University of Toronto, Scarborough Fall CSCC85
CSCC85 Finals Notes

Chapter 1 – Embedded Systems and Sensors

1.0 Introduction to Embedded Systems

Embedded System – A system designed for a special purpose with the ability to execute software and interact with the environment in a specific way

CONTAINS:

- I/O
- Processing Unit (CPU/MCU/SOAC) – Central Processing Unit, Microprocessing Unit, System on a Chip
  - CPU (has ALU, databus, control units, etc.)
  - MCU
    - It is a CPU + embedded I/O
    - Arduino development boards, etc
  - DSP (interfaces to connect microphones) Digital Signal Processor
    - Processes Math and Signals
  - Systems on a chip
    - Qualcomm Snapdragon is a SOAC - not just a CPU
    - Everything on a motherboard goes into the same chip
    - Smaller, energy efficient, cheaper
    - Loses ability to upgrade components, have to completely replace once obsolete

- Choice depends on: **Pricing**, **Speed**, and **Codebase**

Common Issues of Embedded Systems Programming:

- Costs
- Reliability/Failsafes
- Specific hardware limitations
- Security Limitations
- FPGAs
  - Programming hardware so that it can implement logical functions

Firmware- The software that runs an embedded system

Embedded System Design tradeoffs:

Price, power, computational power

1.1 Sensors

Sensor – Thing that converts analog data to digital data. Take a physical quantity, and by some process, convert it to an electric signal. This conversion is usually imprecise, or **noisy**

Properties of Sensors:

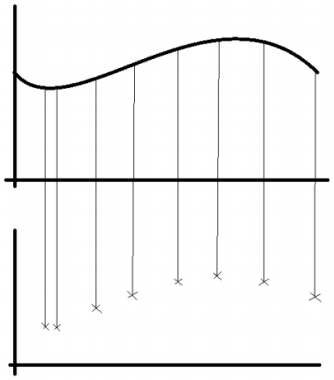
Consistency: When measuring the same property it should give the same result
Precision: How much detail the result has
Accuracy: How close the result is to the actual quantity being measured

Good Sensors: Sensitive to the quantity being measured but not to other quantities not being measured

r(x) = ax+b, where x is physical quantity and r(x) is the response. We want a scaled representation of the quantity we are measuring

r(x) = response
x = physical quantity
b = offset

We can also have stuff like r(x) = a*log(x)
We know that r(x) = x does not exist due to **NOISE.**



Major Steps for sensing:

- Measuring the signal (from physical to electric signal, conversion is noisy)
- Storing electric signal (precision and significant figures, missing data)

Zero Noise Model

Doesn't work at all for particle filtering… not realistic.

Noisy Sensor Model

R(x) = f(x) + noise

Where noise is sampling noise (conversion, sampling, quantization, measuring)

Observation – measuring changes the quantity one is measuring

Noise Filtering

Need a noise model

Uniformly distributed (random static on a channel)

Zero Mean Gaussian noise (evenly distributed lower chances of noise)

ZMG model is superior, as we have a PDF to check the zero-mean. We only need the STD Dev to measure the noise. The smaller the sigma the better.

We assume that usually, the noise is **Zero Mean Gaussian and IID (Independently Identically Distributed)**

Filtering noise out by averaging works if **x is slow-changing**

What if x changes over time? We need to filter/sample!

Capture your signal

Filter noise out (assumption about how noise behaves)

Noise changes faster than the signal

Sampling

Taking uniform data points from analog signal to reconstruct a digital representation. As above ^

Sampling Frequency

More sampling -> More storage required, limited by speed of sampling process

Less sampling -> Loss of accuracy

Critical amount of sampling needed to capture signal in digital format: **2x Max Frequency/Signal**

e.g. since human hearing is 20HZ-20kHz, we should sample at 40kHz

Aliasing

When we sample at a rate that creates a digital representation of data that does **NOT** represent the signal we are sampling. (Fake signal data!)

Chapter 2 – Localization

2.0 Localization

Localization – the process of finding out where you are on a given map when placed at a randomized location. Map contains landmarks that identify locations. Use sensors to determine surroundings and match corresponding landmarks onto the map

Assumptions

1 Map does not change and is up-to-date

2 Robot does not move randomly, only under guidance

3 Robot can estimate its motion

Approaches to Robot Localization: (Both are super noisy and generally deviate from truth as time progresses)

- **Dead Reckoning** – Calculating one's current position using previously determined location and advancing from that position using estimated speeds over elapsed time and course – tracking motion over time
- **Inertial navigation** – Using sensors to determine acceleration and rotation instead of distance travelled and determine position using that
- **Absolute measurement** – The usage of landmark and beacons to triangulate position, landmarks being specific and identifiable. Lots of demand on identifying landmarks and finding specific landmarks. Enormous effort to create

Localization in Robotics

The proper method is to use probability.

Belief -> The certainty of the robot about the world around it

Markov Localization – Sebastian Thrun

$Bel(x_k) = P(x_k \mid d_0, d_1 \ldots d_k)$

This means that the

Belief of robots state at x_k = probability of x_k position in map given data measurements $d_0 \ldots d_k$

The idea is to use recent information, such that

X_{k-1} is where the robot was a moment ago

D_k is the current measurement

A_{k-1} is what the robot just did

Initially at x_0, the robots probability is uniformly distributed all over the map

It takes two different steps:

Acting – $P(x_k \mid x_{k-1}, a_{k-1})$ Remark: Needs to know how robot moves

Sensing – $P(d_k \mid x_k)$ – agreement. Remark: $P(x_k \mid d_k)$ is maximum likelihood

Histogram Localization

Divide map into a grid and assign probabilities to each square.

Localization Steps:

Senses, updates grid probability, normalizes

Moves, shifts grid probability, normalizes

Bel = Sum(P(xk | xk-1))

Difficult – think of difficulty to transform map of Toronto into a grid map

Particle Filtering

Random particles with even distribution with x, y vectors, directional angles, and belief probabilities
Choose available action, perform on robot and all particles
Use sensor to measure surroundings, measure belief of each particle and update it
Replace all particles with particles of same set size, chosen randomly (resampling)

2.1 Particle Filtering

We use **Acting** and **Sensing** to localize a robot.
Acting – Moving, grabbing stuff, sounds
Sensing – Measure environment to gather evidence
Uses **Markov Localization:** Bel(xk | $x_{k=1}$, d_k)
Uses the last location of the robot and current measurements to determine the belief at current location
Particle Filtering uses Acting, Sensing, and Resampling to localize a robot.
Acting Step

- Robot chooses an action
- Applies same action to all particles

Movement of Particles **MUST** be **noisy** as Robot movement is noisy
Sensing Step

- Uses robot's sensors to take measurements
- Compares measurements with simulated **Ground Truth** using map data for particles
- Compares sensor readings from robot to particle
- Updates beliefs for particles depending on the difference of sensed measurements of particle and robot
- Normalizes all particle beliefs

How to compare values
Error = Sensor(Robot) – Sensor(Particle)
Plot normal distribution of error, obtain P(Error)
Take Bel(p_k) = P($Error_k$) * Bel(p_{k-1})
Multiple Values?
Find the Euclidean error, that is →

$$p(error) = \sqrt{\sum_{i}^{n}(\vec{R}_i - \vec{P}_i)^2}$$

Resampling Step
Put all particles on a line, choose between [0, 1] randomly until a particle is chosen. Add to set. Repeat until set is full again
We can choose to replace a percentage (10%) of particles before resampling with new random particles and small beliefs.
If particles continue to disagree, the robot is **unrecoverable** and must be fully reset to try to re-localize.

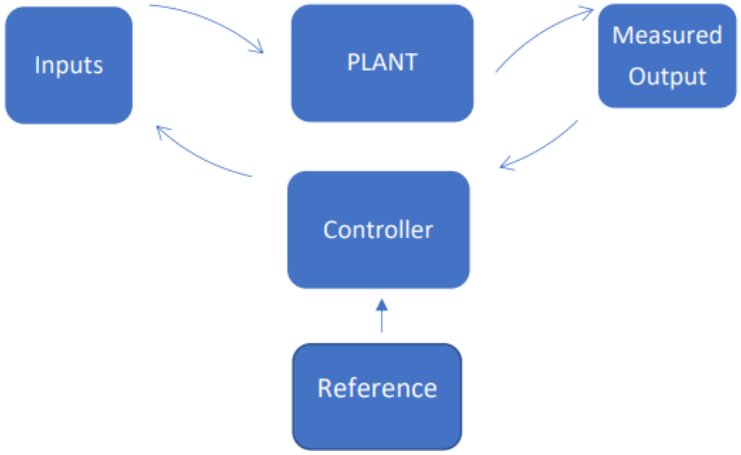
Motion Model – The way the robot moves through the environment including all mechanical models and noise

---

Chapter 3 – Control Systems

3.0 Control Systems

Dynamical Systems – Airplanes, cars, quadcopters, etc.
Control – Establish Desired Behavior
Control System – The component (mechanical/electronic + algorithm) intended to bring the system to the reference state
Must work in the presence of disturbances
Example – Cruise Control for a Car
Reference: Desired speed
Measure: Current speed
Solution: Account for the relative speed of the car and increase/decrease or accelerate/brake relative to that
Model – dx/dt (^x) = f(x, u), where:
x is state vector that describes the system
u is the input
Example – For a 2D car travelling on 1D road,
Constant speed x = [Px, Vx] (pos/velocity)
^x = [0 1, 1 0][Px, Vx]



Plant - The component that generates feedback output?
Closed Loop Feedback - Control system where controller applies an input based on output value

Modern Control Systems

Input -> [Actuator, System, Sensors] -> Output |
| |
----------Control----------

PID Controllers

Needs to measure error = reference – measured value

Uses three components – P, I, D:

Proportional Component:

(control input)

$u = k_1 e$

Integral Component

(time component, exploring how long error has existed, forces error to zero)

$u = k_1 e + k_2 \int e\, dt$

Derivative Component

(dampening and making sure not to overshoot, reduces oscillations)

$u = k_1 e + k_2 \int e\, dt + k_3\, de/dt$

3.1 Kalman Filters

Neural Net Controllers, ran millions of times to figure out how the difference between what it is supposed to do and what it does, coming closer every time. (Trained on simulator)

Also called **Model Predictive Control**

Kalman Filtering Getting a solid and reliable estimate for the state of a system

Feedback controller uses the correlation between estimated and measured to improve the estimate of needed

Works in a dynamic system

Keeps track of the state vector

Requires a model for the system that can account for bad movements

Assume that both the sensor and process are noisy, noise assumed to be Zero mean Gaussian

Model Used – State is a vector X_t that contains set of state variables. The goal is to produce values of X_{t+1}

Equation for the evolution of a system –

$X_t = F_t X_{t-1} + B_t U_t + W_t$

$F_t X_{t-1}$ is the old state and how it affects the old state (F_t matrix)

$B_t U_t$ are the new inputs

W_t are the noises

Sensor Model

$Z_t = H_t X_t + V_t$

Z_t is sensor measurement at t

X_t is the state variable

V_t is the noise

H_t is how much the state affects sensor measurements

Kalman filters are limited to linear systems

There are tricks to apply Kalman filters for non-linear systems though

---

Chapter 4 – Reliability and Robustness

4.0 Building Fault-Tolerant Systems

Case study – the Therac-25

- Medical radiation machine
- First one to be software controlled
- 2 paths of operation based on time constraints, really fast triggers start therapy before inputs are entered
- Bad error messages
- Software did not check if input was safe
- Design process was broken

Things to consider:

Overconfidence in software
Confusing reliability with safety
Complacency
Discounting of software risk
Inadequate software engineering
Software reuse

Safe vs User friendly user interfaces

Case study – the Mars rover

- Planned to last 90 days but lasted 30x longer, almost died on Day 18
- Wasn't entering sleep cycle
- Ran out of memory mounting the file system
- Stuck in a reboot loop
- Off shelf software did not delete info just marked for deletion

Mars orbiter -> Burnt on arrival, didn't account for unit conversion

Best Engineering Possible

Reliability – Probability that a system will work as intended over a given period of time (i.e. planes will work 99.9% over 5h)
MTTF – Mean time to failure, average time until failure occurs, doesn't say anything about safety, better to have a backup
Availability – The % of the time the system is up (used for services)

Model Reliability –

Coverage – If some component fails, others can be used to mask that
We want systems with good coverage
Dependency – Try to eliminate single points of failure
Components depend on something that can fail, invalidates the coverage

Combinatorial Parts Model

Can model reliability with success tree, such as reliability of P = 0.95 and C = 0.999, reliability of system that uses P and C is (0.95)*(0.999) per 5h.

AND – Multiply
OR – $(1 - (1 - REL_1) * (1 - REL_2))$

Redundancy in Computer Systems

- Need to vote (odd number of sensors, odd one's value is considered faulty)
- Need redundant hardware, computing hardware, software
- Replicate sensors (vote with majority, use median)

Redundancy in Software:

- Safe software, different teams, different companies, different languages
- Even number of components, 1 is different for redundancy
- Expected to fly plane given any 2 failures
- Failure detection is hard
  - Abrupt failure, failure can be detected right away
  - Gradual failure, results seem correct but will gradually worsen (bad!)
  - Random failure, results are correct but sometimes not (BAD!)

Consensus and voting-out reconfiguration

Technique to detect faulting computer systems so that they can be ignored and do not cause systems to failure

Fault Tolerant Control (FTC):

Passive
Robust control
Good controller
Active
On-line (learns on the fly, neutral networks)
Projection
Train or build a controller for every situation
Failure system will select a controller based on diagnosis of sensor failure
Should be big enough for every situation

4.2 Human or Computer Control

Case Study – Electronic Stability Program (ESP)

- Makes adjustments to individual brakes to control the car better than any human can
- Increased safety
- Reduced car crashes by 43% if all installed in cars

Things to consider:
How much control should we give to automated systems vs human control?

Case Study – Boeing vs Airbus
Boeing:

- Less automation
- Allows pilot to do what they want but provides feedback

Airbus:

- More automation
- Computer provides oversight (can override pilot)
- Complications arise when emergencies happen

Case Study – Air France Flight 447

- Autopilot failed, control to pilots
- Pilots should have been able to fly, but couldn't due to lack of knowledge

4.3 Finite State Control Systems

Components of AI:
Input/Perception -→ Decision Making/Planning → Control
We can use **Finite State Machines (FSMs)** for AI

- States are equivalent to behaviors
- Transitions are triggered by input, or changes in behavior

Pros:

- Efficient
- No complex computation
- Predictable
- Simulation is possible
- Can be mathematically analyzed

Cons:

- Finite amount of created states
- Cannot adapt to experience
- Predictability can be exploited
- Can result in spaghetti code

4.4 Robot Perception

Robot perception is buggy:

- We lose depth
- We blur motion
- Lose ability to make sense of image data

One solution -> Make patches, match patches, and if enough match, we can assume that it is the same picture
Localization using Landmarks is possible
In case we do not have a map, we use SLAM
Self Localization and Mapping

1. Determine Which landmarks are visible and check against an array of known landmarks for positions of landmark
2. Move robot to new location
3. Sense and measure again in step 1

4.5 Code Optimization

Pipelining – (Fetch, Decode, Execute, FDE), improves number of instructions per clock cycle (IPC) and shorter clock cycles
Branch Prediction – Determine which condition will be ran most often and try to replace to be optimal (less checks). Technique that reduces impact of conditional statements on processor pipeline
Caching – Faster than RAM, diferent levels in CPU, code and data are separated, but can reuse data and code, and especially **access pattern**

Optimization Patterns:
Local Variables - minimize local variables in functions, stored in a stack, have to be allocated then deallocated.
Parameter passing and returning value: Don't return large data structures by value, pass DS by reference
Caching - Think locality and order-of-access (multi dimensional arrays)
Strength Reduction – Less usage of multiplication, more addition/subtraction, etc.

Optimizing Code Flow:
Sequential Code – Sequential from start to end, no branching or cases
Minimize Branching/Function Calls
Inline shorter functions

Simplify if statements
Use **Boolean Algebra** to simplify
Nested If Statements

Code for common cases first

Single If statements

Code for most constraining condition first

Switch statement instead of nested if/else

Can be transformed into a **Jump Table**

Loop Unrolling

We can reduce the number of evaluations for each loop by unrolling it

Profiling Code

The technique used to fnd bottlenecks in a program so that the slowest things can be optimized

Use

```
valgrind --tool=callgrind ./program
```

Check

```
kcachegrind callgrind.out.nnnnn
```

Chapter 5 – Real Time Systems

Operating Systems are responsible for **Scheduling Processes** and handling **Resource Management**
OS needs to handle all to these things because all modern systems have multiple processes running
Everything must ask OS for resources, OS can pause/start any other process

Real Time Operation

Deals with immediate changes in environment
Needs to minimize lag:

- Has to have as little lag as possible
- Ensure that there are no conditions under which lag can grow to very long periods

Control systems needs to be run on real time systems
After some time, plans, control input, AI, etc. is not valid for the current state of the system anymore
Case Study: Video games and the ATARI 2600

- Can't have lag
- Interrupt service routines (ISR)

Real Time Constraint

- Operational deadlines must be met
- Multiple processes have deadlines and must be scheduled
- Asynchronous events must be handled
- Types of RTCS:
  - Hard (missed deadlines means system failures)
  - Firm (allows infrequent missed deadlines)
  - Soft (missed deadlines means degraded service)

Schedulers – Scheduled processes to be completed

- **Earliest Deadline First –** not a good idea unless you have enough resources
- **Round Robin –** maybe nothing is completed on time
- **Priority Based** – Processes have priorities, CPU time goes to higher priorities, longer process idles, higher priorirty

Preemptive Multitasking:

Pre-emption: Interrupting a task before it has completed

Context Switch: Switching between tasks, time required to swap depends on what resources needs to be released on task

Task 2, having a higher priority than task one, triggers a context switch. Task 1 releases its resources and Task 2 begins to run. When Task 2 finishes, Task one resumes.

Priority Inversion

Task 2 triggers a context switch and Task 1 begins to release the resources that Task 2 requires. Then Task 3, not needing any of Task 1s resources, interrupts Task 1 and begins to run. When it finishes, Task 1 resumes releasing its resources so that Task 3 can run.

PROBLEM: A medium priority task is stopping a low priority task from releasing its resources, thus a high priority task (T2) is stalled.
SOLUTION: Lower priority task will inherit priority from higher priority task when releasing its resources for the higher priority task

Real Time OS Examples:
QNX: Canadian company working on Operating Systems
Micro kernel -> Minimum set of operating system functionality, scheduling, memory management, interprocess management

Partitioning Operating Systems:
Separation of system resources into partitions to dedicate to individual applications
Each process can only use their own specific partitions
Ensures that process's resource consummation does not grow out of control