Robotics & Autonomous Systems

A Practical Introduction with NXT and JAVA

Localization & Mapping

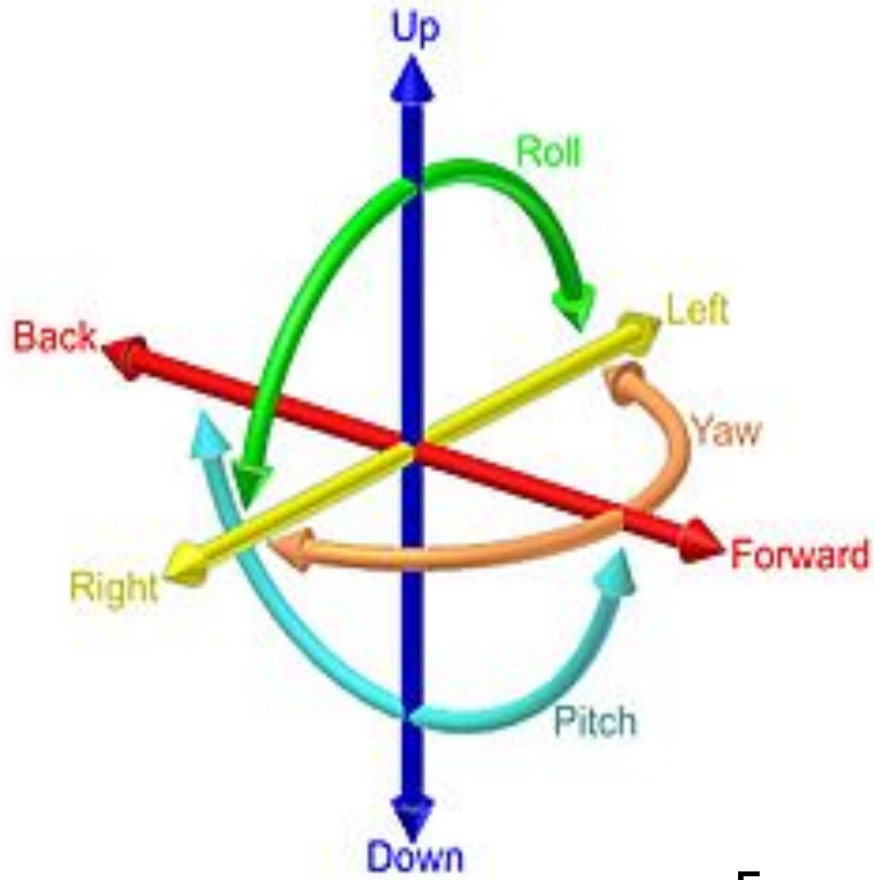
Michael Wooldridge
(mjw@liv.ac.uk)

UNIVERSITY OF LIVERPOOL

Pose

- A robot's *pose* is that part of its state that is external to the robot
- How complex a robot's pose is will depend on the type of robot, and in particular, how many *degrees of freedom* it has
- The number of degrees of freedom a robot has are the different ways that it could move

Six Degrees of Freedom



Examples: aircraft, spaceship

Pose for NXT Robots

- Our NXT robots have only 2 degrees of freedom:
  - Forward/backward
  - Left/right
- An NXT robot's pose can be defined by a triple (x,y,θ)
  - X is x coordinate relative to some known start point
  - Y is y coordinate relative to some known start point
  - θ is the robot's heading (in degrees).
- For a robot arm or humanoid robot, pose would include information about configuration of arms & legs (hence name pose)

Localization and Mapping

- *Localization* is the problem of determining the robot's current pose
- *Mapping* is the problem of the robot figuring out a map of its environment – what is around it.
- Localization is easier if you have a map…
- …mapping is easier if you know your pose…
- But usually we don't have available either pose *nor* a map!
  - A chicken and egg situation!

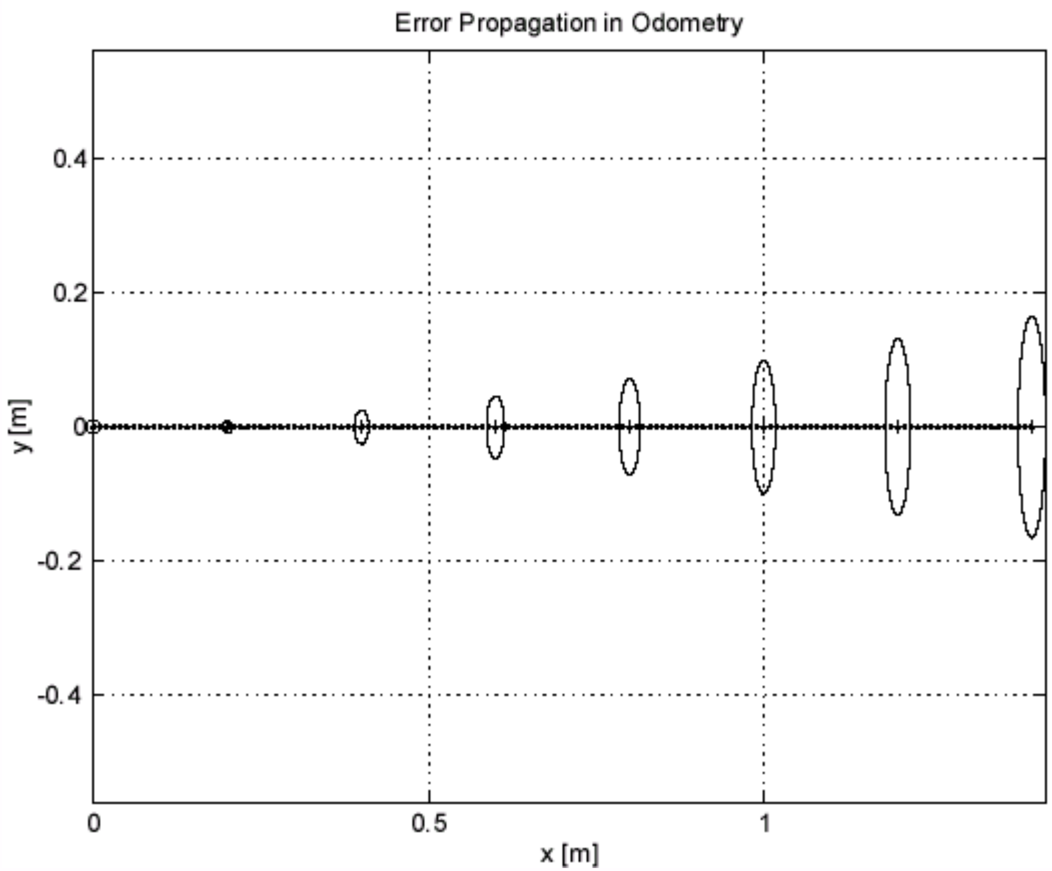
Simultaneous Localisation & Mapping (SLAM)

- If we don’t know our pose and don’t have a map available, then we do *simultaneous localization & mapping*
    - Build a map of the environment *while* figuring out pose

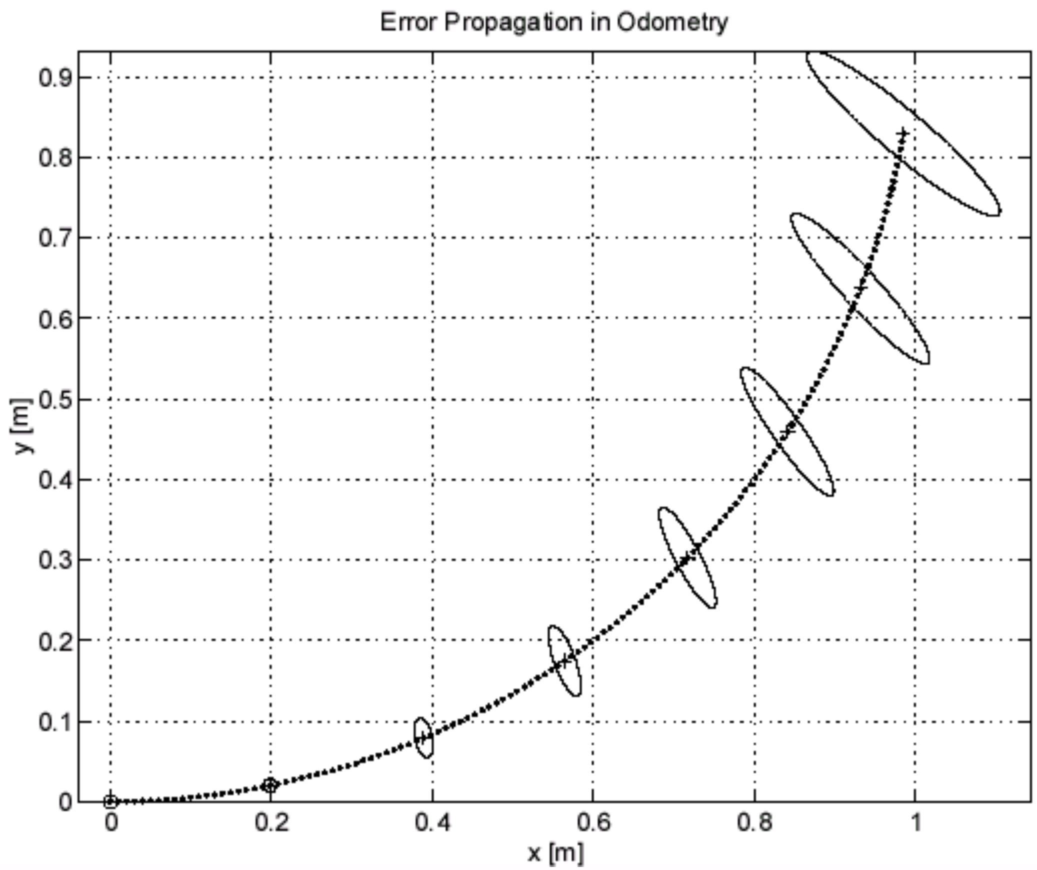
Localization via Odometry

- *Odometry* is the most basic form of localization technique
- *Odometry data* is the raw data we get record from our motors/actuators
  - Tachometer count etc
- Given raw odometry data, we can use a form of dead reckoning to figure out where we are and where we are heading
- However, odometry is error prone

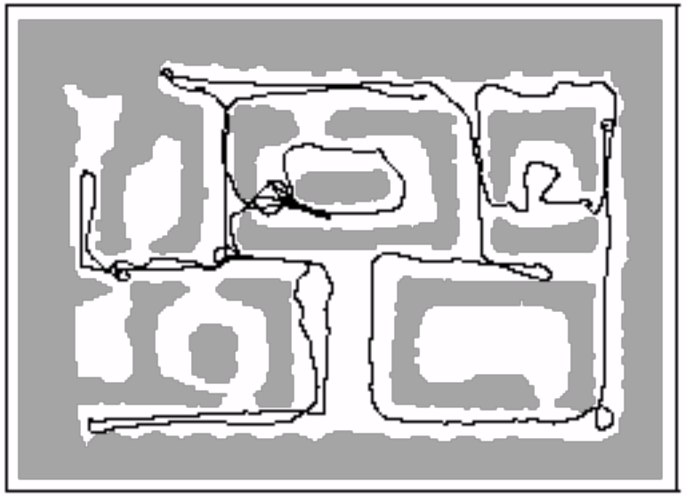
Error Propagation in Odometry: It's bad even if we travel in a straight line…



Error Propagation in Odometry:
… but it's much worse when we turn!



Error Propagation in Odometry



The course the
robot actually took…

…and the course according
to raw odometry data.

Odometry in LEJOS

- We we use the DifferentialPilot class to control movement, then we can use an associated class, OdometryPoseProvider to keep track of an interpret our pose
- This greatly simplifies odometry-based localization, BUT…
  - Must be careful ONLY to control movement via DifferentialPilot – don't use other motor APIs!
  - It is only as accurate as (NXT) odometry can be…

OdometryPoseProvider

- The constructor takes as a parameter a DifferentialPilot object
- The DifferentialPilot will keep the OdometryPoseProvider informed about motor movements etc
- Key method is `getPose()`, which returns returns an object of type Pose
- On Pose objects, we have:
  - `float getX()` – get X coordinate
  - `float getY()` – get Y coordinate
  - `float getHeading()` – get current heading (degrees)

Prog11.java (page 1 of 3)

```
import lejos.nxt.*;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.robotics.localization.OdometryPoseProvider;
import java.util.Random;

public class Prog11 {

public static void main(String[] args) throws Exception {
  DifferentialPilot dp =
      new DifferentialPilot(3.22f, 19.5f, Motor.B, Motor.A);

  OdometryPoseProvider opp = new OdometryPoseProvider(dp);

  Random randomGenerator = new Random();

  int move;
```

Prog11.java (page 2 of 3)

```
dp.travel(10);
System.out.println("pose = " + opp.getPose());
Thread.sleep(4000);

dp.travel(20);
System.out.println("pose = " + opp.getPose());
Thread.sleep(4000);
```

Prog11.java (page 3 of 3)

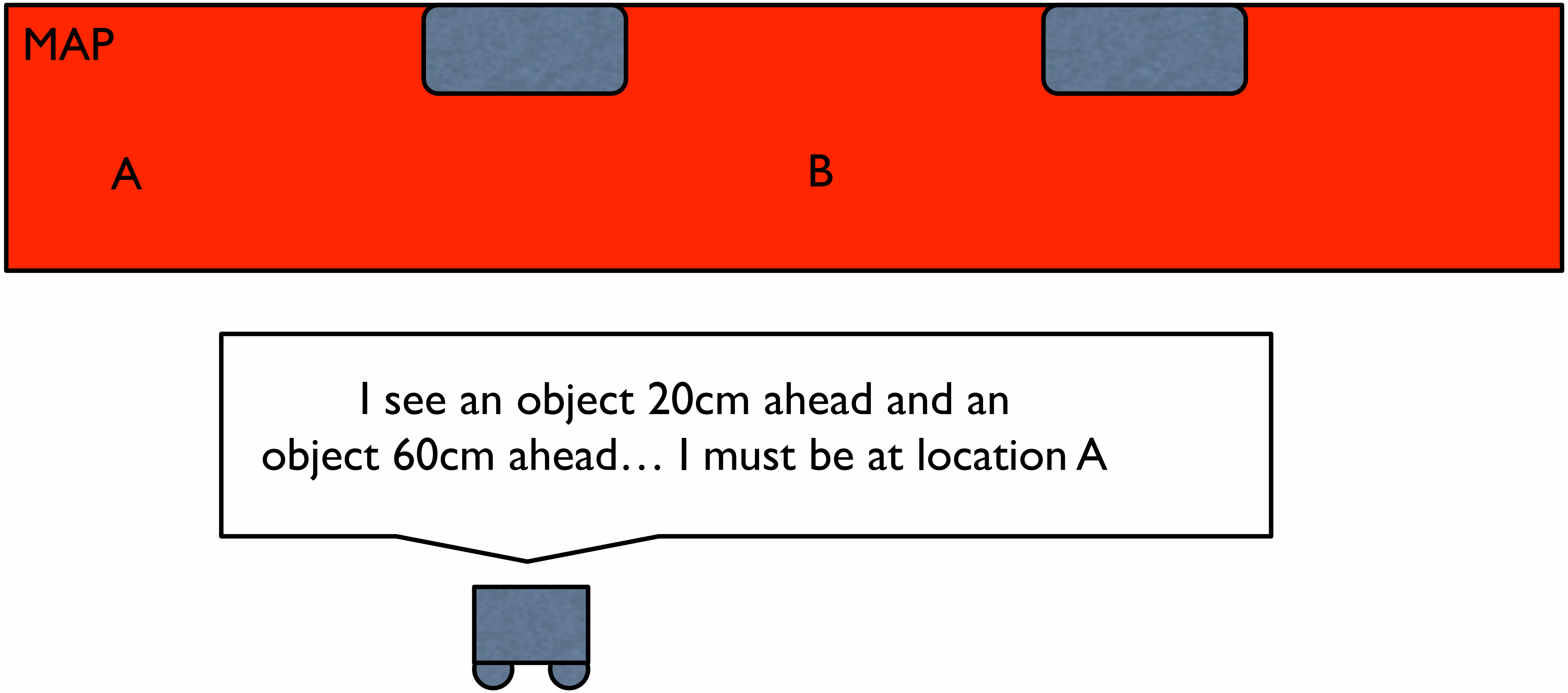
```
        // make 10 random movements
        for (int i=0; i < 10; i++) {
                move = randomGenerator.nextInt(2);
                if (move == 0) { // move forward
                        dp.travel(10); //Moves forward for 10cm
                        dp.stop();
                } else {
                        dp.rotate(90);
                }
                System.out.println("pose = " + opp.getPose());
                Thread.sleep(4000);
        }
        Button.waitForPress();

    }
}
```

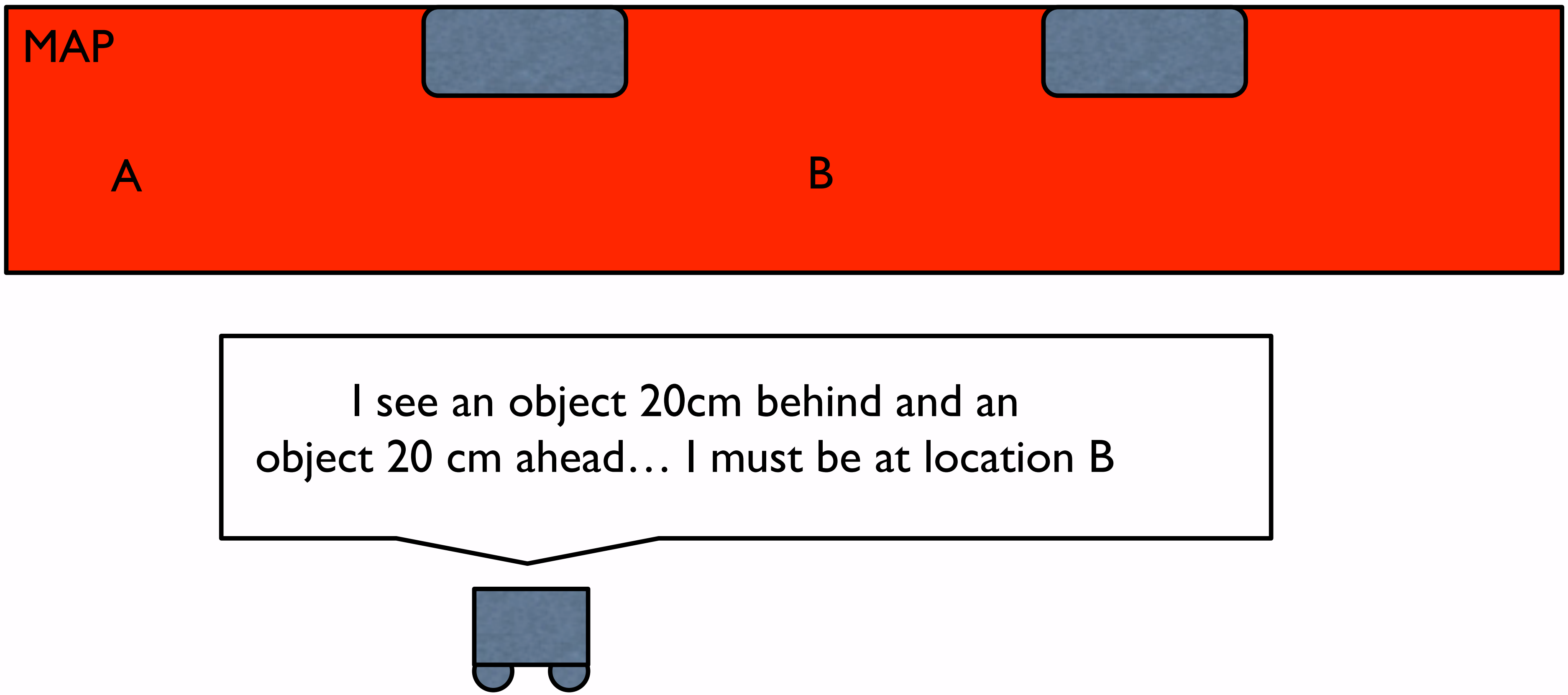
Localisation with Maps

- If you have a map, then you can use the information this map provides to augment raw odometry data
- Basic idea: sensors gives you *percepts* which carry *information*… you can use this information to eliminate possibilities about where you are
- But of course: *sensors are not reliable!*
- *The solution is to <u>make multiple observations…</u>*

Using Maps for Localisation



Using Maps for Localisation



Occupancy Grid Mapping

- The most intuitive form of mapping is *occupancy grid mapping*
- An occupancy grid is a 2 dimensional array of cells, with each cell having the value
  - 1 is the cell is occupied
  - 0 otherwise
- More generally, a cell value can be the *probability* that the cell is occupied

Occupancy Grids

1. Let M[xmax,ymax] be an array of integers, initialised to 0 with xmax being the maximum x coordinate, ymax being the maximum y coordinate – this will be the map
2. Let C[xmax,ymax] be a corresponding array counting the number of observations we've made of each cell – initialised to 0
3. Make an observation (eg with ultrasonic sensor)
4. Given current pose, determine which cells in M are occupied according to the current observation, and
    1. Increment count C[x,y] of each cell in perception field of sensor reading
    2. If cell is occupied according to observation, then increment M[x,y], otherwise decrement M[x,y]
5. Repeat from step 3 as often as needed – move around, making a series of observations
6. If C[x,y] == 0 then we have no information on (x,y) – never observed it
7. If M[x,y] > 0 and C[x,y] > 0 then [x,y] is "probably occupied"
8. If M[x,y] < 0 and C[x,y] > 0 then [x,y] is "probably unoccupied"

Bayesian Updating

- What we have described is an approximation of a sophisticated technique called *Bayesian mapping*
- This technique makes use of *Bayes rule*: a rule which tells us how we can update beliefs on the basis of observations

Bayes Rule

- Let P(X) denote the probability of event X occurring
- Let P(Y) denote the probability of event Y occurring
- Let P(X | Y) denote the probability of event X occurring *given that event Y has already occurred*
  - *The event will often be an* ***observation***
- Values P(X) and P(Y) are *priors* (prior probabilies)
- Value P(X | Y) is a *posterior*

Bayes Rule

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}$$

Allows us to compute a posterior probability
from priors and a known posterior.

Bayes Rule: Example 1

- Imagine you are a doctor seeing a patient, who has red spots (RS)
- You know the *prior probability* of a patient having lassa fever (LF) is 1/50,000 (i.e., about 1 in 50,000 people has lassa fever)
  - this is P(LF)
- Prior probability of a patient having red spots is 1/20
  - P(RS)
- You know that 50% of patients with red spots have lassa fever
  - P(RS | LF)

Bayes Rule: Example 1 (continued)

- P(RS | LF) = 0.5
- P(LF) = 1/50,000
- P(RS) = 1/20
- Then:

$$
\begin{aligned}
P(LF \mid RS) &= \frac{P(RS|LF)P(LF)}{P(RS)} \\
&= \frac{0.5 \times 1/50000}{1/20} \\
&= 0.0002
\end{aligned}
$$

Bayes Rule: Example 2

- What is the probability that someone you meet on the street in Liverpool wearing a blue shirt is an Everton fan?
- $P(B) = 0.1$ [about 1 in ten people wears a blue shirt]
- $P(E) = 0.01$ [about 1 in a hundred people like Everton]
- $P(B \mid E) = 0.8$ [80% of Everton fans wear blue shirt]
- $P(E \mid B) = (0.8 * 0.01)/0.1 = 0.08$
- This makes sense: blue shirts are about 10 times more popular than Everton…

Bayes Rule and Mapping

- What does Bayes rule have to do with mapping?
- A conditional probability P(M | R) can be understood as
  - "the probability of M occurring given that we've observed R"
- In mapping, we can read this as
  - **"the probability that the map M is correct given that we have sensor reading R"**
- We can use Bayes rule to *rationally update our map, based on observations that we've made!*

Bayesian Mapping

- Let MAPS = {M1, M2, …, Mk} be the set of all possible maps
- Let's assume these maps are 2 dimensional occupancy grid maps, of dimensions 0 .. XMAX, 0 .. YMAX.
- Each cell M[x,y] takes value either 0 or 1
  - M[x,y] == 1 indicates cell (x,y) occupied
  - M[x,y] == 0 indicates cell (x,y) empty
- How many elements are there in MAPS? $2^{XMAX*YMAX}$
  - For our 6 x 8 grid that's about 4 trillion maps…

Bayesian Mapping

- Let t be the current time
- Let (S1, …, St) denote all sensor readings to time t
- Let (J1, …, Jt) denote all poses up to time t
  - Assume here that we *know* the poses Ji
- The aim is to compute for each M ε MAPS the value
  - P(M | S1, …, St, J1, …, Jt)
  - That is, the posterior probability that the map M is correct given that we've seen sensor readings S1,…,St and been through poses J1, … Jt
- We have to compute this value for MANY maps! (4 trillion for our NXT example!!)

Bayesian Mapping

- Instead of computing posterior for each *map*, we instead compute it for each *grid cell*
- Instead of having to compute 4 trillion posteriors, we only have to compute 6 * 8 = 42 posteriors

Bayesian Mapping

- Current time is t, and M'[x,y] be previous map estimate, and S is current sensor reading
- Let M be current map (ie the one we're constructing)
- For each grid cell (x,y) in perceptual field of S:
  - Compute P(M[x,y] | S) =
    - (P(M[x,y]) * P(S | M[x,y])) / P(S)
  - For P(M[x,y]) we can use the probability computed on previous iteration, i.e., M'[x,y]