will move forward and stop when it localizes
  - If it fails to localize after moving the full map length it will move backwards and continue trying to localize
  - MCL_PC will show localization progress in real-time
  - RCX LCD will display mean estimate of location in inches

Lab Tips

- Make sure you have irtower.dll in the same folder as MCL_PC.java
- Do not forget to add LejOS libs folder to classpath
  - For example: add it to system properties>Advanced>Environment Variable> System variable
  - Variable=classpath
    Value=.;C:\lejos\lib\classes.jar;C:\lejos\lib\pcrcxcomm.jar
- MCL_PC is not compatible with serial towers
- If your robot does not move accurately, you can change the transition model in Sample.java
- If your samples are converging in the wrong places, re-consider your sonar WALL threshold
- LejOS has some memory leak so running the program for a very long time may cause a system crash
- Due to limited memory samples are not redistributed or added during localization. So, the robot shouldn't be moved during the localization process