Self Playing Harmonica

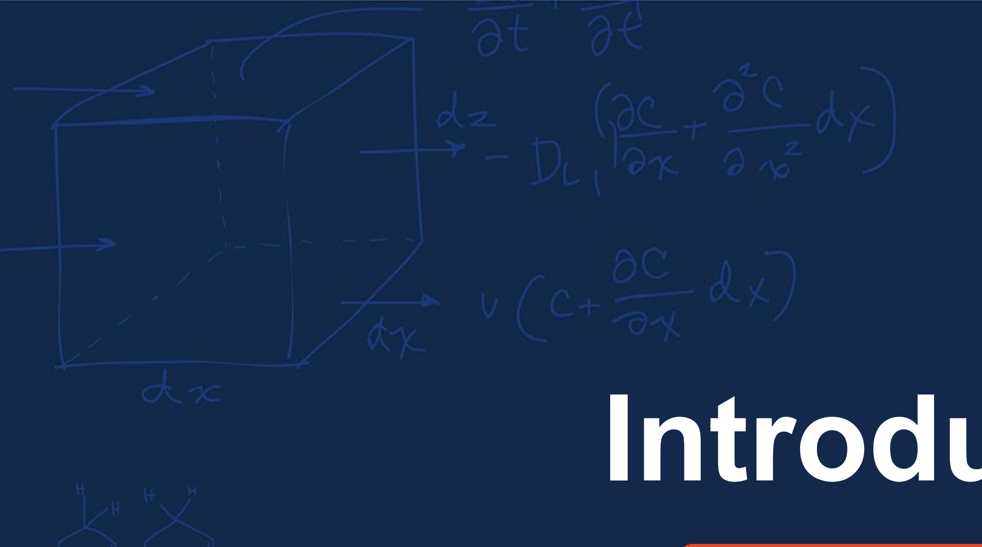
Team 66

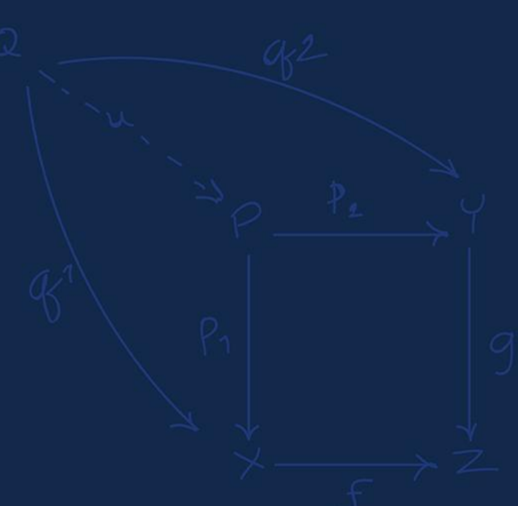
Sean Jasin, David Zhang, Robert Zhu

TA: Wenjing Song

Electrical & Computer Engineering

05/05/26





Introduction

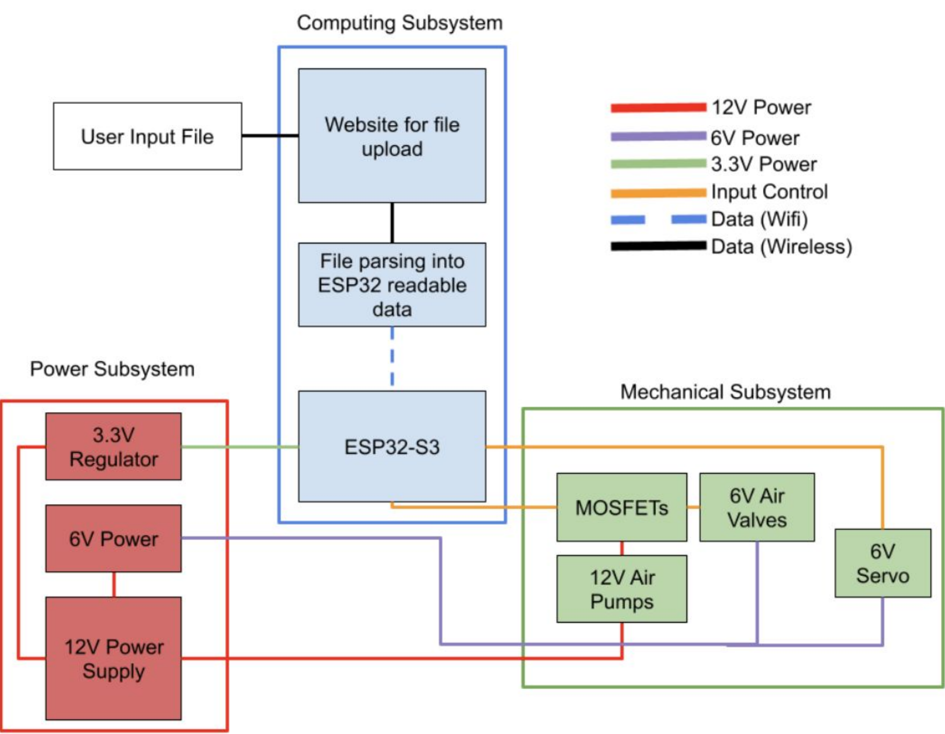
Problem

The harmonica is a technically difficult instrument to play. There is a need for the background music of a live instrument, yet it is difficult to master the harmonica. Some lack the time to practice and learn the harmonica. For others, they may no longer be able to physically play the harmonica, or do not have access to training or musical education. Existing self playing musical devices exist for keyboard, but not for wind instruments. There is a need for a self-playing harmonica that can produce melodies without requiring manual lip or breath control.

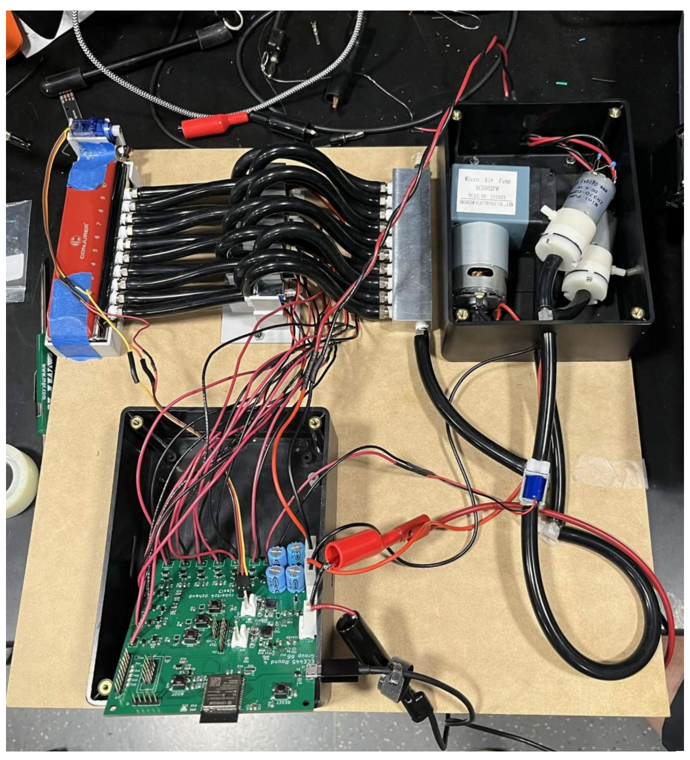
Solution

The self playing harmonica is a device that can be controlled via WiFi, and play a song that is uploaded to the website. Air pumps enable the device to produce airflow in both directions, while air valves control where the airflow is directed. A servo is used to actuate the harmonica's slide, allowing for the harmonica to play all notes.

Block Diagram



Self-Playing Harmonica System



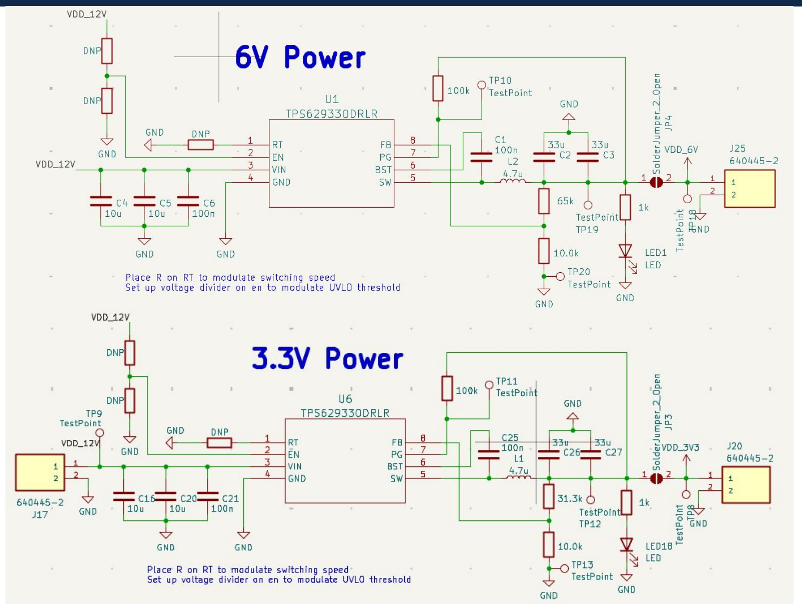
Self-Playing Harmonica System

Requirements (High Level)

1. The device must be able to operate all basic functionality of a chromatic harmonica.
   a. Playing all notes (blow in and suck out air for all holes)
   b. Engage and disengage the slide.
   c. The notes must be audible at 10m from a listener
2. The device must be able to receive song data wirelessly.
   a. User interface latency must not exceed 1s
3. The device must be able to operate the harmonica continuously for a minimum of 5 minutes

Subsystems

Power Subsystem



Components:

- Buck converters (12V to 6V and 3V3)
- Power LEDs (Power good indicator)
- Microcontroller (3V3)
- Drivers & Controllers (12V, 6V)

Testing & Verification

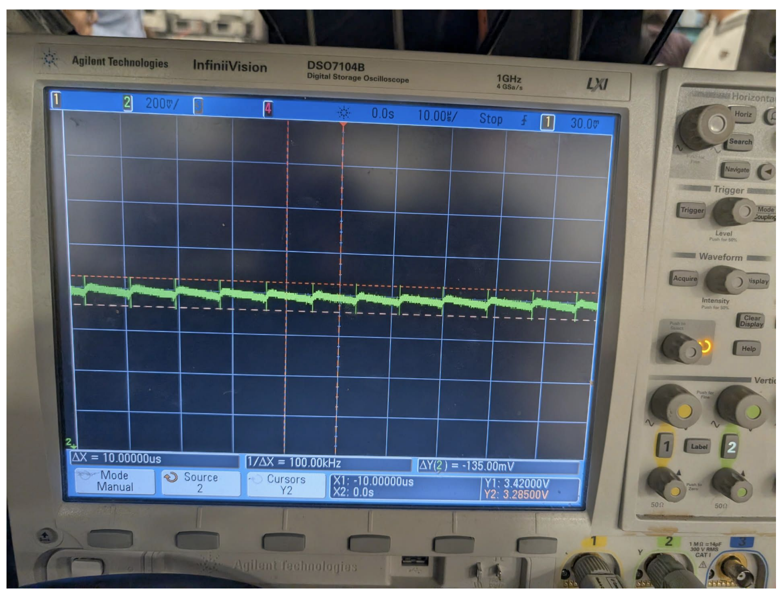
- Testing methods
  - Modular testing
  - Full testing
- Verification
  - Oscilloscope (voltage sag, brownout, etc)
  - Bench PSU current draw

ECE445 Round 4
Group 66
RESET
BOOT
USER IO
12V Input
robertz4 dzhan6
sjasi3

Power Subsystem Verification: PSU Ripple

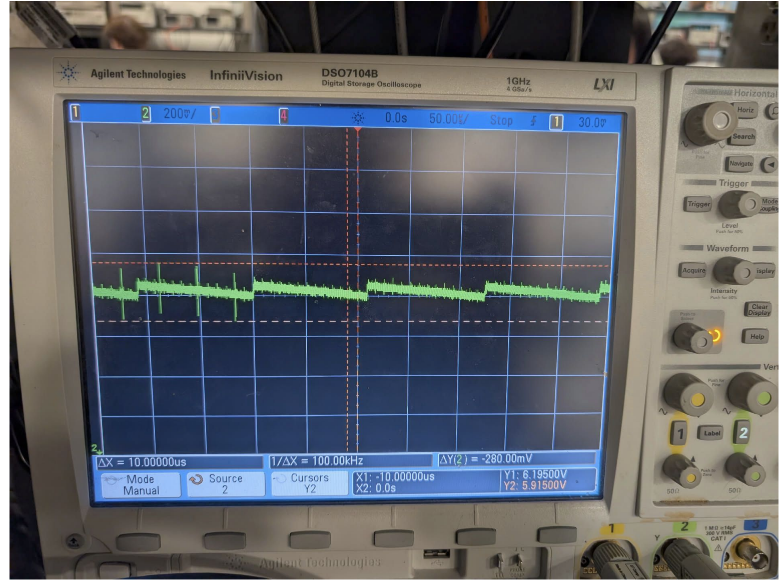
3.3V

- Average voltage: 3.327V
- Ripple: 135 mV



6V

- Average voltage: 6.036
- Ripple: 280 mV

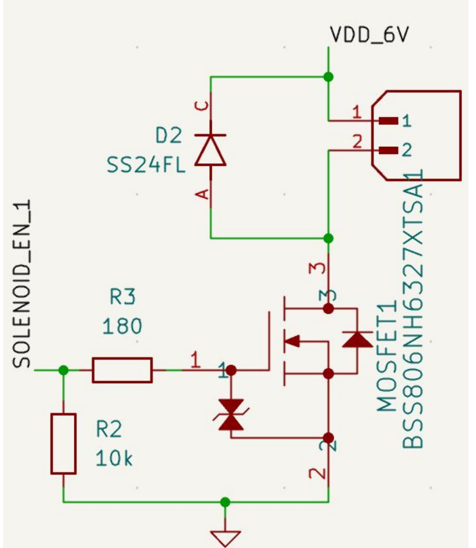


Components:

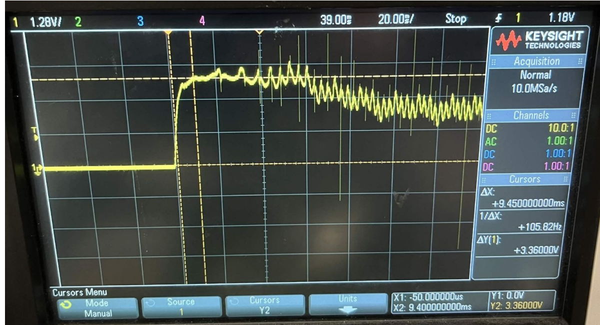
- MOSFET drivers to convert 3V3 signal to 6V solenoid power and 12V motor power
- Servo control and motor control using PWM

Testing & Verification

- MOSFET Driver oscilloscope testing
- Inrush current testing
- Proper actuation of solenoids and harmonica lever



Motor driver circuit

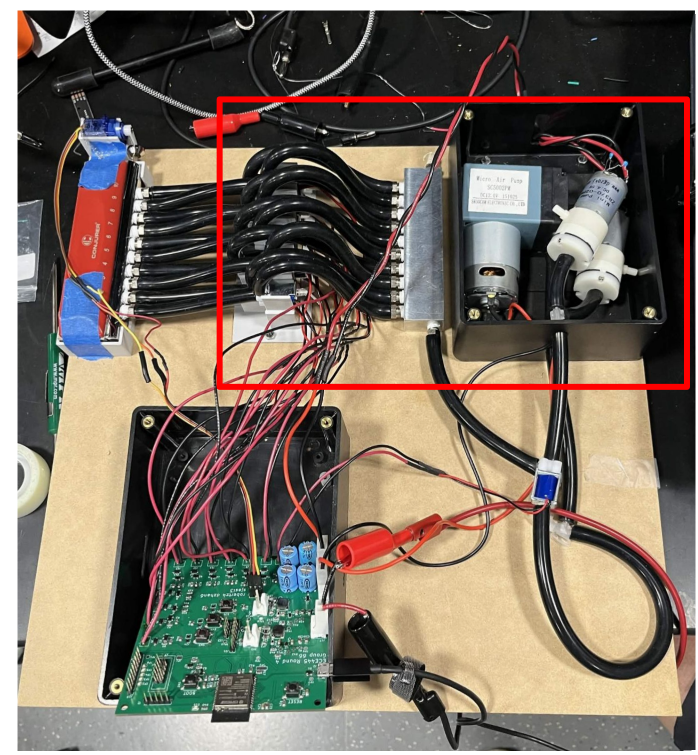


Motor inrush current testing

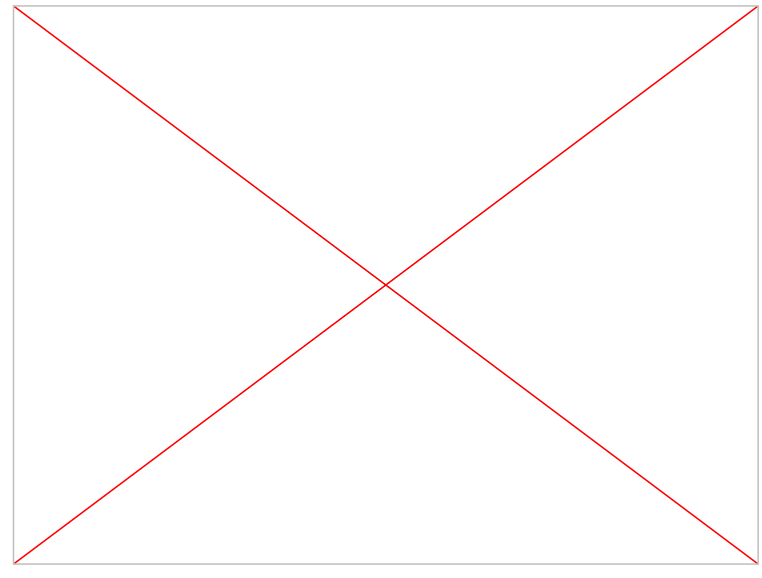
Mechanical Subsystem

Components:

- Air pumps
- Solenoids
- 2 to 10 manifold
- Air connections using ⅛” ID rubber hose tubing



Mechanical subsystem highlighted in red



Computing Subsystem

- Wireless AP
- Self hosted website (API based)
- LED control
- Mechanical Control

Testing & Verification

- Wireless network is hosted
- Website API can be accessed
- Setting values in the API is reflected in the hardware

Computing Subsystem

```
async function playMidi() {
    const fileInput = document.getElementById('midi-upload');
    if (!fileInput.files[0]) return alert("Select a MIDI file!");

    const reader = new FileReader();
    reader.onload = async (e) => {
        try {
            const midi = new Midi(e.target.result);
            log("SUCCESS: Parsed " + (midi.name || "Song"));
            isPlaying = true;
            document.getElementById('status').innerText = "Status: Playing...";

            midi.tracks.forEach(track => {
                if (track.notes.length === 0) return;
                track.notes.forEach(note => {
                    // FIX: Use note.velocity, not note.volume
                    let tOn = setTimeout(() => {
                        if(isPlaying) triggerHarp(note.midi, note.velocity, note.name);
                    }, note.time * 1000);

                    let tOff = setTimeout(() => {
                        if(isPlaying) triggerHarp(note.midi, 0, note.name);
                    }, (note.time + note.duration) * 1000);

                    activeTimeouts.push(tOn, tOff);
                });
            });
        } catch (err) {
            log("ERROR: MIDI Parser failed.");
        }
    };
    reader.readAsArrayBuffer(fileInput.files[0]);
}
```

.MIDI File Parser

- Converts MIDI file into harmonica device instructions
- Parses .MIDI file using Tone.js library
- Reads length, pitch, and volume of each track in the .MIDI file
- Has a separate on time and off time to play multiple notes at once

Computing Subsystem

```
function triggerHarp(midiNum, volume, name) {
    if (!isPlaying && volume > 0) return;

    let data = HARP_LOGIC[midiNum] || {h: 0, a: 0, s: 0};
    let a_val = 0;

    if (volume > 0) {
        // Map 0.0-1.0 to 0-100 range for PWM
        let intensity = Math.round(volume * 100);
        a_val = (data.a === 1) ? intensity : -intensity;
    }

    sendManualCmd(data.h, a_val, data.s);

    if (volume > 0) {
        log(`ON | ${name} | Hole: ${data.h} | Vol: ${Math.round(volume * 100)}%`);
    }
}
```

```
function sendManualCmd(h, a, s) {
    const cmd = `/play?h=${h}&a=${a}&s=${s}`;
    const url = "http://192.168.4.1" + cmd;

    const liveUrlDisplay = document.getElementById('live-url');
    if(liveUrlDisplay) liveUrlDisplay.innerText = url;

    fetch(url).catch(() => {}); // Silent catch to prevent console spam if ESP32 lags
}
```

MIDI Parser Continued

- Uses parsed information to create values for hole number, volume/PWM, and slide in/out
- Builds url using values and sends via HTTP request to ESP32
- Hole numbers are hard-coded based on 10-hole chromatic harmonica

Chromatic

Holes: 1 2 3 4 5 6 7 8 9 10

C d | E f | G a | C b | C d | E f | G a | C b | C d | E f — KEY OUT

Scale of C — Scale of C

Holes: 1 2 3 4 5 6 7 8 9 10

C♯ d♯ | E♯ f♯ | G♯ a♯ | C♯ b♯ | C♯ d♯ | E♯ f♯ | G♯ a♯ | C♯ b♯ | C♯ d♯ | E♯ f♯ — KEY IN

```
const HARP_LOGIC = {
    // Hole 1
    60: {h:1, a:1, s:0}, 61: {h:1, a:1, s:1}, // C4, C#4
    62: {h:1, a:2, s:0}, 63: {h:1, a:2, s:1}, // D4, D#4
    // Hole 2
    64: {h:2, a:1, s:0}, 65: {h:2, a:1, s:1}, // E4, F4
    65: {h:2, a:2, s:0}, 66: {h:2, a:2, s:1}, // F4, F#4
    // Hole 3
    67: {h:3, a:1, s:0}, 68: {h:3, a:1, s:1}, // G4, G#4
    69: {h:3, a:2, s:0}, 70: {h:3, a:2, s:1}, // A4, A#4
    // Hole 4
    71: {h:4, a:2, s:0}, 72: {h:4, a:2, s:1}, // B4, C5
    72: {h:4, a:1, s:0}, 73: {h:4, a:1, s:1}, // C5, C#5
    // Hole 5
    72: {h:5, a:1, s:0}, 73: {h:5, a:1, s:1}, // C5, C#5
    74: {h:5, a:2, s:0}, 75: {h:5, a:2, s:1}, // D5, D#5
    // Hole 6
    76: {h:6, a:1, s:0}, 77: {h:6, a:1, s:1}, // E5, F5
    77: {h:6, a:2, s:0}, 78: {h:6, a:2, s:1}, // F5, F#5
    // Hole 7
    79: {h:7, a:1, s:0}, 80: {h:7, a:1, s:1}, // G5, G#5
    81: {h:7, a:2, s:0}, 82: {h:7, a:2, s:1}, // A5, A#5
    // Hole 8
    83: {h:8, a:2, s:0}, 84: {h:8, a:2, s:1}, // B5, C6
    84: {h:8, a:1, s:0}, 85: {h:8, a:1, s:1}, // C6, C#6
    // Hole 9
    84: {h:9, a:1, s:0}, 85: {h:9, a:1, s:1}, // C6, C#6
    86: {h:9, a:2, s:0}, 87: {h:9, a:2, s:1}, // D6, D#6
    // Hole 10
    88: {h:10, a:1, s:0}, 89: {h:10, a:1, s:1}, // E6, F6
    89: {h:10, a:2, s:0}, 90: {h:10, a:2, s:1}, // F6, F#6
};
```

Embedded WiFi

```
 5 // WiFi AP Setup
 6 void init_wifi() {
 7     ESP_ERROR_CHECK(esp_netif_init());
 8     ESP_ERROR_CHECK(esp_event_loop_create_default());
 9     esp_netif_create_default_wifi_ap();
10
11     wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
12     ESP_ERROR_CHECK(esp_wifi_init(&cfg));
13
14     wifi_config_t wifi_config = {
15         .ap = {
16             .ssid = WIFI_SSID,
17             .ssid_len = strlen(WIFI_SSID),
18             .password = WIFI_PASS,
19             .max_connection = 4,
20             .authmode = WIFI_AUTH_WPA2_PSK
21         },
22     };
23
24     ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_AP));
25     ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_AP, &wifi_config));
26     ESP_ERROR_CHECK(esp_wifi_start());
27     ESP_LOGI(TAG, "WiFi AP Started. SSID:%s", WIFI_SSID);
28 }
```

```
14     httpd_handle_t server = NULL;
15     httpd_config_t config = HTTPD_DEFAULT_CONFIG();
16
17     if (httpd_start(&server, &config) == ESP_OK) {
18         httpd_uri_t play_uri = {
19             .uri      = "/play",
20             .method   = HTTP_GET,
21             .handler  = play_api_handler,
22             .user_ctx = NULL
23         };
24         httpd_register_uri_handler(server, &play_uri);
25
26         ESP_LOGI(TAG, "Server started on port: '%d'", config.server_port);
27     } else {
28         ESP_LOGE(TAG, "Failed to start server!");
29     }
```

WiFi connection

- Utilizes the esp_wifi.h library for creating an access point which the user can access

Self-Hosted Web Server

- Utilizes the esp_http_server.h library to create an API (application programming interface) which responds to certain http requests it receives
- Sets the URL as http://[IP Address]/play

Embedded Web Server Internal Calls

```
        if (httpd_query_key_value(buf, "a", buf2, sizeof(buf2)) == ESP_OK) {
            int speed = atoi(buf2);

            // Clamp speed between -100 and 100
            if (speed > 100) speed = 100;
            if (speed < -100) speed = -100;

            uint32_t duty = (abs(speed) * PWM_PERIOD) / 100;

            if (speed > 0) {
                // Forward
                solenoid_state[SOLF] = 1;
                solenoid_state[SOLR] = 0;
                mcpwm_comparator_set_compare_value(comp_a, duty);
                mcpwm_comparator_set_compare_value(comp_b, 0);
            } else if (speed < 0) {
                // Reverse
                solenoid_state[SOLR] = 1;
                solenoid_state[SOLF] = 0;
                mcpwm_comparator_set_compare_value(comp_a, 0);
                mcpwm_comparator_set_compare_value(comp_b, duty);
            } else {
                // Stop
                solenoid_state[SOLF] = 0;
                solenoid_state[SOLR] = 0;
                mcpwm_comparator_set_compare_value(comp_a, 0);
                mcpwm_comparator_set_compare_value(comp_b, 0);
            }

            ESP_LOGI(TAG, "Motor Speed: %d", speed);
            char resp[64];
            snprintf(resp, sizeof(resp), "{\"status\":\"ok\",\"val\":%d}", speed);
            httpd_resp_set_type(req, "application/json");
            httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
        }
```

Motor API call

- “a” value
- The “a” values determines the **a**ction of the motor
- Positive represents pushing
- Negative represents pulling
- The integer represents the percentage of the duty cycle for the motor
- For example
    - a=100: 100% duty, push
    - a=-100: 100% duty, pull
    - a=67: 67% duty, push
    - a=-32: 32% duty, pull

Embedded Web Server Internal Calls

```c
      if (httpd_query_key_value(buf, "h", buf2, sizeof(buf2)) == ESP_OK &&
          httpd_query_key_value(buf, "a", buf3, sizeof(buf3)) == ESP_OK) {
        int whole_note = atoi(buf2);
        int active = atoi(buf3);
        ESP_LOGI(TAG, "%d is set: %d", whole_note, active);
        if (whole_note > 0 && whole_note <= SCNT) {
          solenoid_state[whole_note - 1] = active;
        }
        for (int i = 0; i < SCNT; i++) {
          gpio_set_level(solenoid_pins[i], solenoid_state[i]);
          ESP_LOGI(TAG, "%d is pin: %d and is set: %d", i, solenoid_pins[i],
                   solenoid_state[i]);
        }

        char resp[64];
        snprintf(resp, sizeof(resp), "{\"status\":\"ok\",\"pin_state\":%d}",
                 aux_state);
        httpd_resp_set_type(req, "application/json");
        httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
      }
```

Solenoid API call

- “h” value (“a” as well)
- The “h” values determines the **h**ole which the air must go through
  - In order for a solenoid to open, the “a” value must not be 0, this means there is air being pushed or pulled
- If h=5 and a!=0, the solenoid leading to hole 5 will be opened
- If h=5 and a=0, the solenoid leading to hole 5 will be closed

Embedded Web Server Internal Calls

```
if (httpd_req_get_url_query_str(req, buf, sizeof(buf)) == ESP_OK) {
    char ang_str[10];
    if (httpd_query_key_value(buf, "s", ang_str, sizeof(ang_str)) == ESP_OK) {
        int angle = atoi(ang_str);
        if(angle > 0) {
          set_servo_angle(70);
          ESP_LOGI(TAG, "Servo Angle set to: %d", 70);
        } else {
          set_servo_angle(90);
          ESP_LOGI(TAG, "Servo Angle set to: %d", 90);
        }
        httpd_resp_send(req, "{\"status\":\"ok\",\"type\":\"servo\"}", HTTPD_RESP_USE_STRLEN);
    }
}
```

Servo API call

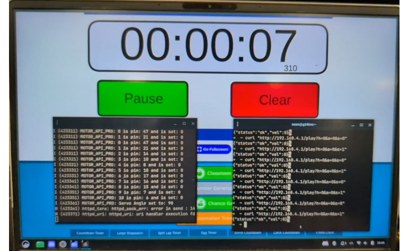
- “s” value
- The “s” values determine the **s**ervo position for the harmonica slider. It has 2 states, slide engaged or slide disengaged
- If the value of “s” is greater than 0, then the servo engages the harmonica slider
- This call makes a call to the set_servo_angle function for readability

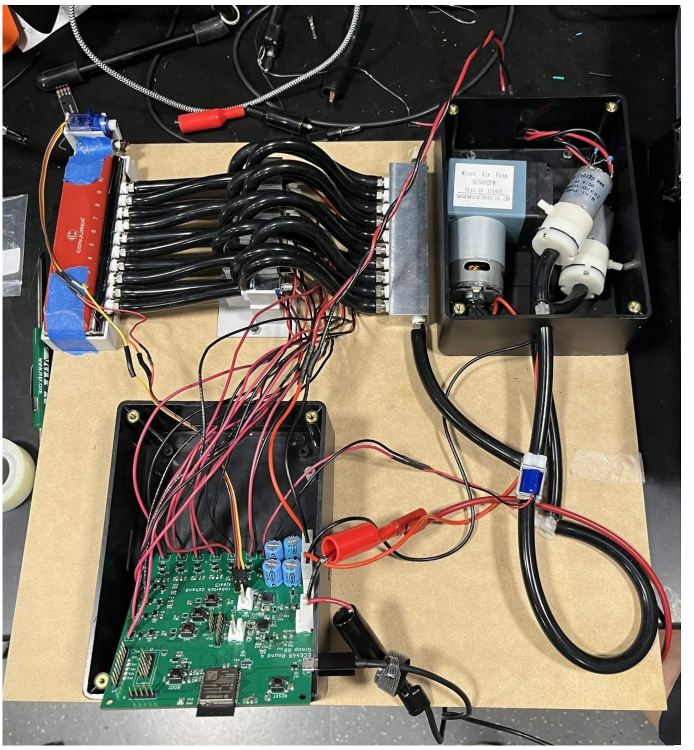
Latency testing

- Start time: (7.246s)
- End time: (7.310s)

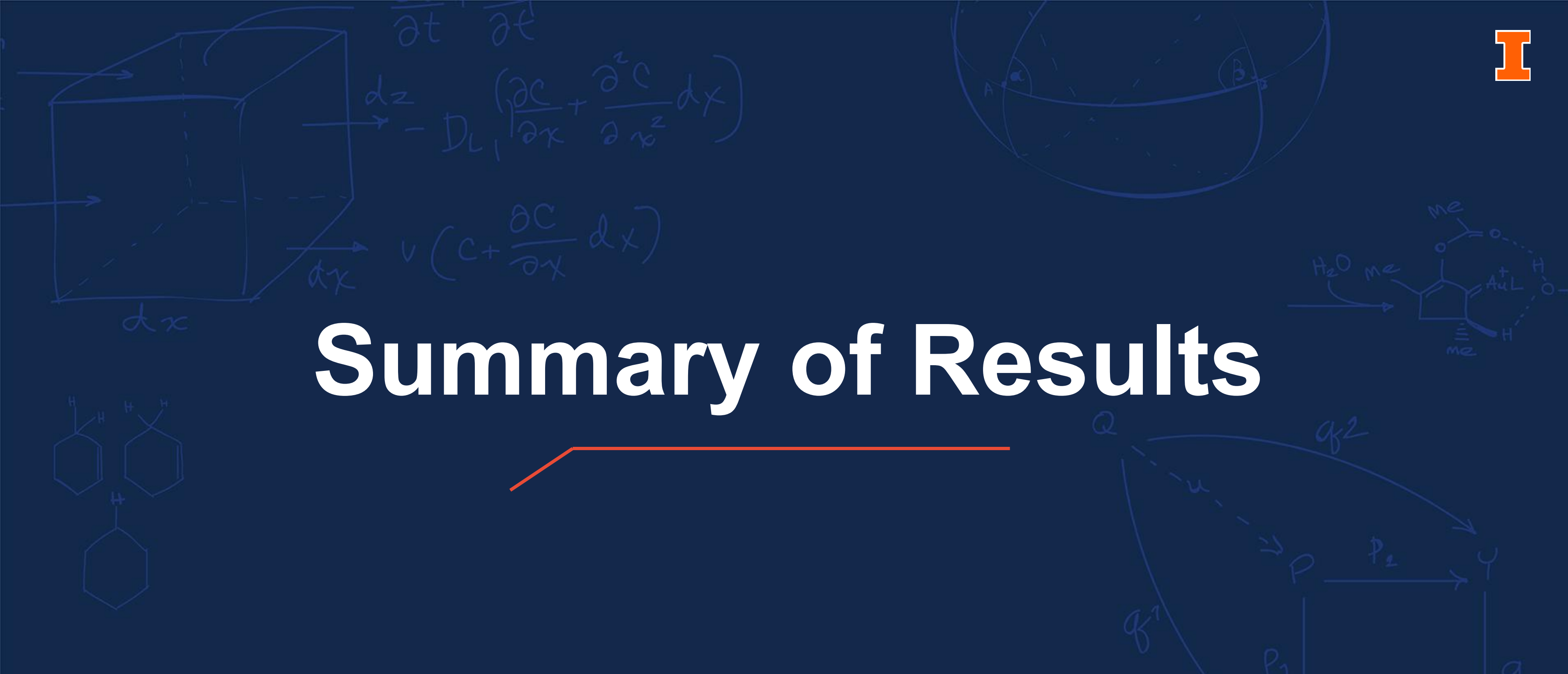
Play stability & duration

- Able to sustain playing for 5 minutes without malfunctioning.





Self-Playing Harmonica System



Summary of Results

Project Successes

Power Subsystem

- Fully functioning PCB
  - All motor drivers are fully functional with minimal turn-on time
  - Reduction in ringing from motors
  - Proper buck-converter voltage values
  - No overheating from usage

Computing Subsystem

- Low latency HTTP connection
- MIDI parsing for note length and pitch

Mechanical Subsystem

- Proper actuation distance of servo
  - Actuates harmonica lever with sufficient speed
- Fast on-off time for motors
- Solenoids turn on with sufficient speed

High Level Requirements

Requirements	Verification
The device must be able to operate all basic functionality of a chromatic harmonica. - Playing all notes (blow in and suck out air for all holes) - Engage and disengage the slide. - The notes must be audible at 10m from a listener	• ❌ All notes are not able to be played ○ 3/10 holes were able to be played, one at a time • ✅ Slide was able to be fully engaged and disengaged • ❌ Notes were not audible above the motors from 10 feet away
The device must be able to receive song data wirelessly. - User interface latency must not exceed 1s	• ✅ Air pumps are able to both push and draw air • ✅ Latency did not exceed 1s (latency of 64ms)
1. The device must be able to operate the harmonica continuously for a minimum of 5 minutes	• ✅ Self-playing harmonica can operate safely for at least 5 minutes

Challenges

Project Challenges

Power Subsystem

- Initial ringing from motor
- Small component soldering

Computing Subsystem

- Understanding ESP32 standard libraries
- Initial learning curve of idf.py build system

Mechanical Subsystem

- 12V motor does not have consistent airflow, which affects harmonica sound
- Motors do not have enough airflow to generate sound in the harmonica
- Solenoid inner tube diameter is too small, restricting and preventing sufficient airflow
- Connections of tubing to solenoids are not sealed
- Solenoid heating under prolonged testing

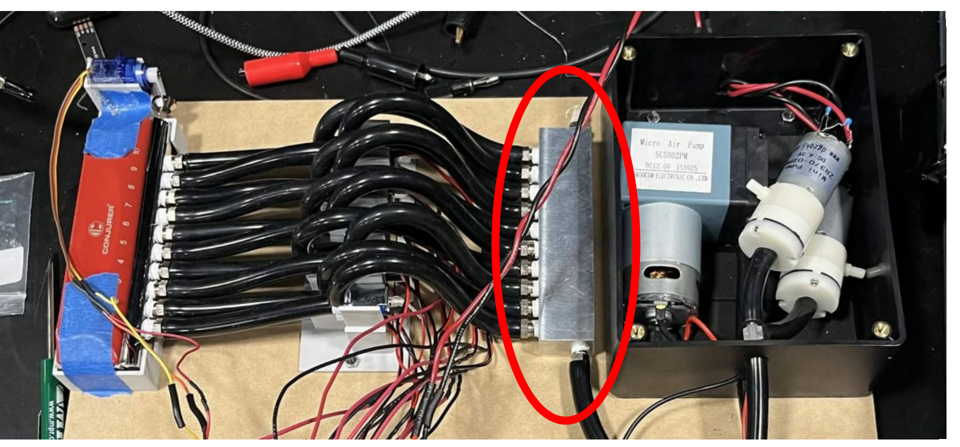
Project Challenges

Mechanical Subsystem

- Manifold adds too much internal volume
- 12V motor does not have consistent airflow, which affects harmonica sound
- Motors do not have enough airflow
- Solenoid inner tube diameter is too small, restricting and preventing sufficient airflow
- Connections of tubing to solenoids are not adequately sealed
- Solenoid heating under prolonged testing

Manifold problems

- Large internal volume of manifold and associated tubes increases delay, decreases airflow
- Replaced with T-joints
  - Less notes
  - Reduces internal volume, reducing delay



Air Pump Iteration Comparison

Pump 1

- 4.5V, 1.3A inrush, 0.55A DC
- 2.5 LPM flow rate
- 7.9 PSI static pressure

Pump 2

- 12V, 1.3A inrush, 0.7A DC
- 12 LPM flow rate
- 29 PSI static pressure

Pump 3

- 12V, current not tested
- 680 LPM flow rate
- 0.5 PSI static pressure

Final pump used: pump 2

Advantages:

- Highest air flow rate/static pressure balance

Disadvantages:

- Inconsistent air flow (blows air in pulses)
  - Pulsing air flow affects harmonica reeds and creates harmonics
- Not enough air flow to harmonica

Air Flow Problems

- Rated for 10-12 L/min flow rate (but when venting to atmosphere)
- However, more resistance -> lower flow rate
- With two solenoids, theoretical maximum airflow reduced to ~4.9 LPM flow rate (if initially 12 L/min)

性能曲线图 Performance curve:

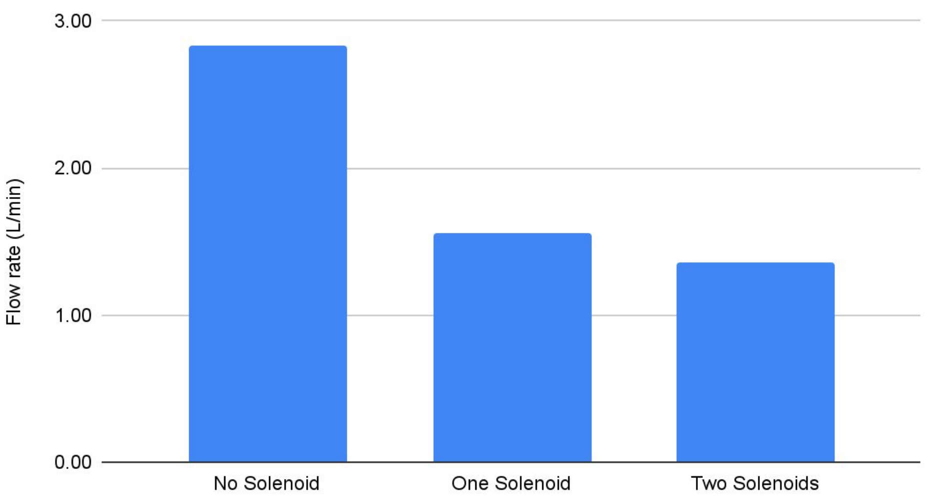
性能曲线图 Performance curve:流量和噪音变化

Flow/L
Noise/dB
Flow（L/min）
Noise（dB）
Pressure（kPa）
14
12
10
8
6
4
2
80
70
60
50
40
30
20
10
0

Air Flow Restriction due to Solenoid

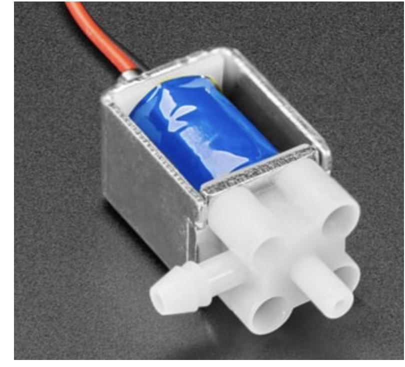
30 second water displacement test

	No Solenoid	One Solenoid	Two Solenoids
Mass Before	1654.7	1645.6	1707.7
Mass After	241.1	865.11	1026.1
Difference in Mass	1413.6	780.49	681.6
Flow rate (L/min)	2.83	1.56	1.36

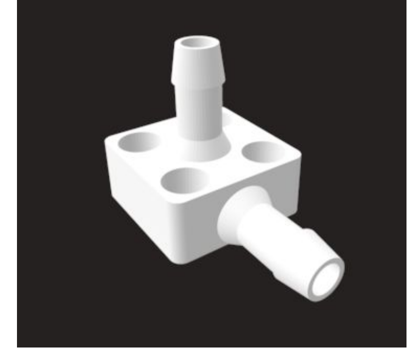


Restrictive Solenoid Bore

- Air flow was heavily reduced by passage through solenoids
    - Original solenoid bore was 1mm
    - Updated 3D printed solenoid increases bore to 2.5mm
- Increased air leakage



1mm (original)



2.5mm (printed)

Conclusion

Future Work

Mechanical Subsystem

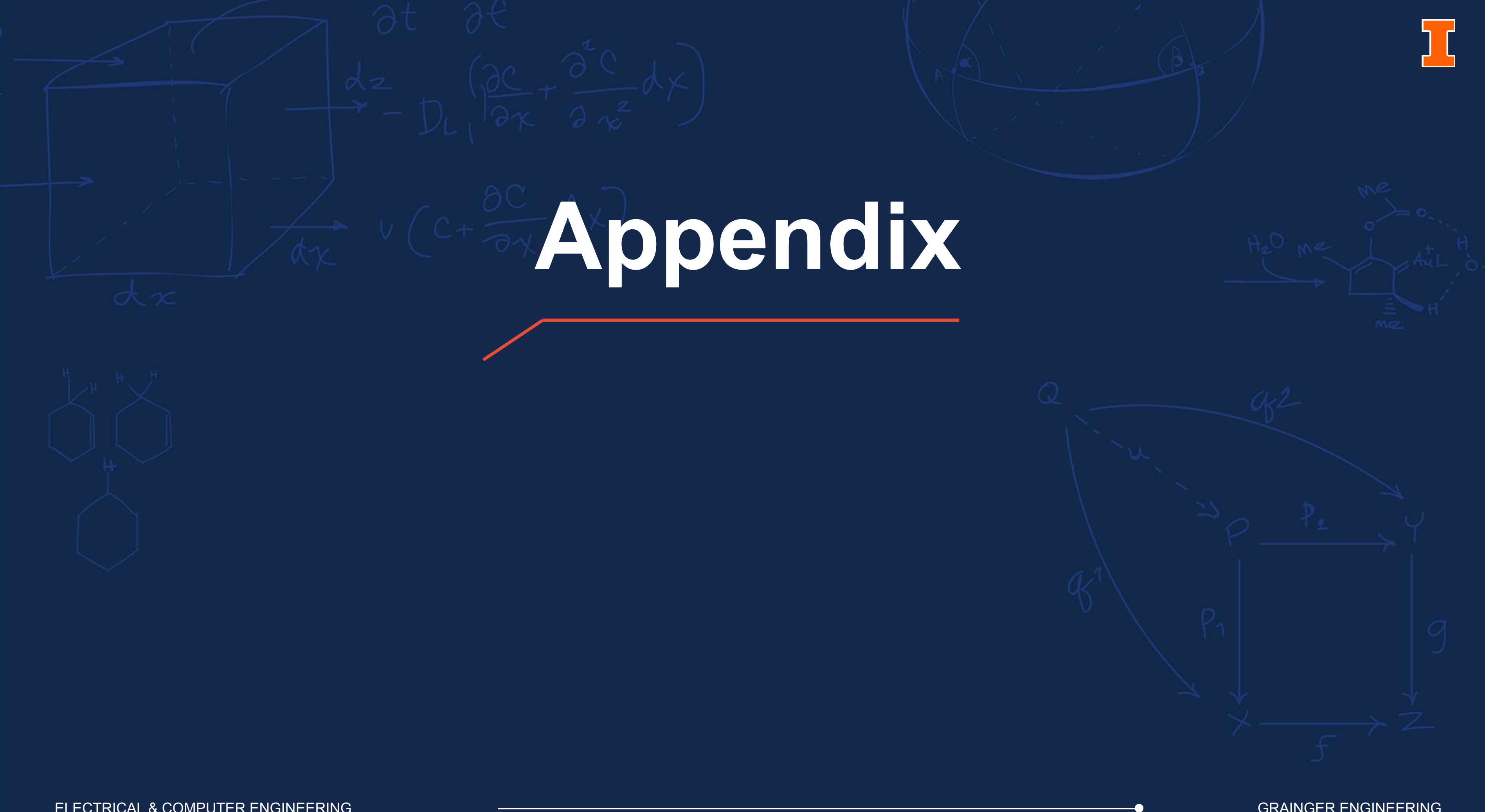
- Replace solenoids with wider bore solenoids that lessen the reduction in pressure
- Replace motors that have higher static pressure in order to play the harmonica
- Reduce noise from motors
- Ensure seal in joints to reduce pressure loss
- Reduce length of hosing and connectors to reduce airflow latency

Computing Subsystem

- LAN support rather than self-host AP
  - Don’t have to disconnect from the internet
- Integrate a touchscreen and more physical buttons for easier usability

What we learned

- PCB Design using microcontroller
- Robust motor driver design
- API based MCU control
- Web design
- Flaws in early faulting when testing
- Mechanical complexity of a harmonica



Appendix

FLOW RATE CALCULATIONS

From solenoid datasheet: with 500CC volume, it takes less than 5 seconds to exhaust the pressure from 300 mmHg to 15 mmHg.

Pressure difference: 285 mmHg=0.385 atmospheres
Volume of air exhausted: $V_{standard}$=500 cm^3 *0.385=0.01766*0.385=0.00662 Cubic feet of air at atmospheric pressure at sea level

Average flow rate: Q=6662/0.0833 minutes (5 seconds) =0.0795 SCFM (standard cubic feet per minute)

Using subsonic flow rate equation, we can relate flow rate coefficient to average flow rate to average pressure and temperature.

$$-\frac{V}{P_{atm}}\frac{dP}{dt} = 16.05 \cdot C_v \cdot \sqrt{\frac{P^2 - P_{atm}^2}{T \cdot S_g}}$$

Subsonic gas equation

Flow rate coefficient: C_v=0.0137

$$\int_{P_{final}}^{P_{initial}} \frac{1}{\sqrt{P^2 - P_{atm}^2}} dP = \int_0^t \frac{16.05 \cdot C_v \cdot P_{atm}}{V \cdot \sqrt{T \cdot S_g}} dt$$

Solution to subsonic gas equation

Solenoid Datasheet Pump Datasheet

FLOW RATE CALCULATIONS CONTD.

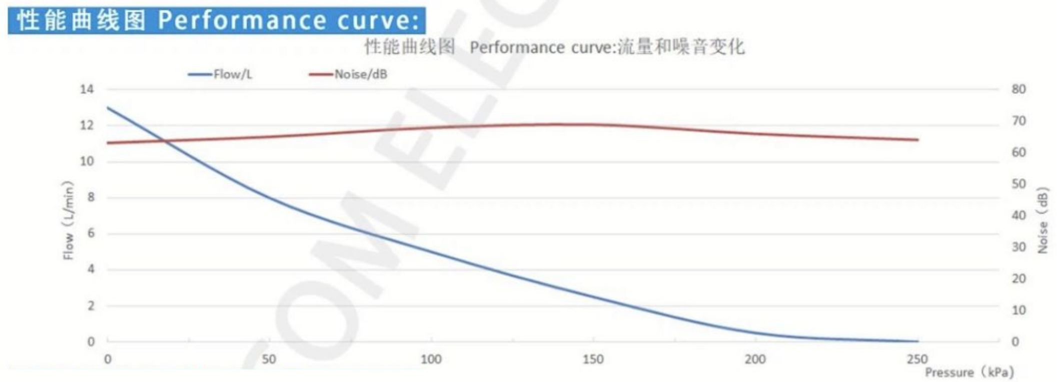
Through one solenoid:

Flow rate can be calculated using the subsonic gas equation

Equilibrium point: 80 kPa backpressure, **flow rate of 6 LPM.**

$$Q = 16.05 \cdot C_v \cdot \sqrt{\frac{P_1^2 - P_2^2}{T \cdot S_g}}$$

Subsonic gas equation for constant flow rate Q



[Solenoid Datasheet](#) [Pump Datasheet](#)

FLOW RATE CALCULATIONS CONTD.

Through two solenoids:

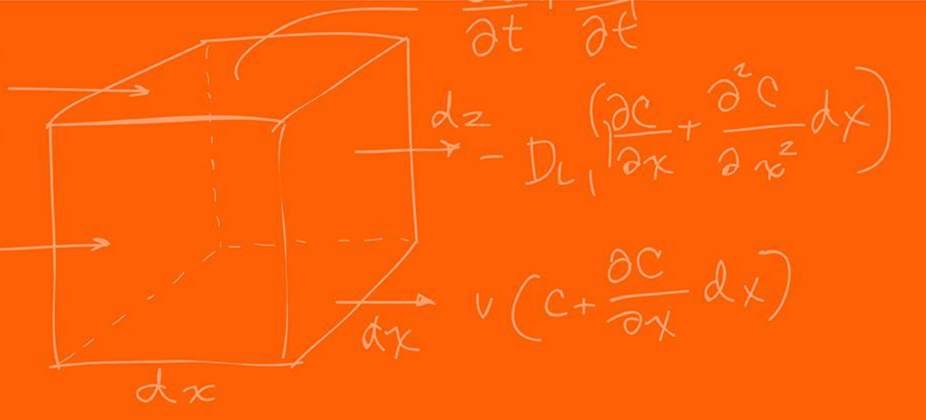
We can calculate an effective C_v value of the entire system by using the following equation:

$$\frac{1}{C_{v(sys)}^2} = \frac{1}{0.0137^2} + \frac{1}{0.0137^2}$$

$C_{v\ (sys)}$=0.00969

Equilibrium point: 102 kPa backpressure, **flow rate of 4.9 LPM**

Solenoid Datasheet Pump Datasheet



The Grainger College of Engineering

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN