Lecture 3: Advanced Techniques

- Happyboard status
- Balls
- Filters: low-pass, Kalman
- Control: feedback, PID
- FSMs
- Threads
- Event loops
- Design tips
Happyboard Status

- Solder it, then give it back
- You’ll get it back (for keeps) when you hand in Assignment 2
  - Before midnight Monday
  - Don’t forget your writeup
Balls

- You’ll get standard-size red and yellow balls after the workshop
- Along with sensors
- Skunk balls will be 3” and red
Other general announcements

• Glue guns, Heat guns, Soldering irons
  – Please take care of them
  – Do not take away from stations
  – Use solder fans when working at a station

• Limited Supplies
  – Share tools; don’t steal them
  – Don’t take unreasonable amounts of sticky tape, wire, etc.

• Don’t wander into the office
  – If you need something, ask a staff member (should be nearby)
Rules Clarifications

• We are here to ensure standardization. The better your questions, the more quickly we can set reasonable standards of competition.

• Check the website for more
The AI: How to Code a Robot

• Not simple
• Programming language is easy; programming style is difficult, especially with a team (any 6.170 alums?)
• Some effective patterns
  – Filters
  – Controllers
  – Finite State Machines
  – Event Loop
Filters
Low-pass Filter

- Eliminates high-frequency components from noisy sensors
- Used for
  - Debouncing
  - Ignoring spurious inputs
Debouncing

```c
int read_debounced() {
    int ctr = 0;
    for (int i = 0; i < 10; i++) {
        if (bump[i] == 1) {
            ctr++;
        }
    }
    if (ctr > 8) {
        return 1;
    } else {
        return 0;
    }
}
```
Kalman Filter

- Adaptive LPF
- Determines state given noisy measurements
- Optimal (awesome)
- Linear (simple)
- Estimator (maybe not perfect)
Definitions

- $x_k$ – state vector at time $k$
- $F_k$ – state transition model
- $u_k$ – control vector
- $B_k$ – control input model
- $w_k$ – process noise $\sim \mathcal{N}(0, Q_k)$
- $z_k$ – measurement
- $v_k$ – measurement noise $\sim \mathcal{N}(0, R_k)$
Predict

• Predicted state
\[ \hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_{k-1} \]

• Predicted estimate error
\[ P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_{k-1} \]
Update

- Measurement error
  \[ \tilde{y}_k = z_k - H_k \hat{x}_{k|k-1} \]
- Error covariance
  \[ S_k = H_k P_{k|k-1} H_k^T + R_k \]
- Optimal Kalman gain
  \[ K_k = P_{k|k-1} H_k^T S_k^{-1} \]
- Updated state estimate
  \[ \hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k \]
- Updated estimate error (covariance)
  \[ P_{k|k} = (I - K_k H_k) P_{k|k-1} \]
Example: Model

- Position and velocity
  \[ x_k = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \]

- From k-1 to k...
  \[ x_k = F x_{k-1} + G a_k \]

- Where
  \[ F = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \]
  \[ G = \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} \]
Example: Measurement

- Noisy measurement
  \[ z_k = Hx_k + v_k \]
- Of position
  \[ H = \begin{bmatrix} 1 & 0 \end{bmatrix} \]
Example: Initialization

- Initial state estimate
  \[ \hat{x}_{0|0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \]

- Perfect, so no initial estimate error
  \[ P_{0|0} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \]
Example: Watch It Go...

- Predicted state is exactly starting state
- Predicted estimate error is exactly measurement variance
- Update, predict again, and so on
Control
PID Loop

• What is it?
  – Simple, effective feedback controller
  – Tunable gains for proportional, integral, and derivative components

• Why?
  – Used for precise movement
  – Accurately snap to turns, stop quickly on target
  – Implemented in JoyOS!
Closed-loop Controller
PID Loop Block Diagram

\[
\begin{align*}
\text{P} & : K_p e(t) \\
\text{I} & : K_i \int_0^t e(\tau) d\tau \\
\text{D} & : K_d \frac{de(t)}{dt}
\end{align*}
\]
Finite State Machines
High-Level Behavior: FSM

- What is a finite state machine (FSM)?
  - Defines what the robot should do at a given point in time
  - Each state has predefined outputs
  - Transitions to other states depend on inputs

- Why?
  - Effective way of thinking about your strategy
  - Define what to do for any combination of inputs
Implementing a State Machine

- Each action is a state
  - Moving forward
  - Turning
- Actuators are outputs of the FSM
- Sensor inputs determine next state
Example FSM

- Orient Robot
- Move Straight
- Collect Ball
- Release ball
- Wall Follow Forward
- Detect bin
- Detect opposing robot
- 180-degree turn
- Ball not detected
Coding an FSM

- While loops
  - Continue an action until input is received
- Multithreading
  - Processes that determine the inputs
  - Processes that determine outputs and state transitions
- Don’t do it the 6.111 way
  - Don’t need a variable to keep track of what state you’re in
  - Instead think conceptually; think before you code
FSM Issues

- **Inputs**
  - Check only those that matter at that state
  - Determine what is important

- **Storing State**
  - Make your robot smarter
    - Use the state as well as the inputs to determine action
    - Store last actions in state variables
  - Helpful if robot gets disoriented
Error Detection

• Your robot **will** mess up :(  
• How can it find out what’s wrong?  
• **Timeouts** are key
Watchdog

- Regardless of state, keep track of the last time your robot saw an input
- If it’s been more than N seconds, you’ve got a problem
Timeouts

• Detect when robot is stuck in a state
  – Probably waiting for input – bump into wall, light reading

• Force out of stuck state
  – Error correcting routines
Error Correction

- Try again, harder
- Back up, try again
- Wiggle around
- Guess what it should try next
- Skip to next part of routine
- Line following: what to do about the n/a states
  - In this case, using an FSM may help you figure out what to do
Threads
Quick Note on Threads

• What is a Thread?
  – Separate processes running at the same time
  – Allows you to multi-task
    • Motors run *and* watch if a sensor is pressed

• How does one processor run two threads?
  • Executes a process certain number of ticks (ms)
  • Processor switches from one process to another
Example

```cpp
move() {
    while (true) {
        Turn 90 degrees
        Wait until gyro angle increased by 90 degrees
        Go forward
        Wait until sensor pressed
    }
}
```

```cpp
update_gyro() {
    while (true) {
        Get angular velocity
        Adjust for offset
        Find time difference
        Angular_dist += angular_velocity*time_difference
    }
}
```
Why Was This Example Easy?

- Threads are nearly independent of each other
- One thread produces information (angle) and the other consumes it
How Threads Communicate

• Communicate through global variables
• One thread can use the global variable that another thread is changing
  – Producer-consumer model
• For more complex inter-thread communication, use semaphores to prevent “dirty” accesses, deadlock
For the Contest

• You will be using threads, even if you don’t know it
  – JoyOS updates robot angle asynchronously
Asynchronous Code
Without the Headache:
The Event Loop
Motivation

• You want to think about your code as though multiple functions are running simultaneously
  – E.g., you check for wall bumpage while also updating your gyro and checking for timeouts
  – Or, multiple functions are waiting for an input or state before they take effect
The Big While Loop

```c
while (1) {
    sense();   // read inputs
    plan();   // figure out what to do
    act();   // do it
}
```
Abstracting it Away

```
fn[0] = pSense;
fn[1] = pPlan;
while (1) {
    for (int i = 0; i < 2; i++) {
        (*fn[i])();
    }
}
```
Taking It a Step Further

threshold[0] = 90; ...
variable[0] = &angle; ...
while (1) {
    for (int i = 0; i < n; i++) {
        if (*variable[i] > threshold[i]) {
            (*event[i])();
        }
    }
}
Design Tips
Code Implementation

• Avoid complexity, especially at first
• Use functions
  – Code is then legible for everyone on your team and for us (impounding)
• Avoid dynamic memory allocation (4k max!)
Programming Methodology

• Top-down programming
  – Good for initial design
  – Overall view without details

• Bottom-up programming
  – Good for code creation
  – Allows individual testing of functions
Programming Methodology

- Figure out the functions you need ("Design")
- Implement by function—modularity is key
- Test
- Integrate into other code
- Test
- Repeat
- Test again
Testing and Debugging

- Most important part of the design
- Significant testing is necessary to do well
- Test and debug *incrementally*
Iterative Design

• Start stupidly simple
• Get that working, then add sophistication
• Repeat
  – Guarantees you’ll have something reasonable for the final competition
What’s Next

• Assignment 2
  – Due by midnight Monday!
  – Go to Workshop 1 to help complete assignment
    • If you cannot make it to a workshop session, go see an organizer to make other arrangements
  – Don’t forget about the written component
What’s Next

• Check the website religiously for updates
  – Read and bring the handouts to the workshops
  – Be sure to bring your kit to the workshops as well
Get up to lab!

Credit: Wikipedia for PID and Kalman slides