Principles of Robot Autonomy I

Finite state machines
Logistics

• It’s the final (project) stretch!
  • All sections are open office hours for project discussion with TAs
    Monday: 5:30 – 7:30pm (virtual) rabrown1
    Tuesday: 4:30 – 6:30pm (in-person) lewt
    Wednesday: 10am – 12pm (in-person) somrita
    Wednesday: 12pm – 2pm (in-person) schneids
    Wednesday: 5 – 7pm (in-person) rabrown1
    Thursday: 11:45am – 1:45pm (in-person) somrita
    Thursday: 4:30 – 6:30pm (virtual) lewt
    Friday: 9:45am – 11:45am (in-person) rdyro
    Friday: 12 – 2pm (in-person) schneids

• Final project demos: Wednesday, December 8th, 8:30 – 11:30am
• Simulation server should be more stable now, but perhaps see “Running ROS locally” (EdStem post)
The see-think-act cycle

Knowledge

Localization Map Building
- environmental model
- local map

position global map

Decision making Motion planning
- trajectory
- actuator commands
- Actuation

Mission goals

Information extraction
- raw data
- Sensing

Real world environment

See-think-act

Knowledge

Mission goals

Localization Map Building

Information extraction

Decision making Motion planning

Trajectory execution

Actuation

Today’s lecture

• **Aim**
  - Introduce and formalize the concept of Finite State Machines (FSMs)
  - Discuss their relevance, strengths and limitations
  - Introduce tools to allow you to use them effectively

• **Readings**
  - Chapter 4 of Leslie Kaelbling, Jacob White, Harold Abelson, Dennis Freeman, Tomás Lozano-Pérez, and Isaac Chuang. *6.01SC Introduction to Electrical Engineering and Computer Science I*. Spring 2011. Massachusetts Institute of Technology: MIT OpenCourseWare.
Motivation
Finite State Machines

Definition: A computational model for systems whose output depends on the entire history of their inputs.

*A finite state machine is a modeling framework, NOT an algorithm (similar to Markov decision processes, probability densities, factor graphs etc.)*
Finite State Machines in practice

• In practice, used in many different ways
  • Synthetically (specifies a program)
    • E.g. a product manager and an engineer specifies how an ATM machine should “behave” before starting its implementation
  • Analytically (describe the behavior of a combination of systems)
    • E.g. two self-driving cars could be modeled as FSMs. An engineer could try to see if they might end up stuck in some infinite loop at an intersection
  • Predictively (to predict interaction with an environment)
    • A self-driving car could have an internal model of a pedestrian as an FSM and use it to figure out how it should behave around it
Why are we teaching FSMs?

• For the practitioner: designing the extremely complex state machines required to fly drones, drive self-driving cars or operate warehouse robots is still one of the most time-consuming/difficult tasks faced by companies…

• How do we handle the failure of a combination of sensors gracefully?
• How do we negotiate an intersection?
• How do I get my turtlebot to start backtracking after a collision?
Why are we teaching FSMs?

• For the researcher: It’s a fundamental building block of how we understand computation, and still relevant to research today...

Mathematical definition

• Sets:
  • A set of states $S$
  • A set of inputs $I$, called the input vocabulary
  • A set of outputs $O$, called the output vocabulary

• Maps:
  • Next-state function that maps input and the state to the next state $n(i_t, s_t) \rightarrow s_{t+1}$
  • Output function $o(i_t, s_t) \rightarrow o_t$

• An initial state $s_0$
Graphical representation

• Given the sets \((S, I, O)\), it is common to express the maps \((n, o)\) by using a graph

\[ S: \{s_0, s_1, s_2\} \]
\[ I: \{i_0, i_1, i_2\} \]
\[ O: \{o_0, o_1\} \]
Graphical representation

The transition (next-state) map is represented by arrows between states, with their associated input alongside it.
Graphical representation

The output map is written alongside each transition
Example: parking gate control

The gate can be in one of three positions: ‘top’, ‘middle’ or ‘bottom’
A sensor tells the gate if a car is waiting in front of it
A sensor tells the gate if a car has just passed through it
The gate can take the following actions: raise the gate, lower the gate, no operation (nop).

We want the following behavior:
• If a car wants to come through, need to raise the arm to ‘top’ position
• The gate has to stay there until the car has driven though the gate
• The gate has to go back down after the car has gone through
Example: parking gate control


• Output: ‘raise’, ‘lower’, ‘nop’
Example: parking gate control

- Transitions
Example: parentheses balancing

• We want to design an automata that can read a string of text of any length and say whether or not the parentheses in the string are balanced or not
  • Balanced: ”1 + ( 2 + 3 – ( 4 * 5 ) )”
  • Not balanced: “1 + (2 + 3 – 4 * 5 ))”

• “… a string of text of any length…”

• A robot that can accomplish such a task would need an infinite number of states… and cannot therefore be represented by a finite state machine
FSM in the bigger picture of computation

• In terms of computational power, (deterministic) finite state machines are actually somewhat low on the totem pole of automata… with Turing Machines somewhere close to the top.
Architecture

• The architecture of finite state machines can become quite complex
• Additional states can generate an exponential number of transitions
• Strategies to keep the architecture tractable:
  1. Reduction of redundant states
  2. Hierarchical finite state machines
  3. Composition using common patterns
Finite State Machine optimization

• Algorithms exist to identify and combine states that have equivalent behavior

• Equivalent states:
  • Same output
  • For all input combinations, state transition to same or equivalent states

• Sketch of polynomial time algorithm:
  • Place all states in one set
  • Initially partition set based on output behavior
  • Successively partition resulting subsets based on next state transitions
  • Repeat until no further partitioning
Finite State Machine optimization

Sequence detector for 010 or 110

\[
( S_0 \ S_1 \ S_2 \ S_3 \ S_4 \ S_5 \ S_6 )
\]

\[
( S_0 \ S_1 \ S_2 \ S_3 \ S_5 ) \ ( S_4 \ S_6 )
\]

\[
( S_0 \ S_3 \ S_5 ) \ ( S_1 \ S_2 ) \ ( S_4 \ S_6 )
\]

\[
( S_0 ) \ ( S_3 \ S_5 ) \ ( S_1 \ S_2 ) \ ( S_4 \ S_6 )
\]

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X=0</td>
<td>X=1</td>
</tr>
<tr>
<td>Reset 0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>S6</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>S0</td>
</tr>
</tbody>
</table>
Hierarchical Finite State Machines

• Some states might not be equivalent, but it might still be beneficial to group closely related ones together

• This leads to the following two concepts:
  • Super-states (groups of states)
  • Generalized transitions (transitions between super-states)
Composition

• Cascade
  • Requirement: output vocabulary of m1 must match input vocabulary of m2
  • Resulting state: concatenation of states
  • Resulting input: input of m1
  • Resulting output: output of m2

\[ \text{Cascade}(m_1, m_2) \]
Composition

• Parallel
  • Requirement: Input vocabularies must be the same
  • Resulting state: concatenation of states
  • Resulting input: same as input vocabulary of component machines
  • Resulting output: concatenation of outputs
Composition

• Feedback
  • Requirement: Input and output vocabularies must be the same
  • Resulting state: same
  • Resulting input: partial input
  • Resulting output: same
Implementation

• Aim of this section
  • Understand that you do NOT have to use anything in particular in order to implement a FSM
  • Understand that there are however common ways to implement finite state machines
  • Grow awareness of tools available to help you build and analyze them
Implementation

• A common strategy is to exploit Object Oriented Programming (OOP) and implement a class that corresponds to your finite state machine

• The class keeps track of which state the FSM is in (e.g. in a variable)

• A loop repeats at some fixed rate

• Each loop, the FSM input is read (e.g. sensors, clock)

• The current state is executed (as an if/else block)
  • Actions that need to be taken (e.g. set actuator setpoints)
  • Transition to next state (e.g. state variable updated)
Example implementation

- PX4: in many ways the leading open source flight software for drones
Example implementation

• Commander.cpp

```cpp
while (!should_exit()) {
    bool nav_state_changed = set_nav_state(&status,
```

• state_machine_helper.cpp

```cpp
switch (internal_state->main_state) {
    case commander_state_s::MAIN_STATE_ACRO:
        status->nav_state = vehicle_status_s::NAVIGATION_STATE_ACRO;
        break;
```
Example implementation

• 14 open issues that involve a “state machine”...
Example implementation

• Your very own navigator.py!

```python
# STATE MACHINE LOGIC
# some transitions handled by callbacks
if self.mode == Mode.IDLE:
    pass
elif self.mode == Mode.ALIGN:
    if self.aligned():
        self.current_plan_start_time = rospy.get_rostime()
        self.switch_mode(Mode.TRACK)
elif self.mode == Mode.TRACK:
    if self.near_goal():
        self.switch_mode(Mode.PARK)
    elif not self.close_to_plan_start():
        rospy.loginfo("replanning because far from start")
        self.replan()
    elif (rospy.get_rostime() - self.current_plan_start_time).to_sec() > self.current_plan_duration:
        rospy.loginfo("replanning because out of time")
        self.replan()  # we aren’t near the goal but we thought we should have been, so replan
elif self.mode == Mode.PARK:
    if self.at_goal():
        # forget about goal:
        self.x_g = None
        self.y_g = None
        self.theta_g = None
        self.switch_mode(Mode.IDLE)

    self.publish_control()
rate.sleep()
```
ROS State Machines: SMACH

• A ROS tool that allows you to synthesize FSMs more easily
• Provides visualization tools
• Support hierarchical state machines
• Enables easy composition
• See http://wiki.ros.org/smach/Tutorials/Getting%20Started
SMACH: Basic Syntax

• Two main components:
  • SMACH State
  • SMACH Container (e.g. FSM)
SMACH: Basic Syntax

• SMACH State
  • The basic state abstraction. Corresponds 1:1 with the FSM states described earlier
  • Inherit from `smach.State` and must implement two functions:
    • `__init__`
    • `execute`
  • `execute` should return ‘outcomes’
SMACH: Basic Syntax

class Foo(smach.State):
    def __init__(self, outcomes=['outcome1', 'outcome2']):
        # Your state initialization goes here

    def execute(self, userdata):
        # Your state execution goes here
        if xxxx:
            return 'outcome1'
        else:
            return 'outcome2'
SMACH: Basic Syntax

• SMACH Container
  • Roughly corresponds to the idea of a finite state machine, with variations.
  • You are most likely to use the container `smach.StateMachine`
  • States can be added to containers
  • Containers can be composed
SMACH: Basic Syntax

```python
sm = smach.StateMachine(outcomes=['outcome4','outcome5'])
with sm:
    smach.StateMachine.add('FOO', Foo(),
                            transitions={'outcome1':'BAR',
                                         'outcome2':'outcome4'})
    smach.StateMachine.add('BAR', Bar(),
                            transitions={'outcome2':'FOO'})
```
SMACH: Basic Example
# define state Foo

class Foo(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['outcome1', 'outcome2'])
        self.counter = 0

    def execute(self, userdata):
        rospy.loginfo('Executing state FOO')
        if self.counter < 3:
            self.counter += 1
            return 'outcome1'
        else:
            return 'outcome2'
# define state Bar

class Bar(smach.State):
    def __init__(self):
        smack.State.__init__(self, outcomes=['outcome2'])

    def execute(self, userdata):
        rospy.logininfo('Executing state BAR')
        return 'outcome2'
SMACH: Basic Example

```python
# main
def main():
    rospy.init_node('smach_example_state_machine')

    # Create a SMACH state machine
    sm = smach.StateMachine(outcomes=['outcome4', 'outcome5'])

    # Open the container
    with sm:
        # Add states to the container
        smach.StateMachine.add('FOO', Foo(),
                                transitions={'outcome1':'BAR',
                                             'outcome2':'outcome4'})
        smach.StateMachine.add('BAR', Bar(),
                                transitions={'outcome2':'FOO'})

    # Execute SMACH plan
    outcome = sm.execute()
```
SMACH: Composition

• The composition operations described earlier (cascade, parallel, feedback) are also possible in SMACH

Cascade -> `smach.Sequence`
Parallel -> `smach.Concurrence`
Feedback -> `smach.Iterator`
SMACH: Visualization

• The package `smach_visualizer` allows you to easily inspect and monitor your state machine.
DEMO: AA274 Navigator using SMACH
Thanks for a great quarter!