Building a control system for a mobile robot can be very challenging

Mobile robots are very complex and involve many interacting components

Your control system must integrate these components so that your robot can achieve the desired goal
Building a control system for a mobile robot can be very challenging

Just as you must carefully design your robot chassis you must carefully design your robot control system

• How will you debug and test your robot?
• What are the performance requirements?
• Can you easily improve aspects of your robot?
• Can you easily integrate new functionality?
Basic primitive of a control system is a behavior.

Behaviors should be well-defined, self-contained, and independently testable.

- Turn right 90°
- Go forward until reach obstacle
- Capture a ball
- Explore playing field
Key objective is to compose behaviors so as to achieve the desired goal.
Outline

- High-level control system paradigms
  - Model-Plan-Act Approach
  - Behavioral Approach
  - Finite State Machine Approach

- Low-level control loops
  - PID controller for motor velocity
  - PID controller for robot drive system
Model-Plan-Act Approach

1. Use sensor data to create model of the world
2. Use model to form a sequence of behaviors which will achieve the desired goal
3. Execute the plan
Exploring the playing field using model-plan-act approach

Red dot is the mobile robot while the blue line is the mousehole.
Exploring the playing field using model-plan-act approach

Robot uses sensors to create local map of the world and identify unexplored areas
Exploring the playing field using model-plan-act approach

Robot moves to midpoint of unexplored boundary
Exploring the playing field using model-plan-act approach

Robot performs a second sensor scan and must align the new data with the global map.
Exploring the playing field
using model-plan-act approach

Robot continues to explore
the playing field
Exploring the playing field using model-plan-act approach

Robot must recognize when it starts to see areas which it has already explored
Finding a mousehole using model-plan-act approach

Given the global map, the goal is to find the mousehole
Finding a mousehole using model-plan-act approach

Transform world into configuration space by convolving robot with all obstacles
Finding a mousehole using model-plan-act approach

Decompose world into convex cells
Trajectory within any cell is free of obstacles
Finding a mousehole using model-plan-act approach

Connect cell edge midpoints and centroids to get graph of all possible paths
Finding a mousehole using model-plan-act approach

Use an algorithm (such as the A* algorithm) to find shortest path to goal
Finding a mousehole using model-plan-act approach

The choice of cell decomposition can greatly influence results
Advantages and disadvantages of the model-plan-act approach

• Advantages
  – Global knowledge in the model enables optimization
  – Can make provable guarantees about the plan

• Disadvantages
  – Must implement all functional units before any testing
  – Computationally intensive
  – Requires very good sensor data for accurate models
  – Models are inherently an approximation
  – Works poorly in dynamic environments
As in simple biological systems, behaviors directly couple sensors and actuators. Higher level behaviors are layered on top of lower level behaviors.
To illustrate the behavioral approach we will consider a simple mobile robot.
Layering simple behaviors can create much more complex emergent behavior.

Cruise behavior simply moves robot forward.
Layering simple behaviors can create much more complex emergent behavior.

Subsumption

- Infrared
  - Avoid
  - Cruise
- Motors

Left motor speed inversely proportional to left IR range
Right motor speed inversely proportional to right IR range
If both IR < threshold stop and turn right 120 degrees
Layering simple behaviors can create much more complex emergent behavior.

Escape behavior stops motors, backs up a few inches, and turns right 90 degrees.
Layering simple behaviors can create much more complex emergent behavior

The track ball behavior adjusts the motor differential to steer the robot towards the ball
Layering simple behaviors can create much more complex emergent behavior.

- Ball Switch → Hold Ball
- Camera → Track Ball
- Bump → Escape
- Infrared → Avoid
  - Avoid → Cruise
  - Cruise

Hold ball behavior simply closes ball gate when ball switch is depressed.
Layering simple behaviors can create much more complex emergent behavior.

The track goal behavior opens the ball gate and adjusts the motor differential to steer the robot towards the goal.
Layering simple behaviors can create much more complex emergent behavior

All behaviors are always running in parallel and an arbiter is responsible for picking which behavior can access the actuators.
Advantages and disadvantages of the behavioral approach

• Advantages
  – Incremental development is very natural
  – Modularity makes experimentation easier
  – Cleanly handles dynamic environments

• Disadvantages
  – Difficult to judge what robot will actually do
  – No performance or completeness guarantees
  – Debugging can be very difficult
Model-plan-act fuses sensor data, while behavioral fuses behaviors

Model-Plan-Act
(Fixed Plan of Behaviors)

Behavioral
(Layered Behaviors)
Model-plan-act fuses sensor data, while behavioral fuses behaviors

Model-Plan-Act
(Sensor Fusion)

Behavioral
(Behavior Fusion)
Finite State Machines offer another alternative for combining behaviors

- **Fwd** (dist) behavior moves robot straight forward a given distance
- **TurnR** (deg) behavior turns robot to the right a given number of degrees
Finite State Machines offer another alternative for combining behaviors.

Each state is just a behavior and we can easily link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Each state is just a behavior and we can easily link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Each state is just a behavior and we can easily link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Each state is just a behavior and we can easily link them together to create an open loop control system.
Finite State Machines offer another alternative for combining behaviors.

Since the Maslab playing field is unknown, open loop control systems have no hope of success!
Finite State Machines offer another alternative for combining behaviors.

Closed loop finite state machines use sensor data as feedback to make state transitions.
Finite State Machines offer another alternative for combining behaviors

Closed loop finite state machines use sensor data as feedback to make state transitions
Finite State Machines offer another alternative for combining behaviors

Closed loop finite state machines use sensor data as feedback to make state transitions.
Finite State Machines offer another alternative for combining behaviors.

Closed loop finite state machines use sensor data as feedback to make state transitions.
Finite State Machines offer another alternative for combining behaviors.

Closed loop finite state machines use sensor data as feedback to make state transitions.
Implementing a FSM in Java

```
// State transitions
switch ( state ) {
    case States.Fwd_1 :
        if ( distanceToObstacle() < 2 )
            state = TurnR_45;
        break;

    case States.TurnR_45 :
        if ( distanceToObstacle() >= 2 )
            state = Fwd_1;
        break;
}

// State outputs
switch ( state ) {
    case States.Fwd_1 :
        moveForward(1); break;

    case States.TurnR_45 :
        turnRight(45); break;
}
```
Implementing a FSM in Java

- Implement behaviors as parameterized functions
- First switch statement handles state transitions
- Second switch statement executes behaviors associated with each state
- Use enums for state variables

```java
// State transitions
switch (state) {
    case States.Fwd_1:
        if (distanceToObstacle() < 2)
            state = TurnR_45;
        break;
    case States.TurnR_45:
        if (distanceToObstacle() >= 2)
            state = Fwd_1;
        break;
}

// State outputs
switch (state) {
    case States.Fwd_1:
        moveForward(1); break;
    case States.TurnR_45:
        turnRight(45); break;
}
```
Finite State Machines offer another alternative for combining behaviors

Can also fold closed loop feedback into the behaviors themselves
Simple finite state machine to locate red balls

- Wander (20sec)
- Scan 360
- Fwd (1ft)
- Align Ball
- TurnR
- Stop

Transitions:
- No Balls from Wander to Scan 360
- Found Ball from Scan 360 to TurnR
- Ball < 1ft from Align Ball to Stop
- Ball > 1ft from Align Ball to Fwd
- Lost Ball from Scan 360 to Wander
- Move from Fwd (1ft) to Align Ball
Simple finite state machine
to locate red balls

Wander (20sec) ➔ Scan 360 ➔ TurnR ➔ Align Ball ➔ Fwd (1ft) ➔ Stop

- If no balls are found, it returns to wander.
- If a ball is found within 1 foot, it aligns the ball.
- If an obstacle is within 2 feet, it stops.
To debug a FSM control system verify behaviors and state transitions

What if robot has trouble correctly approaching the ball?
To debug a FSM control system verify behaviors and state transitions

Independent verify Align Ball and Fwd behaviors
Improve FSM control system by replacing a state with a better implementation

Could replace random wander with one which is biased towards unexplored regions
Improve FSM control system by replacing a state with a better implementation

What about integrating camera code into wander behavior so robot is always looking for red balls?

- Image processing is time consuming so might not check for obstacles until too late
- Not checking camera when rotating
- Wander behavior begins to become monolithic

```plaintext
ball = false
turn both motors on
while ( !timeout and !ball )
capture and process image
if ( red ball ) ball = true
read IR sensor
if ( IR < thresh )
stop motors
rotate 90 degrees
turn both motors on
endif
endwhile
```
Multi-threaded finite state machine control systems

Controller FSM

Obstacle Sensors Thread

Image Compute Thread

Short IR + Bump

Camera

Drive Motors
Multi-threaded finite state machine control systems

Controller FSM

Obstacle Sensors Thread

Image Compute Thread

Short IR + Bump

Camera

Drive Motors
Multi-threaded finite state machine control systems

Controller FSM

Drive Motors

Short IR + Bump

Obstacle Sensors Thread

Image Compute Thread

Stalk Sensors

Sensor Stalk Thread

Stalk Servo
Multi-threaded finite state machine control systems

- Controller FSM
  - Drive Motors
  - Short IR + Bump
  - Camera
  - Stalk Sensors
    - Sensor Stalk Thread
    - Image Compute Thread
    - Obstacle Sensors Thread
  - Mapping Thread
    - Stalk Servo
FSMs in Maslab

Finite state machines can combine the model-plan-act and behavioral approaches and are a good starting point for your Maslab robotic control system.
Outline

• High-level control system paradigms
  – Model-Plan-Act Approach
  – Behavioral Approach
  – Finite State Machine Approach

• Low-level control loops
  – PID controller for motor velocity
  – PID controller for robot drive system
Problem: How do we set a motor to a given velocity?

Open Loop Controller

- Use trial and error to create some kind of relationship between velocity and voltage
- Changing supply voltage or drive surface could result in incorrect velocity

Desired Velocity → Velocity To Volts → Motor → Actual Velocity
Problem: How do we set a motor to a given velocity?

Closed Loop Controller

- Feedback is used to adjust the voltage sent to the motor so that the actual velocity equals the desired velocity.
- Can use an optical encoder to measure actual velocity.
Step response with no controller

- Naive velocity to volts
- Model motor with several differential equations
- Slow rise time
- Stead-state offset
Step response with proportional controller

\[ X = V_{des} + K_P \cdot (V_{des} - V_{act}) \]

- Big error big = big adj
- Faster rise time
- Overshoot
- Stead-state offset
Step response with proportional-derivative controller

\[ X = V_{des} + K_p e(t) - K_D \frac{de(t)}{dt} \]

- When approaching desired velocity quickly, \( \frac{de(t)}{dt} \) term counteracts proportional term slowing adjustment
- Faster rise time
- Reduces overshoot
Step response with proportional-integral controller

\[ X = V_{des} + K_p e(t) - K_I \int e(t) \, dt \]

- Integral term eliminates accumulated error
- Increases overshoot
Step response with PID controller

\[
X = V_{des} + K_P e(t) \\
\quad + K_I \int e(t) \, dt \\
\quad - K_D \frac{de(t)}{dt}
\]

Controller

Motor

Desired Velocity ($V_{des}$)

Adjusted Voltage ($X$)

Actual Velocity ($V_{act}$)

Time (sec)

Velocity

0 0.2 0.4 0.6 0.8 1 1.2

0 1 2 3
Choosing and tuning a controller

- Use the simplest controller which achieves the desired result
- Tuning PID constants is very tricky, especially for integral constants
- Consult the literature for more controller tips and techniques
Problem: How do we make our robots go in a nice straight line?

Model differential drive with slight motor mismatch.

With an open loop controller, setting motors to same velocity results in a less than straight trajectory.
Problem: How do we make our robots go in a nice straight line?

With an independent PID controller for each motor, setting motors to same velocity results in a straight trajectory but not necessarily straight ahead!
Problem: How do we make our robots go in a nice straight line?

- Need to couple drive motors
  - Use low-level PID controllers to set motor velocity and a high-level PID controller to couple the motors
  - Use one high-level PID controller which uses odometry or even image processing to estimate error
Problem: How do we make our robots go in a nice straight line?

Need to couple drive motors

- Use low-level PID controllers to set motor velocity and a high-level PID controller to couple the motors

- Use one high-level PID controller which uses odometry or even image processing to estimate error
Take Away Points

- Integrating **feedback** into your control system “closes the loop” and is essential for creating robust robots

- Simple **finite state machines** make a solid starting point for your Maslab control systems

- Spend time this weekend **designing behaviors** and deciding how you will **integrate** these behaviors to create your control system