6.01 Introduction to EECS via Robotics

Lecture 2: Signals and Systems

Lecturer: Adam Hartz (`hz@mit.edu`)

As you come in...

- Grab one handout (on the table by the entrance)
- Please sit near the front!

Notes

6.01: Introduction to EECS I

The **intellectual themes** in 6.01 are recurring themes in engineering:

- design of complex systems
- modeling and controlling physical systems
- augmenting physical systems with computation
- building systems that are robust to uncertainty

In short, 6.01 is about **engineering design**.

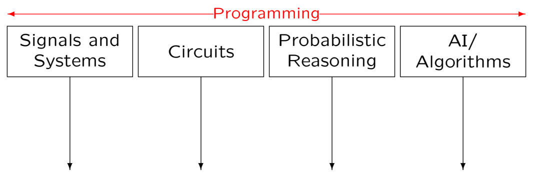
Notes

6.01: Technical Content

6.01 is organized into four modules:

- Signals and Systems
- Circuits
- Probabilistic Reasoning
- Artificial Intelligence / Algorithms

Approach: focus on **key concepts** to pursue **in depth**



Notes

6.01 Structure

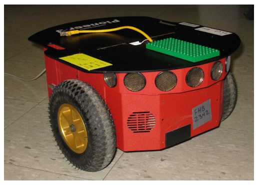
- **Lecture:** Mon, 9:30a-11:00a, in 32-144
- **Software Lab:** Mon, 11:00a-12:30p, in 34-501
  - practice with material from lecture and readings
  - preparation for design lab
  - useful self-check of understanding
- **Design Lab:** Wed, 9:30a-12:30p, in 34-501
  - work with a partner (new partner each week)
  - more open-ended interaction with material
  - checkoffs in lab
- **Homework:** Weekly online readings and exercises
  - Released Monday mornings
  - Due Sunday nights (11pm)
- **Midterm Exams:** Two 2-hour exams (on calendar)
- **Final Exam**

Notes

6.01: Pedagogy

Most of the learning is in the lab!

- active learning with hands-on exercises
- open-ended problems with multiple correct solutions
- multiple levels of individualized help (partners, LAs, TAs, Instructors)



Like anything else, you get good by practicing! Labs and homeworks are an opportunity for *learning*; they are not tests!

Notes

6.01 Logistics

Course web site: https://mit.edu/6.01

- calendar
- grading policy
- collaboration policy
- reference material
- other items

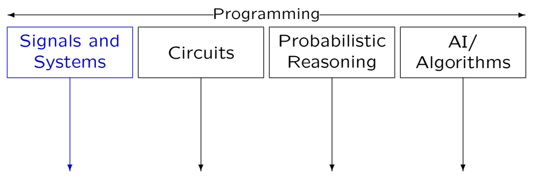
Notes

6.01: Big Ideas

The **intellectual themes** in 6.01 are recurring themes in engineering:

- design of complex systems
- modeling and controlling physical systems
- augmenting physical systems with computation
- building systems that are robust to uncertainty

Approach: focus on **key concepts** to pursue **in depth**

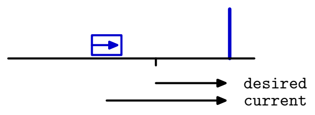


Focus: discrete-time feedback control systems

Notes

Modeling and Analyzing Behavior

Consider the "wall finder" problem from Design Lab 1:



In Design Lab 1, we used a "proportional controller" to cause the robot to keep itself a fixed distance from an obstacle.

Notes

Check Yourself!

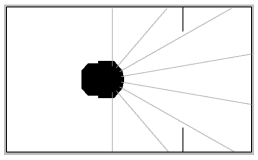
Which of the following has the right form for the robot's forward velocity in a convergent proportional controller, assuming that `k` is positive?

0. `k * (current)`
1. `k * (desired)`
2. `k * (current - desired)`
3. `k * (desired - current)`
4. `k * (current / desired)`
5. `k * (desired / current)`

Notes

Modeling and Analyzing Behavior

Consider the "wall finder" problem from Design Lab 1:



By tuning the gain k in the proportional controller, we can get drastically different behaviors! Consider the following two behaviors.

Notes

Check Yourself!

Which behavior is better?

1. Behavior 1
2. Behavior 2
3. Both are good
4. Both are bad

Notes

Modeling and Analyzing System Behavior

Goal: develop a framework for simulating and analyzing behavior

Many pieces:

- mathematical: difference equations, operator notation
- computational: LTI simulator framework
- analysis: responses, system functionals, poles

Why? – Different perspectives on problem

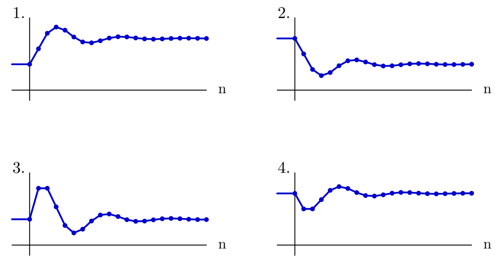
- Mathematical model – crisp description of behavior; analysis predicts future behavior
- Computational model – experimentally test assumptions; debug thinking underlying mathematical model; match observations with predictions

Why? – Need ability to test ideas quickly and efficiently, so need simulation and analysis tools to support

Notes

Check Yourself!

Which of the following best represents the robot's measured distance for the example just shown?



5. None of the above

Notes

The Signals and Systems Abstraction

Signals are functions of a single parameter (time, for our purposes)

- signals represented by capital letters: X
- n^{th} sample of X denoted with lowercase: $x[n]$

Systems transform signals

$$\begin{tikzcd}\text{signal in} \arrow[r] & \boxed{\texttt{system}} \arrow[r] & \text{signal out}\end{tikzcd}$$

In 6.01, we consider **discrete-time** signals and systems.
In 6.01, we consider **LTI** systems.

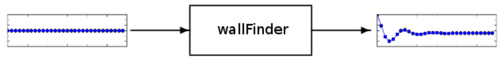
- **linear**: output at time n is a linear combination of previous inputs and outputs (system consists only of delays, gains, and adders)
- **time-invariant**: output of the system does not depend on the absolute time at which the system was started

Notes

Analyzing System Behavior

Represent **systems** (physical, math, computational) by the way they transform an **input signal** into an **output signal**.

$$\begin{tikzcd}\text{signal in} \arrow[r] & \boxed{\texttt{system}} \arrow[r] & \text{signal out}\end{tikzcd}$$



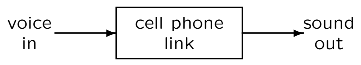
$$\begin{tikzcd}\text{voice in} \arrow[r] & \boxed{\text{cell phone link}} \arrow[r] & \text{sound out}\end{tikzcd}$$

$$\begin{tikzcd}\text{money in} \arrow[r] & \boxed{\text{stock market}} \arrow[r] & \text{money out}\end{tikzcd}$$

Notes

Systems are Modular!

Example: cell phone link



Notes

Modularity and Abstraction

Thinking about complicated systems is *complicated*.

Framework for thinking about complicated systems:

- **Primitives**
- Means of **Combination**
- Means of **Abstraction**
- Recognizing meaningful **Patterns**

Example: Python

- Primitives: `+`, `*`, `==`, `!=`, ...
- Combination: `if`, `while`, `f(g(x))`, ...
- Abstraction: `def`

Notes

Modularity and Abstraction

Primitives:

- fundamental elements on which to build systems
- e.g., ints, floats, simple functions (`+, *`), tests (`==, !=`)

Means of Combination:

- ways to combine primitives to create complex elements
- often used to control flow of information
- e.g., `if, while`, composition of functions `f(g(x))`

Means of Abstraction:

- ways to capture common pattern, treat as if a primitive
- e.g., `def` both captures a pattern within a procedure body, but also gives it a name
- abstraction has standard interfaces – "plug compatible"
- **abstracted components are treated as primitives**
  - suppress details of abstraction from use of abstraction
  - rely on specifications of abstraction to define interface

Notes

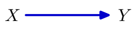
Simple Systems

Wire

Difference Equation:

$$y[n] = x[n]$$

Block Diagram:



Notes

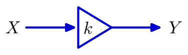
Simple Systems

Gain

Difference Equation:

$$y[n] = k \cdot x[n]$$

Block Diagram:



Notes

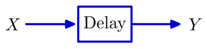
Simple Systems

Delay

Difference Equation:

$$y[n] = x[n-1]$$

Block Diagram:



Notes

Simple Systems

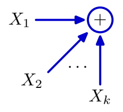
Addition

An adder can be thought of as a system which takes arbitrarily-many inputs and outputs their sum.

Difference Equation:

$$y[n] = x_1[n] + x_2[n] + \ldots + x_k[n]$$

Block Diagram:



Notes

Check Yourself!

Consider the system described by:

$$y[n] = x[n] - x[n-1]$$

What is the output of this system when its input is the "unit sample" sequence Δ?

$$\delta[n] = \begin{cases} 1, & \text{if } n = 0; \\ 0, & \text{otherwise} \end{cases}$$

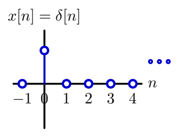
This is often called the **unit sample response** of a system – used in acoustic modeling, radar, ultrasound, control electronics, economics, many other domains.

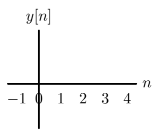
Notes

Representations: Difference Equations

Convenient for step-by-step analysis

Find $y[n]$ given $x[n] = \delta[n]$: $\quad y[n] = x[n] - x[n-1]$



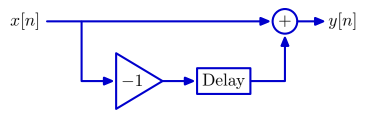


Notes

Representations: Block Diagrams

Graphical representation of the difference equation.

Represent $y[n] = x[n] - x[n-1]$ with a block diagram:



Notes

So far...

Primitives:

- Gain
- Delay
- Adder

Representations:

- Difference Equation
- Block Diagram
- Unit Sample Response

Notes

From Samples to Signals

"Lumping" all of the (possibly infinite) samples into a single object (the **signal**) simplifies its manipulation.

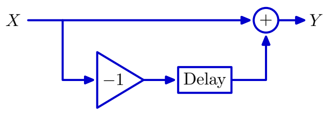
This is analogous to:

- representing coordinates in 3-space as points
- representing lists of numbers as vectors in linear algebra
- creating an object in Python

Notes

From Samples to Signals

Operators manipulate signals rather than individual samples.



Nodes represent whole signals (e.g., X and Y).
Boxes **operate** on those signals:

- Delay: shift whole signal to right by 1 time step
- Add: sum two signals
- -1: multiply by -1

Signals are primitives.
Operators are the means of combination.

Notes

Operator Notation

Systems can now be represented by compact symbolic equations.

Let $\mathcal{R}$ represent the **right-shift operator**:

$$Y = \mathcal{R}\{X\} \equiv \mathcal{R}X$$

where X represents the whole input signal ($x[n]$ for all n) and Y represents the whole output signal ($y[n]$ for all n)

Right-shift means that the entire signal is shifted to the right by one time step.

Notes

Check Yourself!

Let $Y = \mathcal{R}X$. Which of the following is/are true?

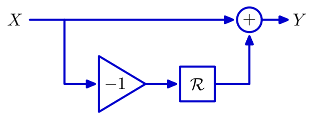
1. $y[n] = x[n]$ for all n
2. $y[n+1] = x[n]$ for all n
3. $y[n] = x[n+1]$ for all n
4. $y[n-1] = x[n]$ for all n
5. None of the above

Notes

Operator Notation

Systems are concisely represented with operators.

Consider:



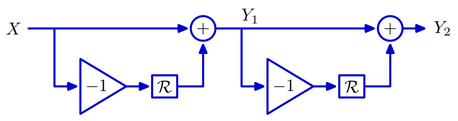
Equivalent representation with $\mathcal{R}$:

$$Y = X - \mathcal{R}X = (1-\mathcal{R})X$$

Notes

Operator Algebra

Cascade of systems: apply second system to output of first

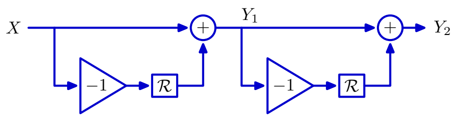


$$Y_1 = (1-\mathcal{R})X$$
$$Y_2 = (1-\mathcal{R})Y_1 = (1-\mathcal{R})(1-\mathcal{R})X$$

Notes

Operator Algebra

Operator expressions expand and reduce like polynomials!



Difference equation:

$$\begin{aligned} y_2[n] &= y_1[n] - y_1[n-1] \\ &= (x[n] - x[n-1]) - (x[n-1] - x[n-2]) \\ &= x[n] - 2x[n-1] + x[n-2] \end{aligned}$$

Operator notation:

$$\begin{aligned} Y_2 &= (1-\mathcal{R})(1-\mathcal{R})X \\ &= (1-2\mathcal{R}+\mathcal{R}^2)X \end{aligned}$$

Notes

Operator Algebra

Expressions involving $\mathcal{R}$ obey all of the familiar laws of algebra. Examples:

- Commutativity:

$$\mathcal{R}(1-\mathcal{R})X = (1-\mathcal{R})\mathcal{R}X$$

- Multiplication distributes over addition:

$$\mathcal{R}(1-\mathcal{R}) = \mathcal{R} - \mathcal{R}^2$$

- Associativity:

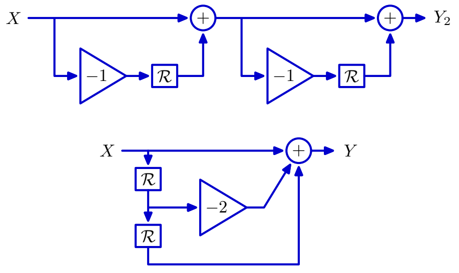
$$(2-\mathcal{R})(\mathcal{R}(1-\mathcal{R})) = ((2-\mathcal{R})\mathcal{R})(1-\mathcal{R})$$

Applies your existing expertise with polynomials to understand block diagrams, and thereby understand systems.

Notes

Operator Approach

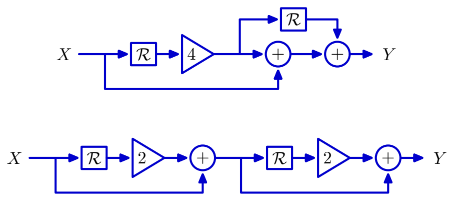
Facilitates seeing relations among systems. The following are "equivalent" in terms of input/output relationship when starting from rest:



Notes

Check Yourself!

Are the following two systems equivalent?



0. No
1. Yes

Notes

Reasoning About L and TI

We can use operator notation to show some effects of:

Time Invariance:
Applying $\mathcal{H}$ to $\mathcal{R}\Delta$ gives the same result as applying $\mathcal{R}$ to $\mathcal{H}\Delta$!

Linearity:
Applying $\mathcal{H}$ to 2Δ gives the same result as applying 2 to $\mathcal{H}\Delta$!

LTI:
Applying $\mathcal{H}$ to $(2\Delta + 3\mathcal{R}\Delta)$ gives the same result as applying $(2 + 3\mathcal{R})$ to $\mathcal{H}\Delta$!

Superposition: powerful tool for analyzing systems.
Can reason about response to arbitrary systems by thinking only about unit sample response (or *vice versa*).

Notes

Check Yourself!

- $y[n]$ is the total number of pairs of bunnies present at time n (regardless of age)
- $x[n]$ is the number of new pairs of (young) bunnies introduced at time n by an outside source
- Young bunnies mature into adult bunnies in one timestep.
- Each pair of adult bunnies produces a pair of young bunnies on each timestep.

Determine an operator expression for the "bunny" system, relating X and Y.

Hint: It may help to introduce two auxilliary signals, one to represent the number of pairs of adult bunnies in the system, and one to represent the number of pairs of child bunnies in the system.

Notes

Block Diagram

Draw a block diagram for the "bunnies" system:

Notes

Simulating

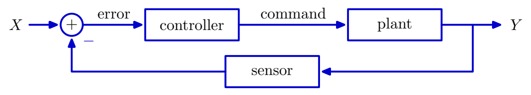
```
>>> children = 1
>>> adults = 0
>>> for i in range(100):
...     print(adults + children, end=" ")
...     newadults = children + adults
...     newchildren = adults
...     adults, children = newadults, newchildren
...

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
832040 1346269 2178309 3524578 5702887 9227465 14930352
24157817 39088169 63245986 102334155 165580141 267914296
433494437 701408733 1134903170 1836311903 2971215073 4807526976
7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
956722026041 1548008755920 2504730781961 4052739537881  ...
```

Notes

Modeling Wall Finder

In order to better understand wall finder, we are going to build mathematical and computational models of the system this week (starting today!). We will break this system into pieces:



We have control over one of these pieces (controller), but the others are part of the system we are trying to control, so we have to try to model what is there.

Notes

Recap

Introduced the signals and systems abstraction.

Introduced several representations of systems:
difference equations, block diagrams, state machines,
operator expressions, sample responses

Discussed applications of operator notation (you'll get more practice in this week's labs and homeworks!)

Saw one example of using the Signals and Systems approach for modeling, and a few examples of converting between different representations.

Software Lab 2:
Begin Modeling Wall Finder

Design Lab 2:
Simulate, Test, and Improve Model

Notes