Measuring analog signals

- This week we start measuring sound with microphone
- A microphone, like all transducers, converts one type of signal into another
  - Acoustic pressure into Current and Voltage
- The output is analog in value and continuous in time

Example: Electret microphone

Dealing with Analog Signals

- A lot of phenomena come at us not as digital (yes/no) things, but rather as continuous signals
- Button pushes are digital, for example
- Microphone signals, light sensors, temperature sensors...these are all analog by their very nature
- We must still deal with these signals in a discrete way, however
Measuring analog signals

- The microcontroller, like almost all computers, is neither analog nor continuous.
- Information in a micro is stored as quantized bits: 0, 1, 2, etc.
- Time passes by in a micro based on clock cycles (which in 6.08 get abstracted up to the loop() function).
- We need to quantize sensor data in value and discretize in time.
- Both of these operations are performed by an analog-to-digital converter (ADC).

This is known as sampling.

On ESP32

- int reading = analogRead(AI0);
- Reading comes back as an integer...not a voltage...

Discrete Signals

- All you have is a list of numbers at discrete points in time!
- Strictly speaking a discrete time signal is undefined for values in between the n’s.
- We often will interpolate if we can assume enough about the system.

Measuring analog signals

- The time between samples
  - T = sampling interval (in sec)
  - 1/T = f_s = sample rate = samples/sec = Hz
- Here:
  - T = 4 msec
  - f_s = 1/0.004 = 250 samples/sec = 250 Hz
You control the sampling rate of the ADC
- May be determined by how quickly you go through your loop
- Or via an explicit timer
- Or via more sophisticated approaches

ESP32 can sample at 10’s kHz (actually, higher)

What about quantization?
- An ADC assigns analog data to a “bin” which will eventually be represented using a fixed number of bits.
- The size of each bin is determined by the ADC resolution aka bit depth
  - 8 bits = $2^8$ bins = 256 bins
  - 12 bits = $2^{12}$ bins = 4096 bins
  - 16 bits = $2^{16}$ bins = 65536 bins

Those bins are arranged over the analog input range
- Example
  - 3-bit ADC has 8 bins
  - Analog input range is 0 to full scale (FS)
  - FS is typically Vcc = 3.3 V for Teensy
  - Each bin is 3.3 V/8 = 0.4125 V

Analog signals are assigned to a bin by the ADC
- More bits ➔ Smaller bins ➔ Capture signals with higher fidelity
- ESP32 has 12-bit ADCs
  - Can use 12-bits or fewer
  - 3.3 V/$2^{12}$ ~ 50 uV/bin
- How many bits to use?
  - Probably fine at 12 bits
  - Use smaller # of bits if you are worried about memory and you don’t need the resolution
- Note: ESP32 cannot read negative voltages! (only 0 to 3.3V!!)
Measuring analog signals

- **To convert back to sensor value**
  - Take read integer value: Ex: 467
  - Know bit depth: Ex: 12 bits
  - Know FS voltage: Ex: 3.3 V
  - Analog value ~ 467/2^{12} x 3.3 V = 0.376 V

Continuous in Value and in Time

Discretization in Time

Discretization in Time and Quantization in Value

4 bit value encoding
Discretization in Time and Quantization in Value

[v(n) = [9,11,5,7,11,11,10,8,5,4,]  

4 bit value encoding

Store in memory

- $v[n] = [9,11,5,7,11,11,10,8,5,4,]$  
- 10 4-bit values: need 40 bits in memory!  
- Easy-peasy. That’s not a lot!

Reproduce

$v[n] = [9,11,5,7,11,11,10,8,5,4,]$  

4 bit value encoding
Compare to original... not bad

\[ v[n] = [9, 11, 5, 7, 11, 11, 10, 8, 5, 4,] \]

4 bit value encoding

Is this good enough?

- Depends on frequency of events in the signal and our sampling rate
- Also depends on the what the range of our signal is expected to be (will it vary by 10 V or 10 mV?...and is there meaning in a variation of 10mV or 10V?)

Continuous in Value and in Time

Discretization in Time and Quantization in Value
Discretization in Time and Quantization in Value

\[ v[n] = \{9,11,5,7,5,12,10,7,5,4,\} \]

4 bit value encoding

Store in memory

- \( v[n] = [9,11,5,7,5,12,10,7,5,4,] \)
- 10 4-bit values: need 40 bits in memory!
- Easy-peasy one-two-threesy

Reproduce

\[ v[n] = [9,11,5,7,5,12,10,7,5,4,] \]

4 bit value encoding

Reproduce

\[ v[n] = [9,11,5,7,5,12,10,7,5,4,] \]

4 bit value encoding
Compare to original... meh

\[ v[n] = [9, 11, 5, 7, 5, 12, 10, 7, 5, 4, \ldots] \]

Really Bad **Discretization Error!**

Continuous in Value and in Time

Discretization in Time and Quantization in Value

\[ v[n] = [9, 9, 9, 9, 9, 9, 9, 9, 9, 9, \ldots] \]

4 bit value encoding

Discretization in Time and Quantization in Value

\[ v[n] = [9, 9, 9, 9, 9, 9, 9, 9, 9, 9] \]

4 bit value encoding
Store in memory

- $v[n] = [9,9,9,9,9,9,9,9,9,9]$  
- 10 4-bit values: need 40 bits in memory!  
- Easy-peasy one-two-threesy

Reproduce

$V(t)$

- $v[n] = [9,9,9,9,9,9,9,9,9,9]$  
- 4 bit value encoding

Reproduce

$V(t)$

- $v[n] = [9,9,9,9,9,9,9,9,9,9]$  
- 4 bit value encoding

Compare... to original also meh

$V(t)$

- $v[n] = [9,9,9,9,9,9,9,9,9,9]$  
- Potentially Really Bad Quantization Error!

Those tiny wiggles might be really important in certain contexts! Mouse heartbeats!
Errors

- Discretization Error: How "off" our readings are in time due to sampling at discrete intervals

- Quantization Error: How "off" our readings are in reproduced value...if our bin size is 50mV and our signal varies only by 20mV this is going to cause problems

Keep in Mind:

- 6.003, 6.302 and others discuss in more depth how good is good enough in terms of things like bit-depth and sampling rate for a particular signal!

- For us, try to sample at least a few times faster than the fastest phenomena we expect to see and for the most part avoid binning if possible (just use finest resolution you can)

Real Signal to Memory

- If we have a known voltage signal in continuous time of $V(t) = \begin{cases} 0 & \text{if } t < 0s \\ t & \text{if } 0 \leq t \leq 3.3s \\ 3.3 & \text{if } t > 3.3s \end{cases}$

- Use an 8-bit ADC to take measurements at 2 Hz

- Starting at $t=0$, what are the first ten measurements?

8 bits give us $b00000000$ to $b11111111$ decimal (0) to decimal (255)

Real Signal to Memory

- Bin size = $3.3V/256 \approx 12.89 \text{ mV/bin}$

  Value 0 covers from 0mV to 12.89mV
  Value 255 covers from 3.2871V to 3.3V
Memory Back to “Real Signal”

• If we wanted we could then use a DAC (Digital-to-Analog Converter) to produce analog voltages based off of stored discrete values.

Flex Resistor Example!!

• Let’s interface with this flex resistor:
  • Use them originally in Nintendo power gloves
  • Resistance varies as they bend
  • Pressure/force resistors are very common

Nintendo PowerPad

• In 1980s/90s Nintendo released Powerpad
• Array of resistive pressure sensors (resistance varied with pressure/mass on them)
• Interfaced with video games so you could “run” or “dance” in the game

Sensor Characterization

Resistance: 60KOhm to 110KOhm

I misread original datasheet so I changed it for our example...handout has real numbers which we don’t use
Flex Resistor Circuit
- Let’s integrate it into a voltage divider (from ex05)
- Assume resistor in our application operates around 80 kOhm value and goes from 60 kOhm to 110 kOhm going from straight (0 degrees) to fully bent (90 degrees))
- $V_{in} = 3.3\text{V}$
- $R_b = 85\text{KOhm}$
- Measure $v_o$ with ESP32
- Use 12 bit ADC

With what resolution can we measure our bending? Can we actually measure all 90 degrees of bending? (assume resistance varies linearly with bend)

Quantization
- 12 bits with 3.3V FS gives us $3.3/(2^{12})$ volts/bin
  - $0.8057 \text{mV/bin}$
- @ straight (0 degrees): $v_o = 1.365\text{V}$
  - Quantized value: floor($1.365\text{V}/(0.8057\text{mV/bin})$) =1694
- @ bent (90 degrees): $v_o = 1.862\text{V}$
  - Quantized value: floor($1.862\text{V}/(0.8057\text{mV/bin})$)=2311
- So we have: 2311-1694 = 617 values to cover 90 degrees of bend. More than enough.
HTTP

• HyperText Transfer Protocol
• Request-Response Protocol
• Client and Server:
  • Client makes a request
  • Server provides a response
• Both sides of this exchange have very strict requirements on their formatting!

Basic Pattern

The client sends a request to the server
• The server parses it, carries out the specified actions (as dictated by internal code), and then returns a response
• There are two major verbs for requests that we use:
  • GET
  • POST

The Meaning of GETs and POSTs

• GETs are meant to request a resource from a server. In general, a server should not update its state (databases) from a GET

• POSTs are meant to report/provide information to a server with the intent for it to be stored/logged
Reality

- You can do whatever with GETs and POSTs (put stuff in database in response to both of them if you want), but it is a good convention to use with when creating an API or interfacing with one.

GET

GET /608dev/sandbox/jodalyst/special.py?cat=brown&dog=blue HTTP/1.1
Host: iesc-s1.mit.edu

On the Python side of our web framework the request dictionary looks like the following:

{"args":["dog", "cat"], "method": 'GET', 'values': {'cat': 'brown', 'dog': 'blue'}}

GET is what you do in a web browser

- ie: UTF8 (text to render)
- qid: 1520861199 (query id...for logging/remembering actions)
- sr=8-2-spons (no idea)
- keywords=esp32 (my search query)
- psc=1 (no idea)

POST

POST /608dev/sandbox/jodalyst/special.py HTTP/1.1
Host: iesc-s1.mit.edu
Content-Type: application/x-www-form-urlencoded
Content-Length: 18

cat=brown&dog=blue

On the Python side of our system the request dictionary looks like the following:

{"method": 'POST', 'form': {'cat': 'brown', 'dog': 'blue'},
'is_json': False, 'values': {}, 'args': []}
POST

POST /608dev/sandbox/jodalyst/special.py HTTP/1.1
Host: iesc-s1.mit.edu
Content-Type: application/json
Content-Length: 28

{"cat":"brown","dog":"blue"}

On the Python side of our system the request dictionary looks like the following:

{"args": [], 'method': 'POST', 'values': {}, 'data': '{"cat":"brown","dog":"blue"}', 'is_json': True}

If you mix up the encoding/content-type some systems will throw errors, and some won't...our's will just shove it into 'data' field, so you might need to do some checking on that to see if it is indeed json.

POST

Json provides more flexibility in the structure of the body

POST /608dev/sandbox/jodalyst/special.py HTTP/1.1
Host: iesc-s1.mit.edu
Content-Type: application/json
Content-Length: 56

{"cat":"brown","dog":"blue","favorite_numbers":[1,4,11]}

On the Python side of our system the request dictionary looks like the following:

{"values": {}, 'data': '{"cat":"brown","dog":"blue","favorite_numbers":[1,4,11]}', 'method': 'POST', 'args': [], 'is_json': True}

Pros/Cons?

• In a GET, all components are in the URL, including potentially things that matter. This is less secure since information is automatically stored in server logs.

• In a POST you have a body (GET does not), if you are sending up potentially secure info, you should put it into the POST (and also encrypt..but do that later)

• POST is also far less limited in what you can put in body while GET is limited mostly to key-value pairs
Putting Query Arguments in POST?

- Yeah you can do it. It does sort of voids the point and benefit of POST, but a number of you did that.

Chained-Events

- Things can act as both servers and clients in certain contexts.
- We used this framework in the Wikipedia exercises, and a number of you used this in recent design exercises!
Chain of Events

1. ESP (Client 1) sends request to iesc-s1.mit.edu (Server 1)
2. iesc-s1 (Client 2) sends request to Wikipedia (Server 2)
3. Wikipedia (Server 2) provides response to iesc (Client 2)
4. iesc (Server 1) provides response to ESP (Client 1)

• The machine in the middle takes turns being both a server and a client (so the role can change)

Record Audio and Generate POST to Google Speech API

More complex...

```
POST /v1/speech:recognize?key=AIpaSyC2hT5F69sBBaldwhMkcf_nLxzpexAMslg HTTP/1.1
Host: speech.googleapis.com
Content-Type: application/json
Cache-Control: no-cache
Content-Length: 32100

{ "config": { "encoding": "MULAW", "sampleRateHertz": 8000, "languageCode": "en-US" },
  "audio": { "content": "Li1e2zZOMkclPzsy………………SkrJisw………………" } }
```

And the response...

```
{
  "results": [
    {
      "alternatives": [
        {
          "transcript": "hey b c d e f",
          "confidence": 0.8089218
        }
      ]
    }
  ]
}
```

The “…” means 32,070 characters go here
Example/Design: Smart Lighting

One way to do it

HTTP Requests

Looking Forward

• The first three exercises this week are basically building up a full-stack system (and we give you a lot of the code). Please use this as a system review!

• The fourth exercise is sort of a high-level walk through of how information is flowing through our system. Again a bit more of a review.