CSCI 1106
Lecture 22

Robotics Review

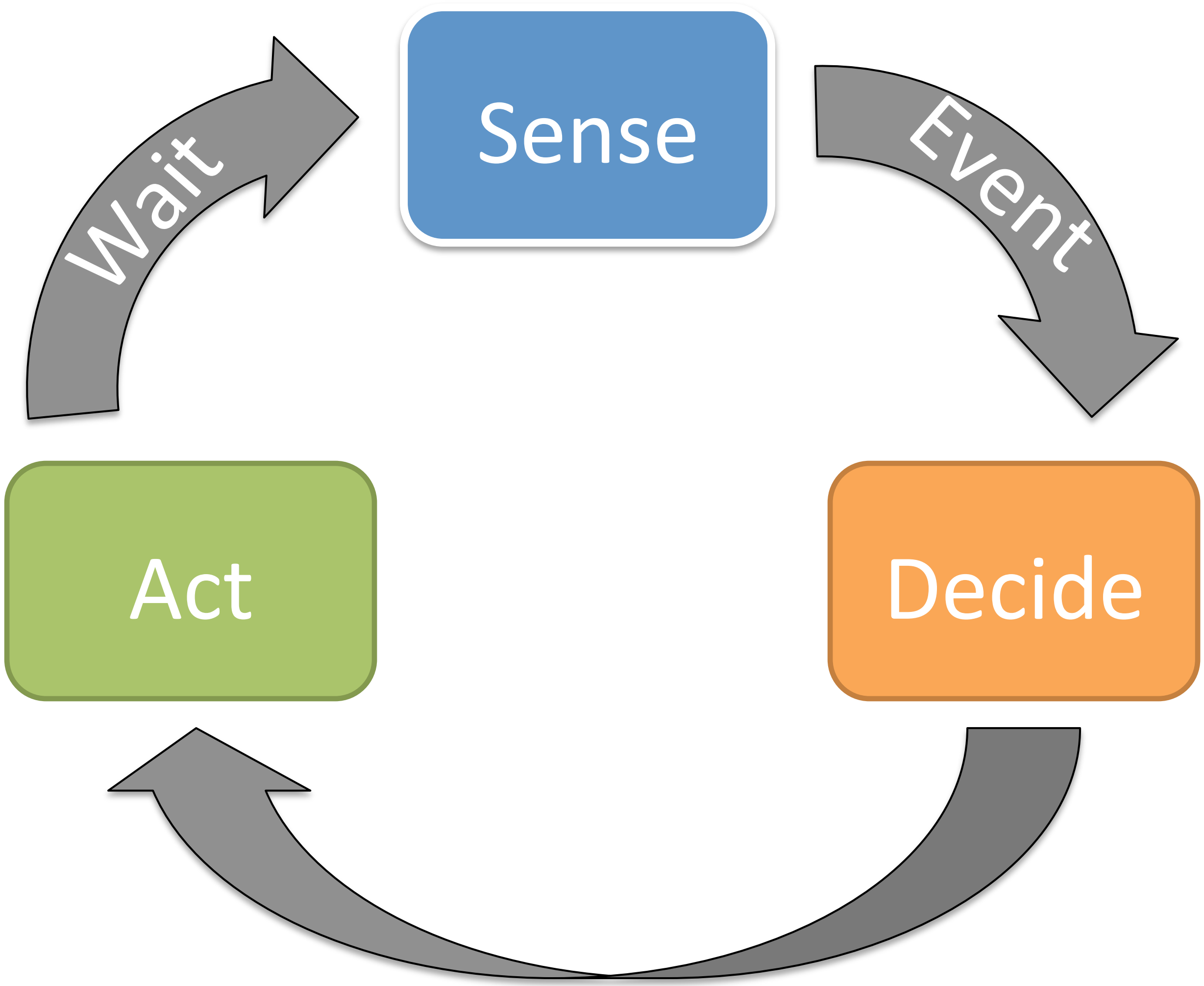
What is Robotics

- From the OED:

  "*Robotics*: The area or science of design, construction, operation, and application of robotics and the like; the study of robots."

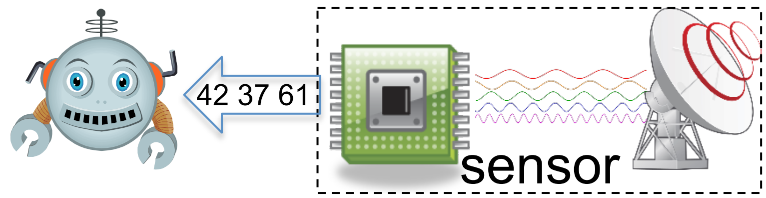
- Anatomy of a Robot
  - Hardware Components:
    - Sensors
    - Control
    - Actuators
  - Software Components:
    - Sensor Input Processing
    - Decision Making
    - Actuator Manipulation and Output

Event Driven Framework
(Wait) Sense (Event)-Decide-Act



Sensors

- A *sensor* senses a property in its environment
  - Input: Analog
  - Output: Discrete



- Have a variety of characteristics
  - *Sensitivity* : minimum change of input that results in change in output
  - *Range* : the minimum and maximum inputs that a sensor can handle
  - Response : the output of sensor for a given input
  - *Response Time* : how quickly the sensor can change state as a result of a change of input
  - *Precision* : degree of reproducibility of the measurement
  - *Accuracy* : maximum difference between the true and measured value
  - ***Bias*** **: the systemic error of the sensor**
  - ***Variability*** **: the random deviation from the true value**

Sensors are Imperfect

- Sensors have two kinds of errors
- Key Ideas:
  - No matter how good a sensor is, it is imperfect
  - Imperfect sensors introduce *uncertainty*
    - *Bias*
    - *Variability*
  - Need to quantify the uncertainty
  - Need to quantify a sensor's characteristics
- Can characterize sensors through a standard process

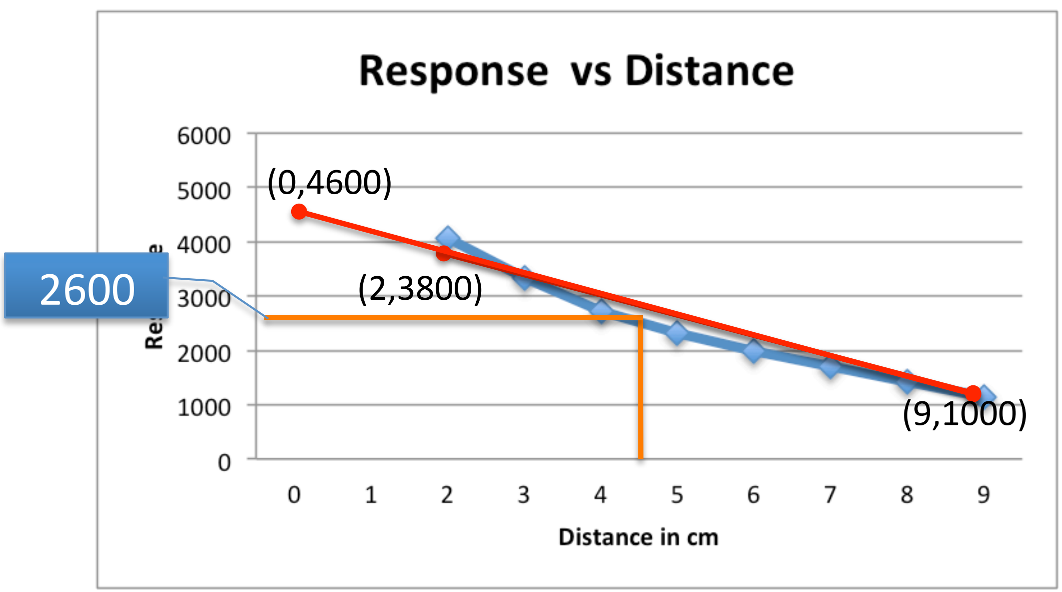
How to Characterize a Sensor

1. Identify the sensor we want to characterize
2. Identify the sensor characteristic we want to measure
3. Identify the possible variables of the characteristic
4. Fix all but one of the variables
5. Create a sequence of known ``actual'' values where the
   - One variable is varied and
   - All other variables are fixed
6. Perform a sequence of measurements (*multiple times)* on the ``actual'' values
7. Tabulate the results and compute means
8. Plot the results
9. Repeat steps 4 – 8, allowing a different variable to vary each time
10. Analyze the plot(s) to derive the sensor's characteristics

Making Use of the Results

- General observation(s)
  - Response decreases as distance increases
  - Useful for visual interpolation
- Create a linear model
  - Draw a linear approximation
  - Compute slope (*m*) and intercept (*b*) of the line
  - Plug into equation of a line
- Then what?

$$m = \frac{rise}{run} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{1000 - 3800}{9 - 2} \cong -400$$



$$x = 2, y = 3800$$

$$y = mx + b \Rightarrow 3800 = -400 \times 2 + b$$

$$b = 4600$$

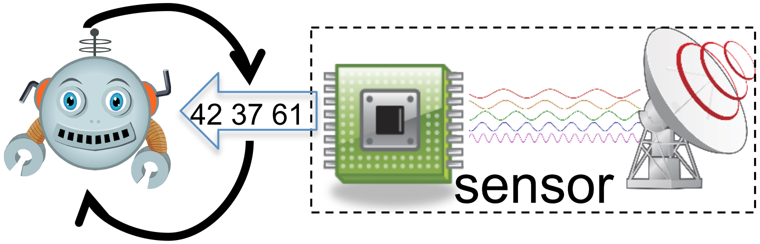
$$y = mx + b \;\rightarrow\; y = -400x + 4600$$

Using Sensors

- Key Ideas:
  - A sensor will not inform your program when a property has changed
  - The program must poll the sensor repeatedly to detect change
- A program
  - *Polls* the sensor to get its current value
  - *Interprets* the value (compares it to a threshold)
- Defn: A *threshold* is a fixed constant such that an event is triggered when a measurement from a sensor returns a value that is above (or below) the constant

Sampling

- Polling Frequency depends on
  - The response time of the sensor
  - The rate at which the environment changes
- The *sampling rate* is the frequency of the polls
- A higher rate means we are
  - Less likely to miss a change in inputs
  - Using more CPU time to poll the sensor



Sensor Variability

- Problem: All sensors have some variability
  - The measured value randomly deviates from the true value
  - A sensor may report different values for the same true value
- Solution:
  - Take multiple measurements
  - Aggregate (mean, median, mode) the results

Actuators

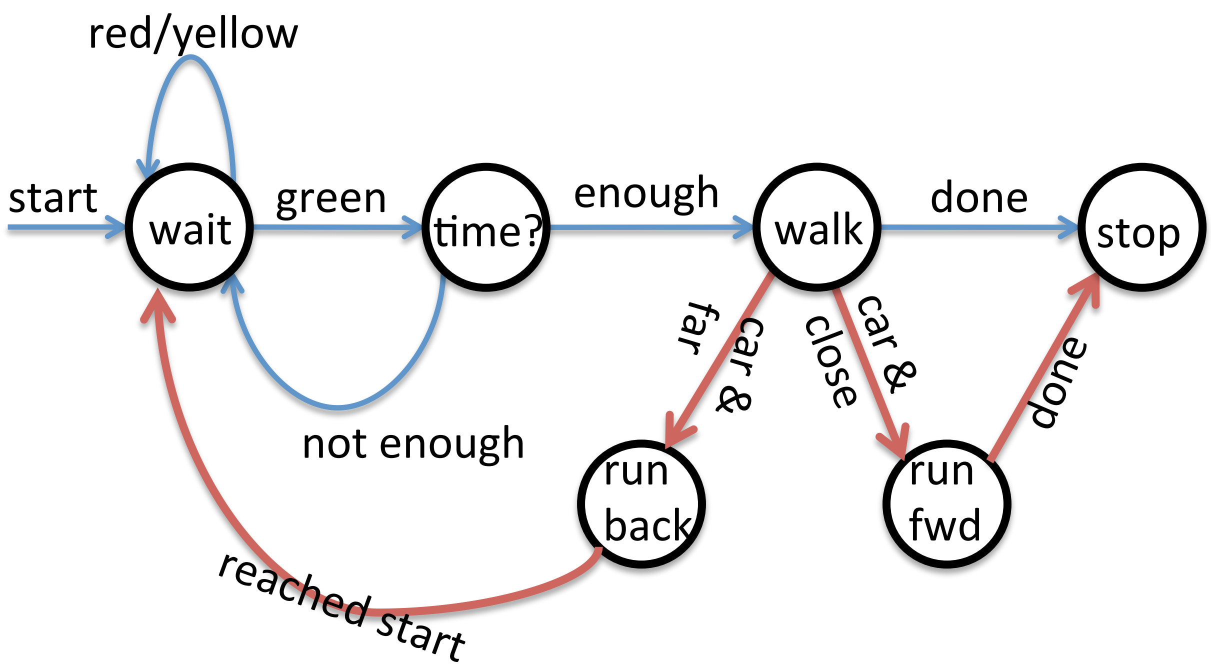
- Actuators allow the robot to affect the world
- Actuators are characterized by their parameters and tolerances:
  - Torque, force, and pressure
  - Speed, power, and strength
  - Accuracy and precision
- Two kinds of uses
  - Synchronous use:
    - Start operation
    - Wait until the operation completes
    - Continue program
  - Asynchronous use:
    - Start operation
    - Continue program
    - Use sensors or poll actuator to determine operation completion

State Transition Diagrams

- Idea: Use state transition diagrams to model
  - Steps of a task
  - Conditions under which the steps are performed
  - Environment of the robot during the task
- Consists of states and transitions
- A *state* is a unique set of conditions that hold at a given time
  - System can only be in one state at a time
- A *state transition* occurs when
  - An *event* occurs
    - External events (sensor input)
    - Internal events (completion of a task, timer)
  - One of the conditions describing the state changes
  - The state of the system changes

State Transition Diagrams

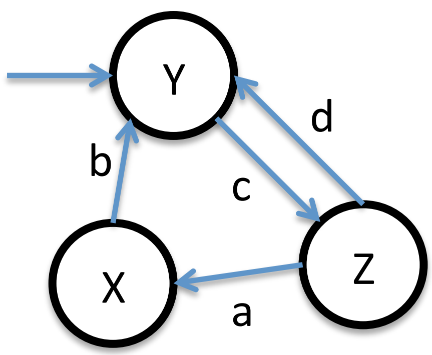
- Idea: We use a state transition diagram to model a task
- States are represented by circles
- Arrows represent transitions between states



- If light is red, wait for light to turn green
- If light is yellow, wait for light to turn green
- If light is green but there is not enough time, wait for light to turn red and then green
- If light is green and there is enough time,
  - Proceed on crosswalk
  - **If a car is speeding at you, get out of the way**
- Stop crossing when other side is reached

Creating State Transition Diagrams

- Identify the states (steps) of a task
  - Determine what actions must be performed
  - Determine groups of unique (relevant) conditions
  - Label each group with a unique name
- Identify state to state transitions
  - What is being sensed?
  - What external events will be sensed?
  - What internal events will occur?
  - What conditions will these events change?
  - Determine which conditions change?
  - Determine the corresponding states in the transition
  - Label each transition with a unique label
- Create diagram
  - Combine states and transitions
  - Refine the diagram by repeating the process
- **This diagram is a blueprint for your program!**



Translating State Transition Diagrams

- Problem:
  - We design our solution by creating a state transition diagram (STD)
  - We need to translate the STD into a program
- Idea: Use a standard process
  - Use a variable to encode the current state
  - Enumerate all states as constants
  - Identify events associated with each transition
  - Gather transition information
  - Implement event handlers to perform the transitions

Tracking and Enumerating States

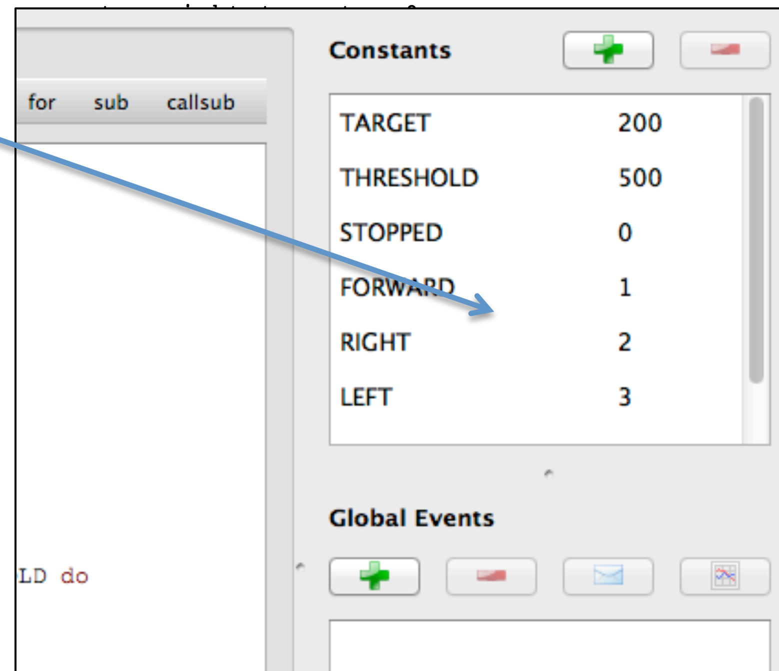
- Use a *state* variable
  - Stores the current state
  - Set to an initial state,
    e.g., STOPPED
- Enumerate all states
  - Select state names
    e.g., STOPPED, RIGHT, LEFT
  - Number consecutively
  - Add states as constants
- Can be done automatically

```
var state = STOPPED

motor.left.target = 0
motor.right.target = 0

onevent button.forward
  state = RIGHT

onevent button.backward
  state = STOPPED
  motor.left.target = 0
```



Identify Events

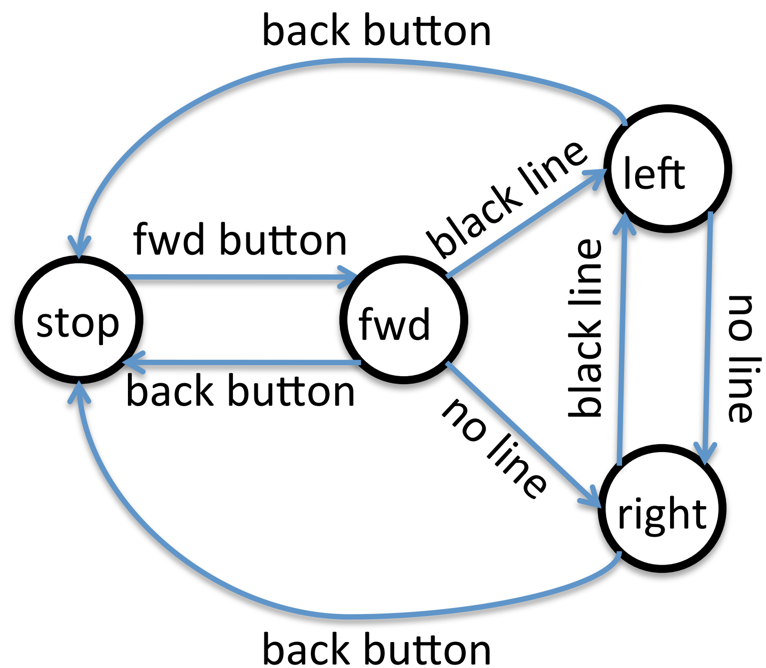
- Identify the events associated with each transition
  - `button.forward`: Forward Button pressed
  - `prox`: horizontal proximity or ground proximity sensors
  - `timer0` or `timer1`: timer has expired
  - `tap`: robot tapped
  - etc
- Add an event handler for each event
  - `onevent button.forward`
  - `onevent prox`
  - `onevent timer0`
- In each handler implement all the transitions associated with the event

Gather Transition Information

- For each transition, identify
  - States (CONSTANTS)
  - Event (handler)
  - Sensor/device
  - Change in sensor/device
  - Thresholds (if any)
  - Action to perform
- E.g., transition: fwd ➔ left
  - States:
    - From: fwd (`FORWARD`)
    - To: left (`LEFT`)
  - Event (Handler): `prox`
  - Sensor: `prox.ground.delta[0]`
  - Change in sensor: response decreases (dark)
  - Threshold: < 500 means dark
  - Turn left

```
motor.left.target = 0
Motor.right.target = 200
```

- Implement the transitions in their event handlers



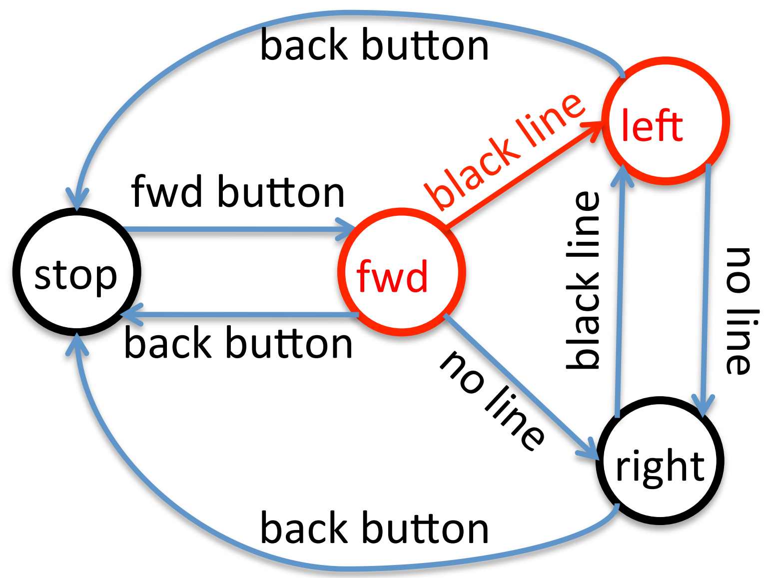
Implement the Transitions

- Inside the handler use template:

```
if state == FROM_STATE and sensor has changed then
  state = TO_STATE
  perform action
end
```

- E.g., transition: fwd ➔ left

```
onevent prox
  if state == FORWARD and prox.ground.delta[0] < 500 then
    state = LEFT
    motor.left.target = 0
    motor.right.target = 200
  end
```



`if` vs `when`

`if`

- Form:

```
if condition then
  body
end
```

- If the condition is true

  the body is executed

- E.g., if we see a stop sign stop, regardless of whether we are already stopped

when

- Form:

```
when condition do
  body
end
```

- If the condition is true now and was not true before, the body is executed
- E.g., if we see a stop sign and we are not stopped, then stop

Dealing with Failure

Things don't always go as planned…

- Need to do two things
  - Identify when a failure has occurred
  - Respond to the failure
- *Failure* is a state that the system should not be in under normal conditions
- *Failure cause is* the physical or functional reason for the failure
- *Failure manifestation* is the detectable effect of the failure
  - To identify failure, it must manifest itself in a detectable way
- Obs: We can only deal with failures that we can foresee

Failure Identification

- Idea: We can identify that a failure has occurred from its manifestation
- To identify a failure, we need to
  - Determine what can cause the failure
  - How the failure manifests
- When designing a program we need to (attempt) to enumerate all relevant failures
- Narrow the enumeration to:
  - Failures we can deal with
  - Failure causes we understand
  - Failure manifestations we can identify

Mechanisms for Detecting Failure

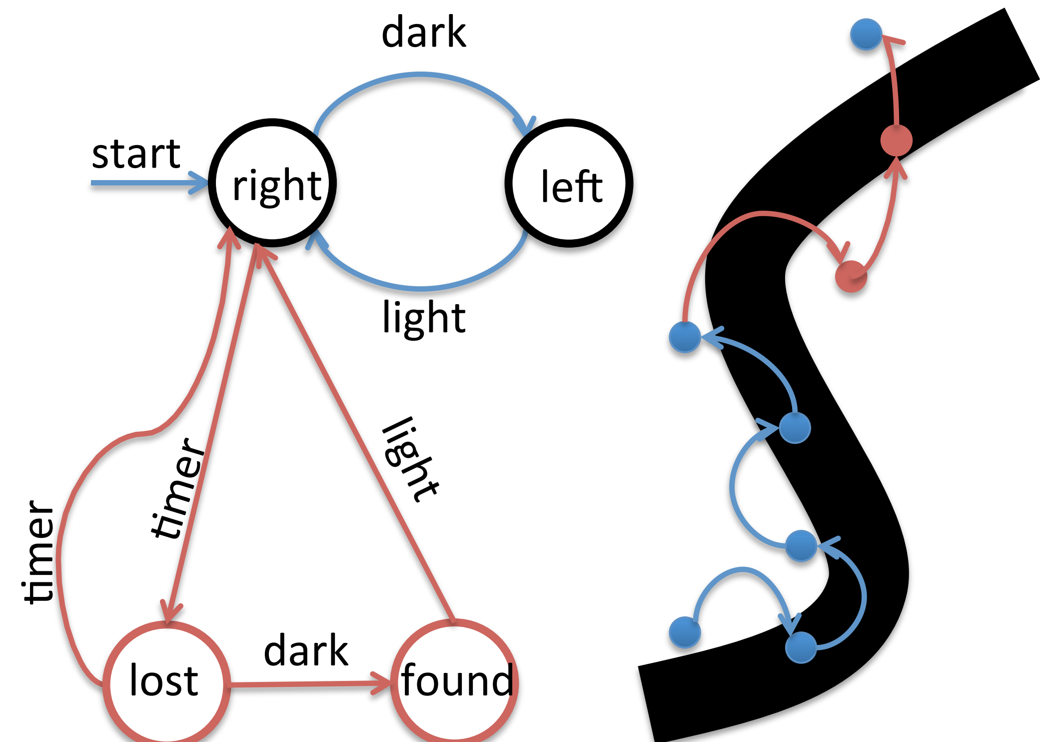
- Unexpected external events
  - Sensors register an unexpected changes in environment
    - Sensors give false readings
    - Sensors give true readings of unexpected conditions
  - Actuators report status errors
    - Actuator fails to perform specified task
    - Actuator reports error where none has occurred
- Lack of expected external events
  - A timer expired while waiting for an expected event
    - Sensor fails to register the expected event
    - Expected event does not occur
  - Actuators fail to move the prescribed amount
    - Encounter unexpected resistance
- Unexpected (or lack there of) internal events
  - Programs run code they are not supposed to (bugs)

Failure Response and Recovery

- Once we determine that a failure has occurred, we need to respond to it
- *Response mechanisms* are parts of the program that respond to the failure
- Two options:
  - Place system in a safe state (shut down)
  - Recover from the failure
- A *recovery mechanism* returns the system to a normal state
- Recovery mechanisms are specific to each failure

Modeling Failure Recovery

- Idea: Use state transition diagrams to model failure identification, response, and recovery

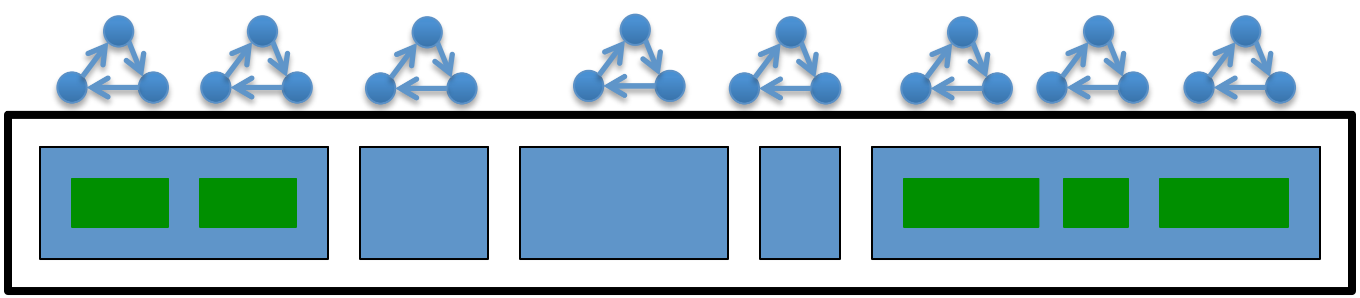


Strategy and Tactics

- How are we going to solve the problem?
  - Typically there is more than one way
  - Can be described in a couple sentences
  - Use one strategy per problem
- A strategy is implemented with *tactics*
  - Tasks
  - Ideas
  - Concepts
- Each part of the strategy is implemented with one or more tactics
  - Tactics may be composed of multiple simpler tactics

Program Planning

- For each problem formulate a strategy
  - Convince yourself that you can implement it
  - Identify the tactics you will need
- For each tactic
  - Design a state transition diagram
  - Design corresponding part of the program
- Put the parts together



Debugging

- Fact: Most programs have bugs
  - Design flaws
  - Typos
  - Bad assumptions
- Fact: Bugs cause programs to misbehave
  - Crash
  - Have incorrect behaviour
  - Corrupt data
  - Can cause loss of life, limb, and property
- Fact: Buggy programs must be debugged (fixed)

The When and the How

- We care about
  - *When* the bug manifests?
  - *How* the bug manifests?
- Because
  - Programs are large and complicated
  - Want to restrict our bug search to part of the program
- Idea: Determine the first instance of program misbehaviour

Manifestation, Location, Location

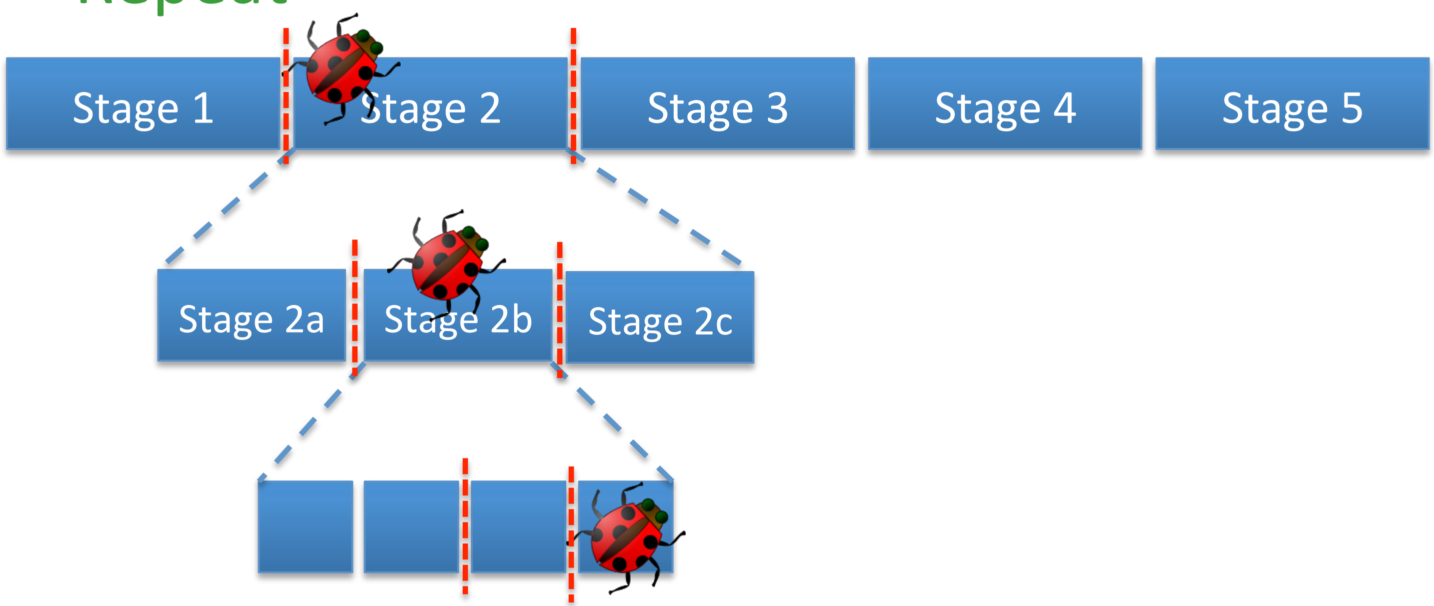
- Idea:
  - Bugs manifest in program misbehaviour
  - Misbehaviour corresponds to a program location
  - Need to match the manifestation to the location
- To do:
  - Identify the bug manifestation
    - How do we know that something is wrong?
  - Identify the manifestation location
    - Where in the code does this something occur?
- We have two options:
  - Stare the code and guess at where the bug is
  - Use a mechanical procedure to narrow our search
- Idea: “Print” to the screen when program reaches a given location

The "printf" Method

- We have two options:
  - Visually match code to execution (ok for small programs)
  - Use a mechanical procedure to narrow our search
- Goal:
  - Need to determine when we have reached specific locations in our program
  - Want the program to let us know when it has reached a specific location
- Idea:
  - Perform output when specific locations are reached
  - I.e., Turn on LEDs when our program reaches a set location

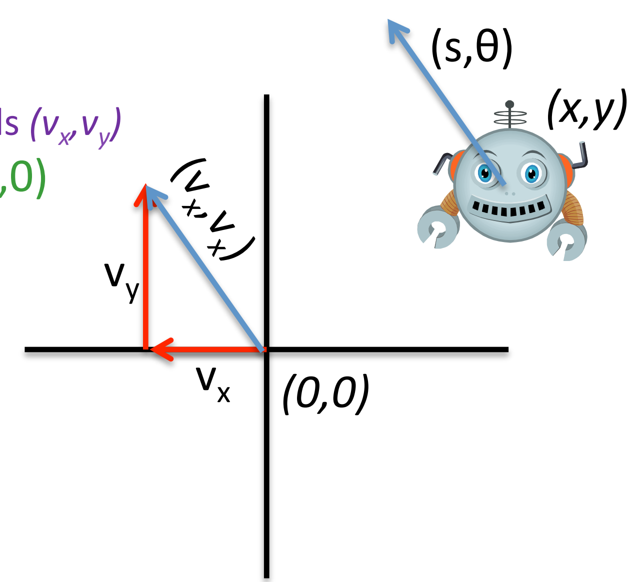
Finding the Bug

- Use divide-and-conquer approach
  - Divide program into stages
  - Narrow location of bug
  - Repeat



Odometry

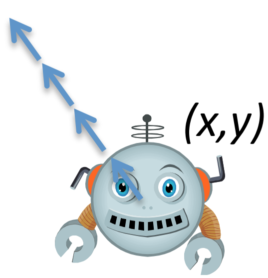
- For many tasks a robot needs to know its
  - Position: physical location (x,y) in the environment
  - Orientation: direction it is facing
- *Odometry* is the use of available sensors to estimate the robot's *current position and orientation*
- At any instant has robot has a
  - Location and orientation
    - Specified by coordinates (x,y) and direction ϕ
  - Velocity
    - Specified by speed *s* and direction θ
    - Specified by horizontal and vertical speeds (v_x, v_y)
  - Coordinates are relative to an origin (0,0)
    - Fixed location in the world
- Typically assume that the robot
  - Knows where it starts or
  - Can determine its starting location



Implementing Odometry

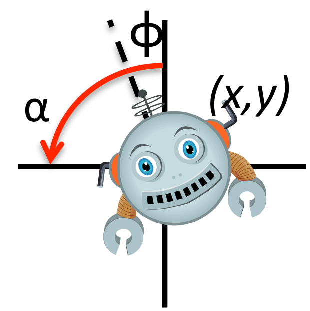
Linear Motion

- Obs: The velocity vector represents distance per unit time, e.g., (cm/s)
- Idea: Update position every second by adding velocity to position
  - new position = old position + velocity
- If velocity is represented by (s,θ)
  - $x' = x + s \times \sin(\theta)$
  - $y' = y + s \times \cos(\theta)$
- If velocity is represented by (v_x,v_y)
  - $x' = x + v_x$
  - $y' = y + v_y$
- Which is simpler?



Angular Motion

- Obs: Robots sometimes need to turn
- Assumption: Robot will turn on the spot
  - Orientation ϕ will change
  - Position (x,y) does not change
  - Angular velocity α (deg/s) is does not change
- Idea: Update orientation every second
  - new orient. = old orient. + angular velocity × time
  - $\phi' = \phi + (\alpha \times t)$
- How do we determine (v_x,v_y)?
- Observations: We know the velocity (s,θ)
  - Speed s is based on motor power
  - Direction θ is equal to the orientation ϕ
- Hence
  - $v_x = s \times \sin(\theta)$
  - $v_y = s \times \cos(\theta)$

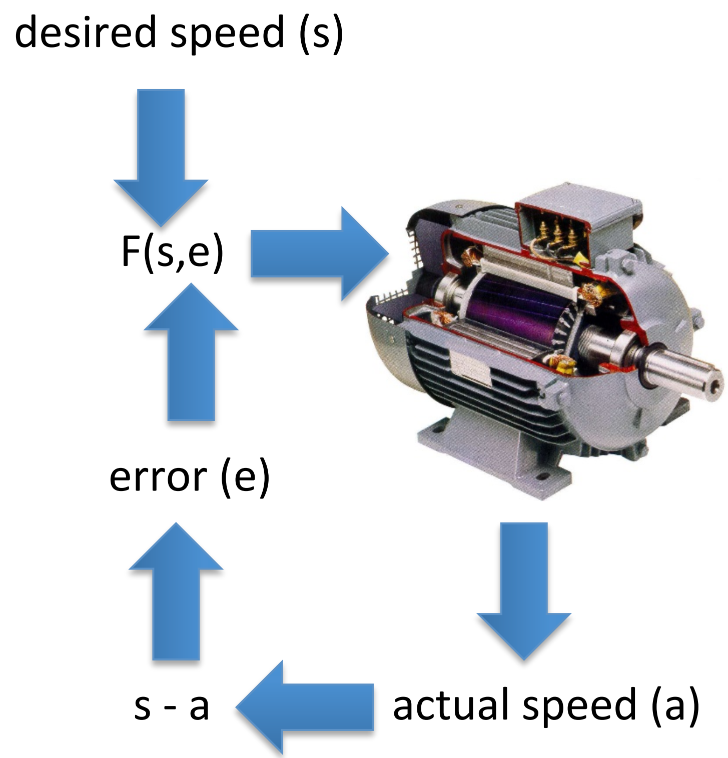


Errors in Odometry

- We know
  - The initial position and orientation
  - The speed of the motors and the robot
- Problem: Errors are introduced into the odometry computations
  - Speed is not constant
  - Motion is not straight
- What could go wrong?
  - Tires don't fully grip
  - Tires are not identical
  - Motors are slightly different
  - Battery is not fully charged
  - Speed sensors have variability
  - Motors engage at different times
  - Robot may bump into objects
- Idea: Use additional sensors to correct for errors
  - Rotation sensors
  - Motion sensors
  - Accelerometers and Gyroscopes
  - Compass
  - Rangefinders (infrared, ultrasonic, or laser)
- Challenges
  - Sensors are imperfect
  - Extracting information from environment is hard
  - Extracted information is incomplete

Rotation Sensors and the Control Loop

- Idea: Many motors have built in rotation (speed) sensors
  - Motor's *actual speed* can deviate from *desired speed*
  - *Actual speed* can be adjusted to match *desired speed*
  - A rotation sensor measures the motor's *actual speed* to adjust motor's speed as needed
- Idea: We use rotation sensors implicitly
  - Robot's motors have a built in control loop
  - We set the desired speed of the motors
  - Assume that the motors run at the desired speed
- What about using other sensors?



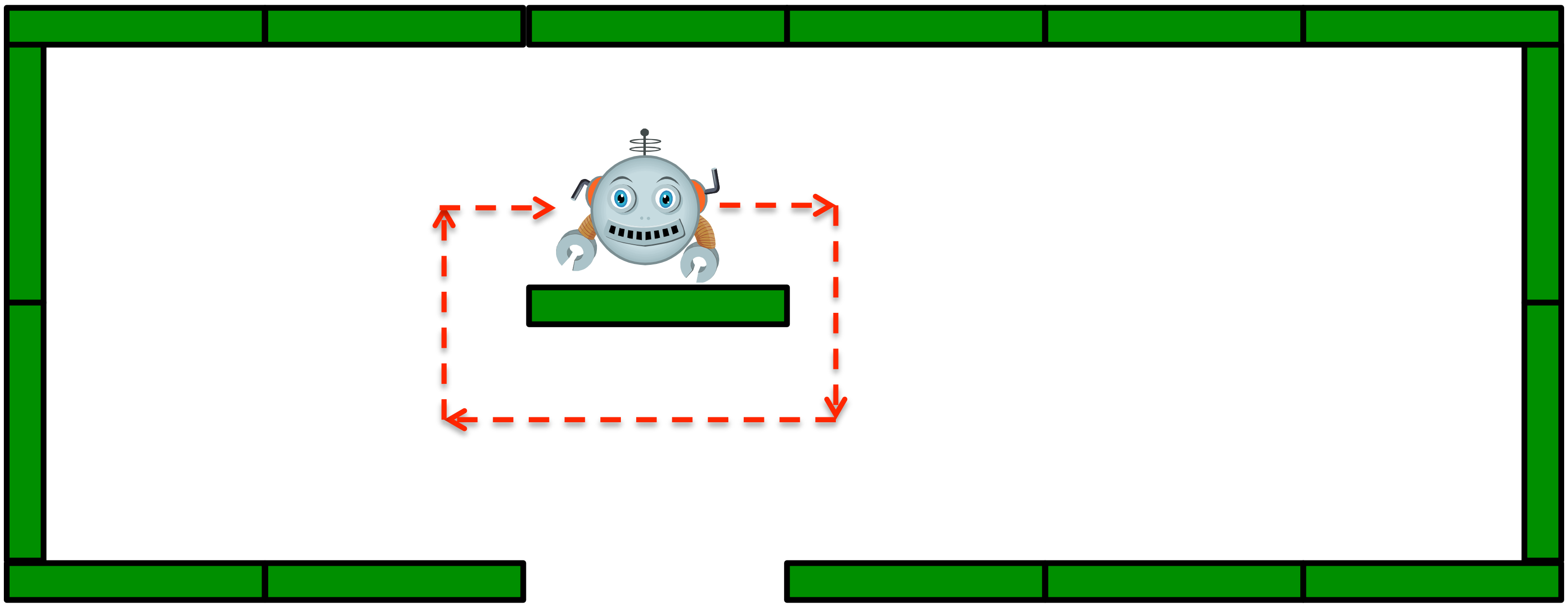
Control Loop

Visual Odometry

- Idea: Use landmarks to gauge position and speed
- Approach 1: Optical Flow based
  - Compute velocity using consecutive camera images
- Approach 2: Landmark (map) based
  - Compute location by matching known landmarks in camera images

Introduction to Search

- One of the most common tasks in robotics is to map (explore) a given environment
  - Robot must know where it is and where it was
  - This includes searching (avoid searching same place twice)
- Example: Can the exit be found without location tracking?

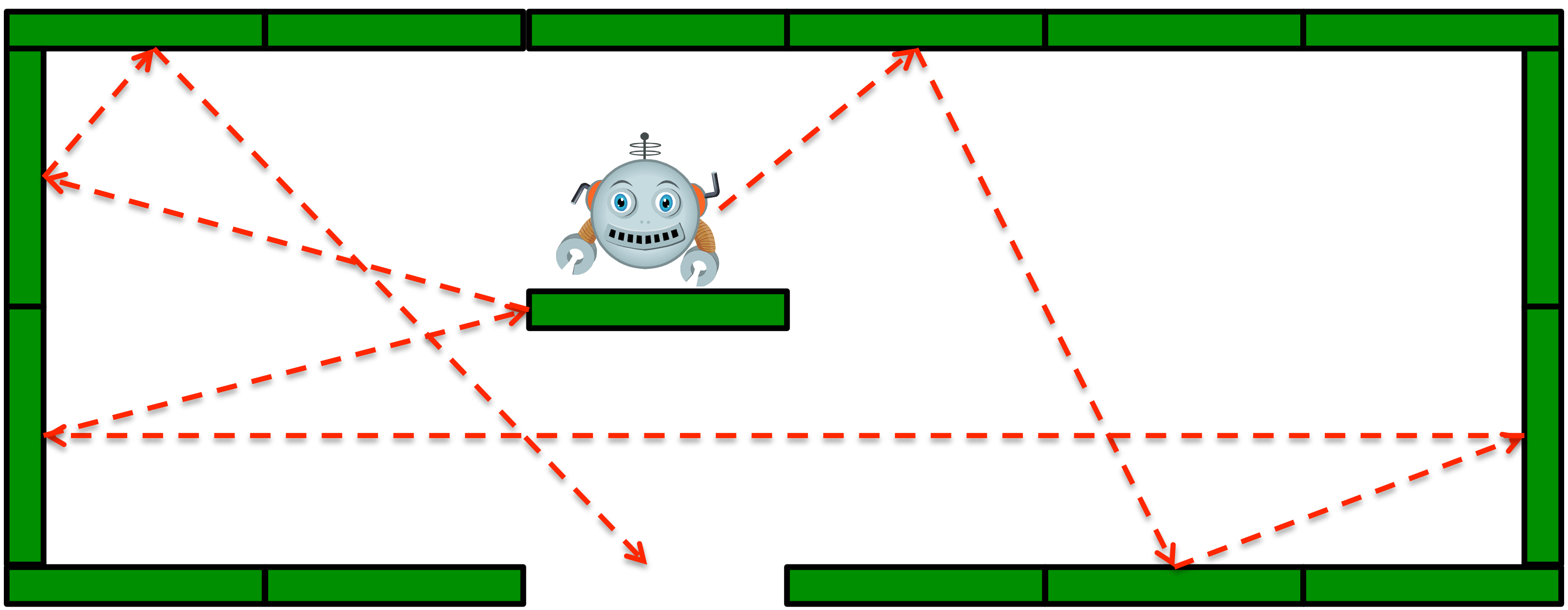


Random Search

- Algorithm:

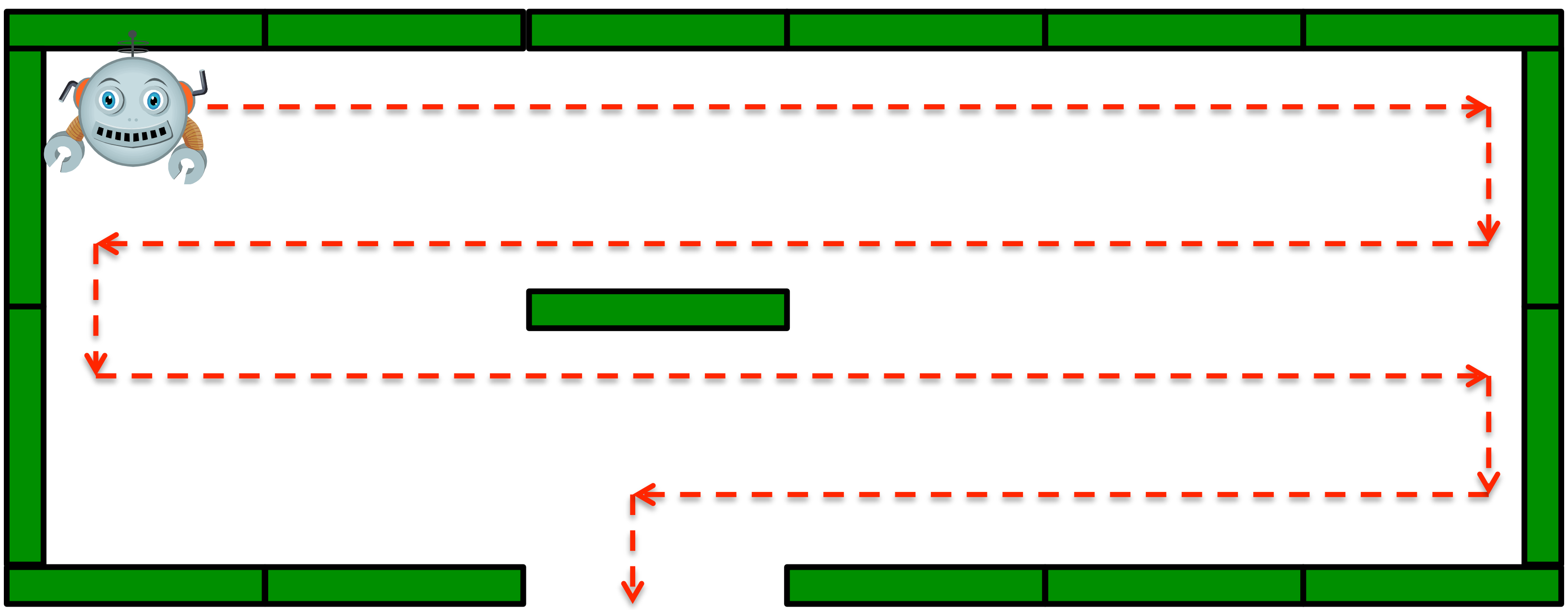
  Loop:
  - Move in a straight line
  - Turn random amount when obstacle encountered

- Reasoning:
  - Robot selects random direction regularly
  - Robot is given sufficient time
  - Robot should eventually visit every location in area



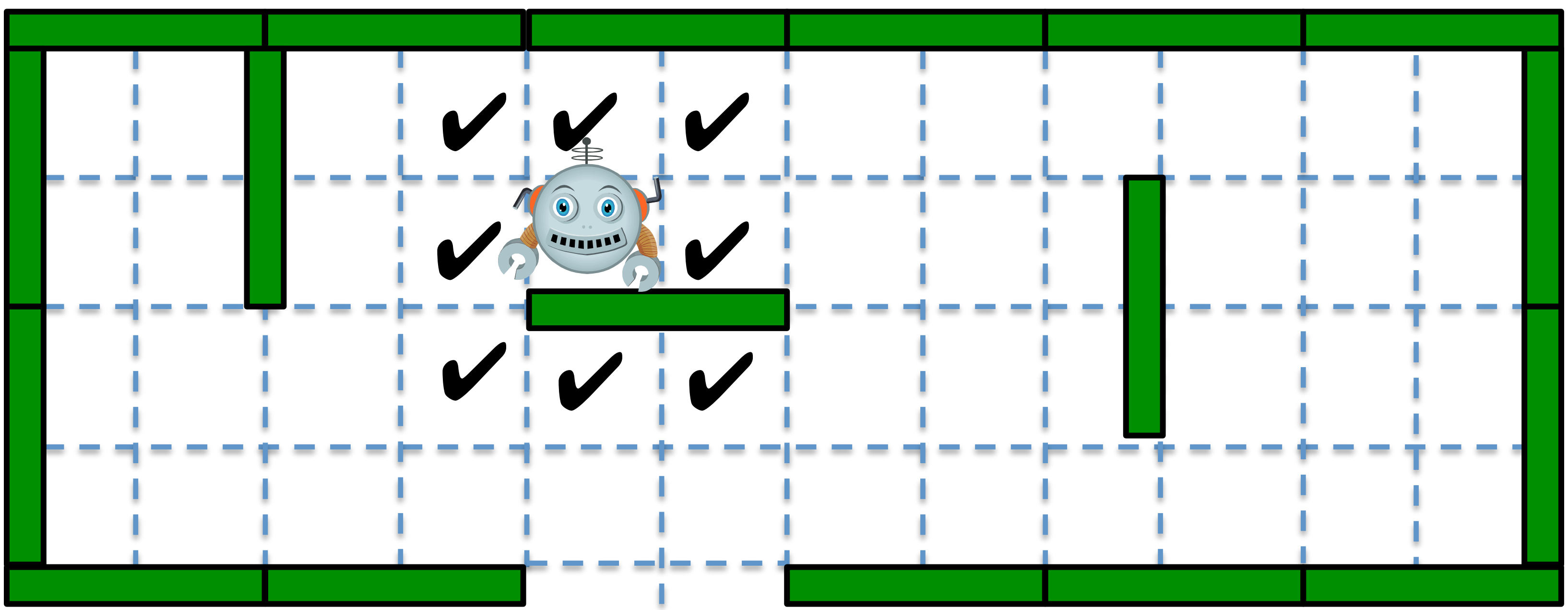
Pattern Based Search

- Algorithm:
  - Move to one corner
  - Sweep back and fourth until area is covered
- Reasoning:
  - Fixed pattern in a regular space will cover entire area
  - Determining where to start is relatively easy



Mark and Sweep Search

- Algorithm:
  - Represent area by a grid
  - Mark keep track of all visited sections
  - Visit nearest unvisited sections
- Reasoning:
  - Grids are easy to store
  - Easy to determine which section to visit next
  - All unvisited sections will eventually be visited



Search Comparisons

Random based Search

- Pros:
  - Easy to implement
  - Almost guaranteed to work
  - Odometry not needed
  - Works in odd shaped areas
- Cons:
  - Inefficient
  - Some locations visited multiple times
  - Can't reproduce search

Pattern based Search

- Pros:
  - Simple and easy to implement
  - Works well in empty rectangular areas
  - Very efficient (time-wise)
  - No need to remember visited locations
- Cons:
  - Requires good odometry
  - Does not work in odd shaped areas
  - Require a priori knowledge of area
  - Hard to implement if area contains obstacles

Search Comparisons

Mark and Sweep Search

- Pros:
  - Efficient
  - Works with obstacles and all areas
  - Easy to track objects in the area
  - Still relatively simple to implement
- Cons:
  - Requires good odometry
  - Uses more memory

General Discussion

- Q: What separates simple from complex searches?
- A: How the searches determines which section to visit next
- I.e.,
  - Simple searches base their decisions on simple things:
    - E.g., where is the nearest unvisited section?
  - Complex searches usually consider a number of factors in determining the next section to visit