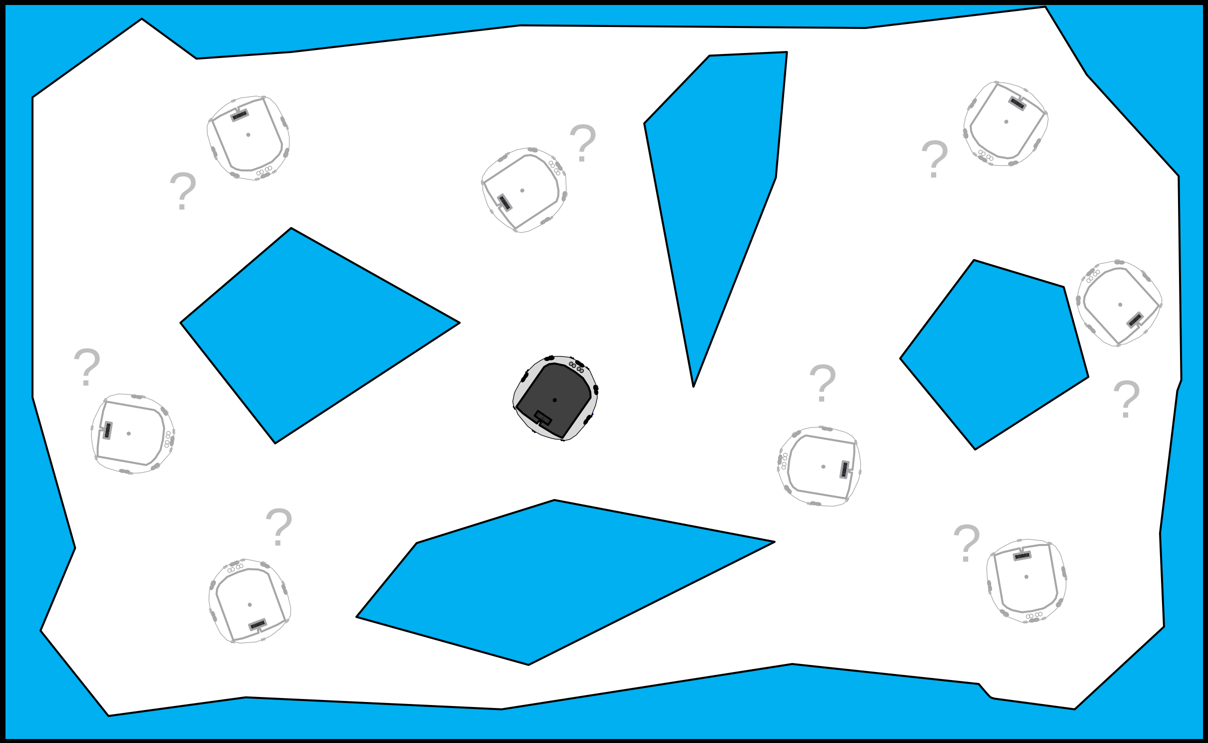
Localization

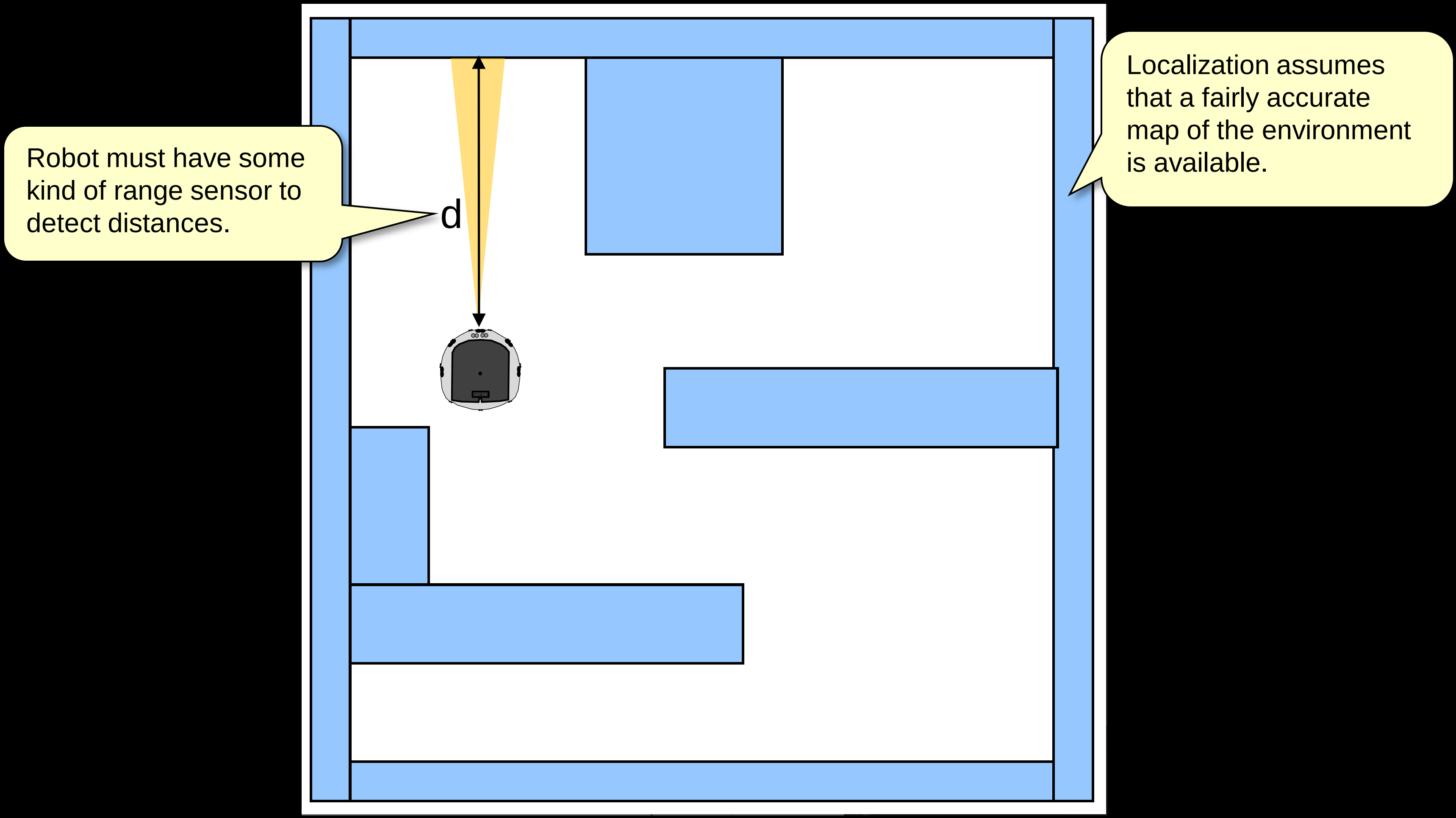
Localization

- Given a robot and a map, find the robot’s pose
  (i.e., where the robot is and which direction it faces)
  - There are many algorithms … we will look at Monte Carlo Estimation (a.k.a. Particle Filtering)



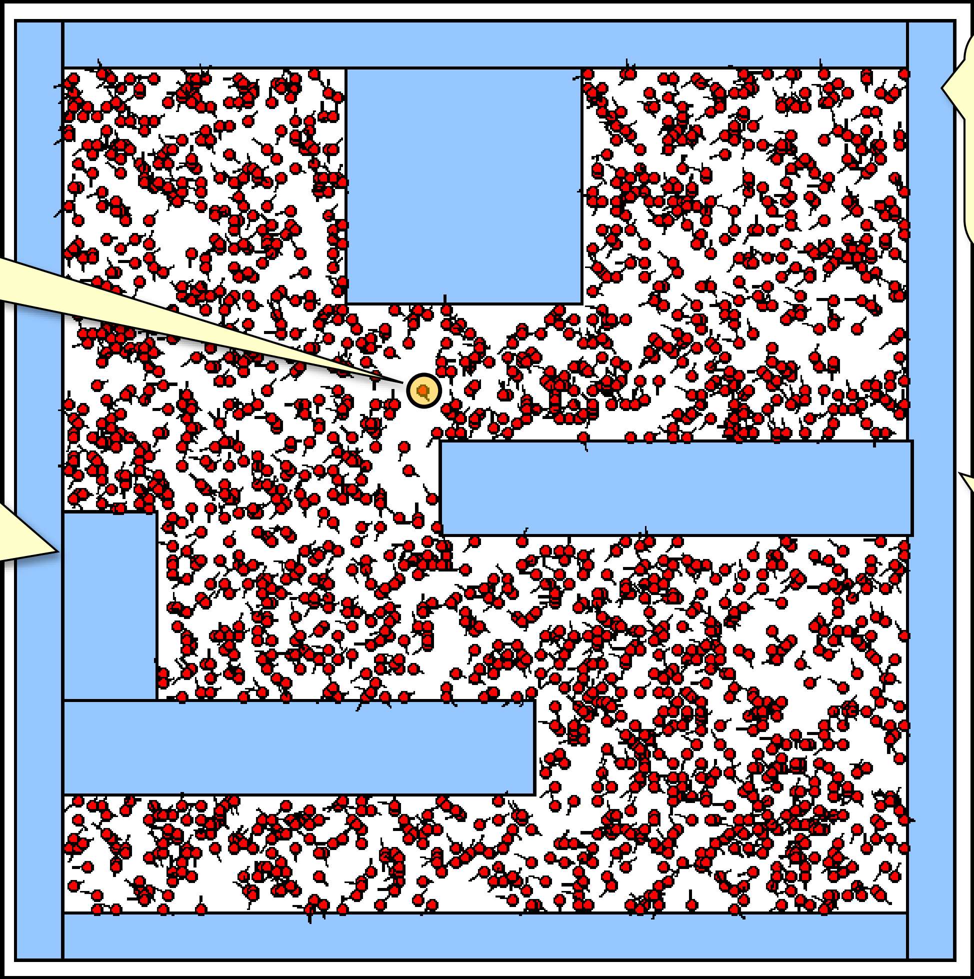
Monte Carlo Localization

- Assume robot is entirely enclosed within its environment at all times.



Monte Carlo Localization

- Start by computing a LOT (i.e., 2000 used here) of random poses as *initial estimates* of where the robot is.



For explanation purposes, assume that the real robot is actually here.

Each red circle indicates the "potential location" of the robot.

The black line indicates the orientation of the robot for that potential pose.

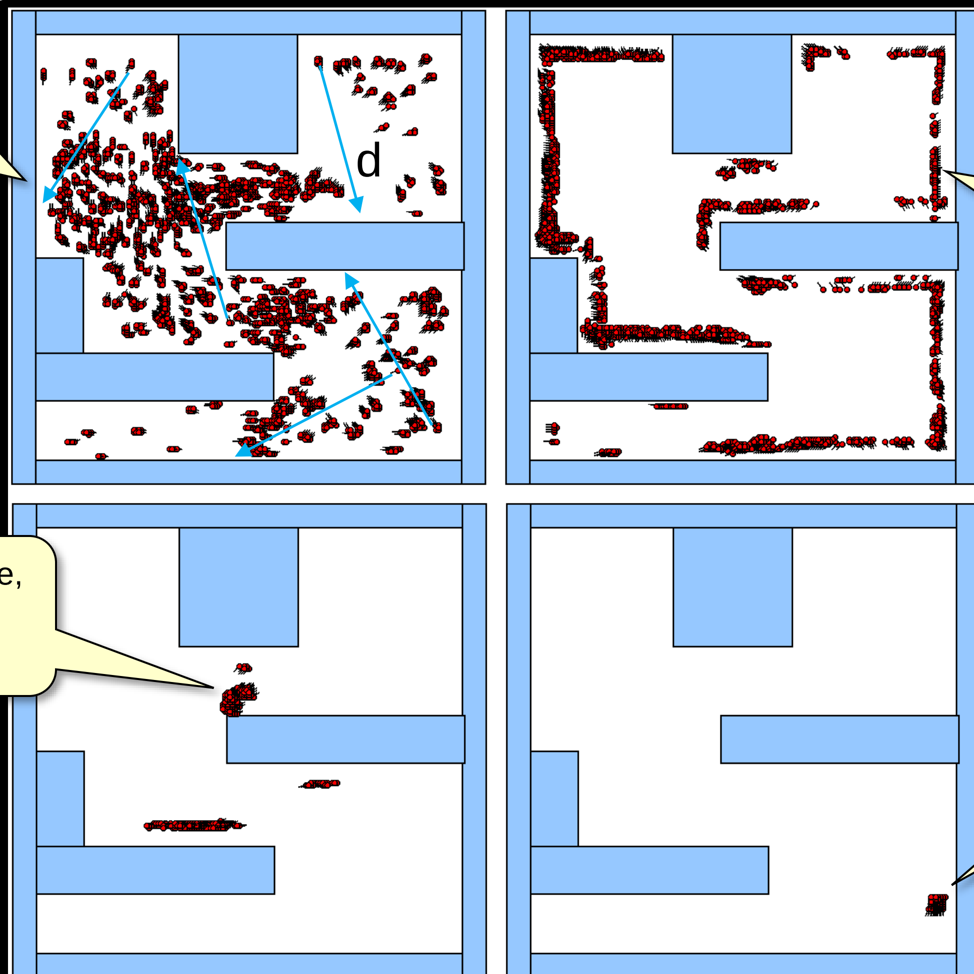
Need to check to ensure that no pose estimate lies **inside** of an obstacle, otherwise discard it.

Need to also check to ensure that no pose estimate lies outside of the boundary (just check against **minX**, **minY**, **maxX** and **maxY** of all obstacles).

Monte Carlo Localization

- As time goes on, many of these poses will be discarded and replaced by "better" estimates.

Here, all remaining poses are distance **d ± 20%** away from an obstacle.



Robot moves around and takes readings in order to narrow down the estimate. Here it can be seen that the robot knows it is close to a wall.

Over time, clusters will form.

Ideally, one cluster remains. The robot is assumed to be here.

Monte Carlo Localization

- Algorithm is as follows:

```
FUNCTION Localize(ArrayList<Obstacle> obstacles, NUM_SAMPLES, %TOL)
   currentEstimates = a list of NUM_SAMPLES randomly chosen poses
   REPEAT {
      message = getMessageFromRobot()
      IF (message is a new distance reading) THEN
         d = getDistanceReading()
         estimateFromReading(currentEstimates, d, obstacles, NUM_SAMPLES, %TOL)
      IF (message is a motion update from forward movement) THEN
         distance = getDistanceMovedForward()
         updateLocation(currentEstimates, distance, %TOL)
      IF (message is a motion update from turning) THEN
         angle = getAngleTurned()
         updateOrientation(currentEstimates, angle, %TOL)
   }
```

Wait until the robot sends some new information …

Now re-compute the **currentEstimates** based on this latest distance reading.

If robot moved forward, move all estimates forward by that amount.

If robot turned, turn all estimates by that amount.

Monte Carlo Localization

- When a distance reading is obtained … update estimates:

```
FUNCTION estimateFromReading(currentEstimates, d, obstacles, NUM_SAMPLES, %TOL)
    goodEstimates = an empty list
    FOR (each pose p in currentEstimates) DO
        IF (isGoodEstimate(p, d, obstacles, %TOL)) THEN
            Add p to goodEstimates if it is a valid pose
    IF (goodEstimates is empty)
        currentEstimates = reset to NUM_SAMPLES random poses
    OTHERWISE
        c = NUM_SAMPLES / number of poses in goodEstimates
        Clear the currentEstimates list
        Add c copies of each pose in goodEstimates to currentEstimates
```

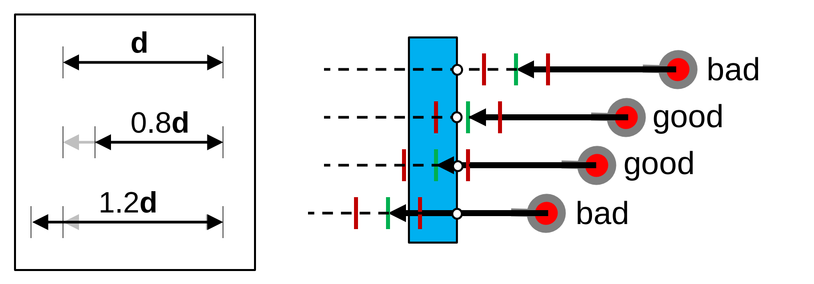
Go through the poses and keep only the "good" ones. Check also that it is valid (i.e., it does not lie within an obstacle).

Duplicate the good ones so that we always have a constant number of samples.

Call **resetEstimates()**

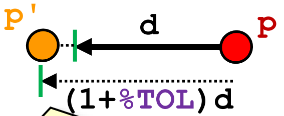
To make a copy, you need to call the constructor: **`new Pose(p.x, p.y, p.angle)`**

- How do we know if an estimate is "good" ?
  - Check intersection with closest obstacle. If within reasonable error tolerance (e.g., 20%), then it is a good estimate.



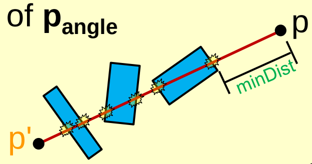
Is the Estimate Good?

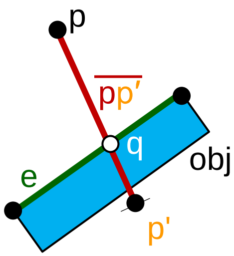
```
FUNCTION isGoodEstimate(p, d, obstacles, %TOL)
    Compute point p' to be (px+(1+%TOL)d·COS(pangle), py+(1+%TOL)d·SIN(pangle))

    minDist = infinity;
    FOR (each obstacle obj in obstacles) DO
        FOR (each edge e of obj) DO
            IF (pp' intersects e) THEN
                q = intersection of pp' with e
                IF (q is not null)
                    qDist = distance from p to q
                    IF (qDist < minDist) THEN
                        minDist = qDist

    IF (minDist is still infinity) THEN
        RETURN false

    IF (minDist > (1-%TOL)d AND minDist < (1+%TOL)d) THEN
        RETURN true
    RETURN false
```



$\mathbf{p'}$ is %TOL further than **d** from $(\mathbf{p_x},\mathbf{p_y})$ in direction of $\mathbf{p_{angle}}$

Minimum distance from $(\mathbf{p_x},\mathbf{p_y})$ to an obstacle in the direction of $\mathbf{p_{angle}}$





Use the **distance** function in the Point class that takes another point's **x**, **y** coordinates as parameters.

Find distance to closest obstacle intersection point **q** among all obstacle edges.

If no intersection, estimate is bad.

If reading is within %TOL range, it is a good estimate.

Monte Carlo Localization

- Updating estimates when a robot moves forward …

```
FUNCTION updateLocation(currentEstimates, d, %TOL)
    FOR (each pose p in currentEstimates) DO
        randomPercent = %TOL·(2R - 1)
        p_x = p_x + d·(1 + randomPercent)·COS(p_angle)
        p_y = p_y + d·(1 + randomPercent)·SIN(p_angle)
```

R is a random number such that $0 \le R < 1$.

The randomness is needed so as to allow the poses to vary slightly over time. This is crucial for the algorithm to work, otherwise it may never converge to a proper pose or may lose a good pose due to inaccurate robot movements.

Move forward by distance **d** cm plus a random amount from -%TOL to +%TOL

Math.random() gives a random # in range from 0 to 0.9999999

- Updating estimates when a robot spins …

```
FUNCTION updateOrientation(currentEstimates, angle, %TOL)
    FOR (each pose p in currentEstimates) DO
        randomPercent = %TOL·(2R - 1)
        p_angle = p_angle + angle·(1 + randomPercent)
```

Turn by **angle** degrees plus a random amount from -%TOL to +%TOL

Start the Lab ...