Abstract

In high-rise apartment buildings, there are many pipes as part of the supply and drainage system for potable water. Leaks in these pipes result in a waste of usable water and money, especially when they are internal. In this project, we define “internal leaks” to be leaks that result from continuous, wasteful water flow rather than resulting from broken pipes or valves. Internal leaks are hard to detect as they do not leave an evident pool of water or drip loudly. By tracking water flow through a certain pipe over time, and looking for a consistent presence of water flow through that pipe, we can detect leaks in the system. Two previously designed prototype systems have done this: one with a water submetering system that fits into a pipe and one with an external analog acoustic detection system. The latter system influenced our design, which is a digital prototype that records water flow data using MEMS microphones. We process the audio signal on the Raspberry Pi, an embedded computer, to determine whether or not water is flowing through a pipe at any given time and transmit that data to an online database. The most important elements of our complete circuit design are the Raspberry Pi, the analog output MEMS INMP411 microphones, an op amp, and an AD converter. Testing our system demonstrated that it can clearly differentiate between the absence of water flow and the presence of a constant water flow. However, it struggles to categorize the absence of water flow and the cases where just a trickle of water runs through the pipe. Extending this project to transfer the programs from the Raspberry Pi to a dedicated audio microcontroller and to put our circuitry on a PCB would improve the system by making flow detection more accurate.
**Table of Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background and Theory</td>
<td>2</td>
</tr>
<tr>
<td>List of Materials</td>
<td>4</td>
</tr>
<tr>
<td>Design Process</td>
<td>4</td>
</tr>
<tr>
<td>1. Hardware Design</td>
<td>5</td>
</tr>
<tr>
<td>a. Key Components</td>
<td>5</td>
</tr>
<tr>
<td>b. Final Circuit Diagram</td>
<td>7</td>
</tr>
<tr>
<td>c. Serial Interfacing</td>
<td>8</td>
</tr>
<tr>
<td>d. Analog Microphones</td>
<td>9</td>
</tr>
<tr>
<td>e. Testing the INMP411</td>
<td>12</td>
</tr>
<tr>
<td>f. Analog to Digital Conversion</td>
<td>13</td>
</tr>
<tr>
<td>2. Software Design</td>
<td>15</td>
</tr>
<tr>
<td>a. record.py</td>
<td>15</td>
</tr>
<tr>
<td>b. filter.py</td>
<td>16</td>
</tr>
<tr>
<td>c. threshold.py</td>
<td>19</td>
</tr>
<tr>
<td>d. installation.py</td>
<td>20</td>
</tr>
<tr>
<td>e. send_data.py</td>
<td>20</td>
</tr>
<tr>
<td>f. Task Scheduling</td>
<td>21</td>
</tr>
<tr>
<td>3. Modularization</td>
<td>22</td>
</tr>
<tr>
<td>Results</td>
<td>22</td>
</tr>
<tr>
<td>Discussion</td>
<td>27</td>
</tr>
<tr>
<td>1. Analysis of Results</td>
<td>27</td>
</tr>
<tr>
<td>2. General Design Analysis</td>
<td>28</td>
</tr>
<tr>
<td>Realistic Design Constraints</td>
<td>30</td>
</tr>
<tr>
<td>Conclusion</td>
<td>31</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>31</td>
</tr>
<tr>
<td>Bibliography</td>
<td>32</td>
</tr>
<tr>
<td>Appendix: Code</td>
<td>32</td>
</tr>
</tbody>
</table>
Background and Theory

In high-rise apartment buildings, there are many pipes as part of the supply system for potable water. Often there will be a water tank in the basement connected to a city water main and a pipe from the tank that supplies water to the whole building. For each floor of the building, a riser pipe travels up from the basement to that floor. From there, distribution riser pipes split off to the different apartments on the floor. Leaks in these pipes result in a waste of usable water and money. In New York City, superintendents (resident managers) check up on a building every day and would notice a dripping leak, such as a broken pipe creating puddles in the basement. There are other types of leaks, however, that are less obvious. They result from running taps or loose valves often within apartments, which cause water to constantly and unnecessarily flow through the system. We call these “internal” leaks, as the water stays within the pipe, but clean, potable water is wasted as in an external leak. In a normal, leak-free apartment building, we would expect intermittent bursts of water use, concentrated in the morning and evening, corresponding to showers, toilet flushes, kitchen sinks, and so on. There should be peak water usage time periods and periods with little to no flow, such as during the early morning hours. If there is a pipe that has water flowing through it continuously over the course of a day or more, we may suspect one of these internal leak. In New York City, losing one half gallon per minute costs over $1,300 per year (“NYC”). Ideally, building managers and superintendents would be able to use a monitoring system to track water flow throughout the building and use the data to assess the probability of an internal leak.

Working towards this goal, Swarthmore alumnus Tom Sahagian developed a water submetering system that uses standard domestic water meters installed on cold water riser pipes. By tracking water flow over time as explained above and sending the data to a website, the system reveals if and when the flow stays constant and above zero, and thus where there is a high probability of a leak. While this submetering system works, it has several disadvantages. The meter must be installed within the pipe, meaning that a plumber must be hired to temporarily turn off water delivery, cut the pipe, and install the meter. For technical reasons, the meter cannot be used on hot water risers. Also, the cost for a 10-meter system is $2500 per installed meter. Finally, it can only identify if a riser has a leak and thus only the general location of the leak. A better water monitoring solution would be something small that anyone could mount external to a pipe without the expense or need of a plumber. Also, keeping the devices cheap and easy-to-install is key for an effective solution. If a specific apartment is causing the leak, having just one sensor in the basement is not very useful. By mounting sensors on the distribution pipes that connect to each apartment, a superintendent could quickly localize the leak.

With plans to improve this system, Tom Sahagian contacted Professor Carr Everbach to design an acoustic sensor that could live external to the pipe and detect water flow within the pipe. Professor Everbach and student Omodayo Origunwa ’18 designed a system which used two microphones: one to pick up the sound of the water flow in the pipe and the other to record ambient background noise around the pipe. They used two analog microphones in conjunction with an analog filter circuit that subtracted the background noise from the acoustic signal in the
pipe. This prototype achieves the design goal of making something small and easy to install, as it sits externally on the pipe and requires no costly plumber.

The circuit logic determined the presence of water flow in a pipe if the sensor detected a significant acoustic signal in the pipe. It did this by analyzing two audio signals. The idea was to have one microphone lying against the pipe, “listening” for water flow. However, this signal would also include some ambient background noise. The second microphone accounts for this problem, as it is oriented away from the pipe and picks up that background noise. Professor Everbach and his students made the simplifying assumption that the background noise from this signal would match the noise in the original signal, which is why the first step of circuit logic subtracts the “noise” signal from the “water” signal. Ideally, this step results in one signal that carries the true sound of water flow through the pipe. This signal is then passed through a filter that attenuates any frequencies outside of the 2500-3500 Hz range, which Everbach determined to be the band most relevant for water data by analyzing flow recordings of New York City pipes. Afterwards, the amplitudes of the signal were compared to an empirically determined threshold to differentiate water noise and ambient room noise. For each value higher than the threshold (that is, for each non-zero output of the comparator), a counter incremented. This data then traveled over WiFi to a central database. Since the counter would increment while water flowed and level off when the water flow stopped, a long period of monotonically increasing values would indicate a leak. The greatest flaw to this system is its inflexibility, as the comparator threshold needs to be set by hand on the physical device using a potentiometer.

While this analog circuit went a long way towards achieving the various design goals, we proposed a digital version to create a device more robust, accurate, flexible and compact. Ideally, a digital acoustic sensor would allow for more accurate measurements, better adaptability to the varied acoustics of pipes, and faster, more useful data processing.

Figure 1: Block diagram of Professor Everbach’s analog prototype. One microphone lies against the water pipe while the other sits a few inches away recording background noise. Both signals are passed into an op amp, then filter, then comparator, which inputs to the counter.
List of Materials

<table>
<thead>
<tr>
<th>What We Tried</th>
<th>What We Kept: Final List</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry Pi</td>
<td>Raspberry Pi</td>
<td>$35</td>
</tr>
<tr>
<td>Raspberry Pi WiFi dongle</td>
<td>Raspberry Pi WiFi dongle</td>
<td>~$9</td>
</tr>
<tr>
<td>Monitor, keyboard, mouse, and 32GB SD card for RPi</td>
<td>SD card</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
<tr>
<td>Smartphone portable charger for RPi</td>
<td>Smartphone portable charger for RPi</td>
<td>None (borrowed from Professor Everbach)</td>
</tr>
<tr>
<td>INMP441 flex board</td>
<td>-</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
<tr>
<td>Electret mic (26MPFET; Electronix Express)</td>
<td>-</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
<tr>
<td>INMP411 flex board</td>
<td>INMP411 flex board</td>
<td>4 x $80 = $320</td>
</tr>
<tr>
<td>MCP3008 (ADC)</td>
<td>MCP3008 (ADC)</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
<tr>
<td>MCP3002 (ADC)</td>
<td>-</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
<tr>
<td>Adafruit Perma-Proto Boards</td>
<td>Adafruit Perma-Proto Board (1)</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
<tr>
<td>Various analog components (TLV2772 op-amps, resistors, and capacitors; also wires)</td>
<td>Various analog components (TLV2772 op-amps, resistors, and capacitors; also wires)</td>
<td>None (previously owned by Engineering Department)</td>
</tr>
</tbody>
</table>

Design

We adapted the design of the analog prototype developed by Professor Carr Everbach and collaborators to design the digital prototype described in this report. Our prototype tracks the presence or absence of flow over 30 minute intervals using the dual microphone design used in the analog prototype. If a leak is present, there should be water flow detected at every time interval. Without a leak, certain times of day, such as the middle of the night, should have absence of flow detected that match residential water use patterns. To determine if there is water flow, we use the audio input signals from each of the two microphones to determine the signal of the water flow by itself. Once we remove noise and recover the water signal, we must decide if the signal indicates the presence or absence of water flow by thresholding. The
relevant results are then stored within a MySQL database for easy interfacing. Our solution outfits a small, fully digital computer, a Raspberry Pi, with two analog-output MEMS microphones. We also designed a circuit that amplifies and converts the microphone output to a digital signal before sending it to the Raspberry Pi. Digital signals provide the opportunity for more precise noise filtering and flexibility while developing software. The device outputs a decimal that represents the fraction of the signal determined to have detected water flow to our database every 30 minutes.

![Diagram of the digital prototype design.](image)

**Figure 2:** The digital prototype design.

In the rest of the Design section, we will first cover the hardware specifications and circuit design of our solution. Next, we will discuss the software control involving the Raspberry Pi Python programs and MySQL database.

1. **Hardware**
   
   In this section, we describe the key components of our hardware setup and how these pieces fit into the final circuit diagram. Next, we explain the process of our hardware design, including selecting components given serial interfacing protocols and why we chose to use the INMP411 analog output MEMS microphone. Finally, we discuss the analog to digital conversion that is key for analyzing the audio signals with the Raspberry Pi. The following subsections, which match the outline in the Table of Contents, cover these topics in order.

   a. **Key Components**

   The key elements of our hardware setup include the MEMS microphones, a low-pass anti-aliasing filter, amplification circuitry, the analog-to-digital converter, and finally the Raspberry Pi 2 Model B. The overall goal of the hardware was to integrate MEMS microphones with the Raspberry Pi.
The Raspberry Pi is a small $35 computer. We chose to work with it for this project because we wanted to understand how to program this popular device, which we have not otherwise had the opportunity to learn. Also, the Raspberry Pi has Python pre-installed on the operating system to allow for easier algorithm implementation. We also knew we wanted to use a computationally heavy digital filter, so having the higher computing power of the Raspberry Pi than of a microcontroller was advantageous. As the first step of our E90, we needed to set up our Raspberry Pi 2 Model B unit. We followed the quick start guide at the official website to format the 32 GB SD card and download the Rasbian operating system for the unit (“Raspberry”). We created a workstation for the Raspberry Pi in Hicks by connecting it to an available monitor, ethernet connection, keyboard, mouse, and power supply. Once the Raspberry Pi booted up for the first time, we were able to change the default password (for security reasons) and update our software over the internet connection. Our login now has username “pi” and password “E90ms.”

After completing the general Raspberry Pi setup, we had to configure it for our project. Anticipating the need to modularize, we set up the WiFi dongle to connect to the SwatDevice network so we could later send data to the database remotely over WiFi. Again, we followed the advice on the official website to do this (“Setting”). Our WiFi dongle provided our Raspberry Pi with an IP address of 130.58.108.77. Also, to enable the connection to the “SwatDevice” network, we had to provide the corresponding WPA2 version 1 password “DeviceSecure”.

The MEMS microphones are another important hardware component in our design. We planned to use the INMP441, a digital output MEMS microphone, as we were able to extract data from it using a Freedom development board in the semester prior to this project. MEMS devices - “microelectromechanical systems” - are micro-scale machines that are used in many engineering applications, especially as sensors in embedded systems. MEMS microphones, which come in both digital and analog output versions, are the perfect size for this sort of project. Furthermore, MEMS manufacturing techniques are advanced enough to achieve detailed hardware specifications for these devices.
### b. Final Circuit Diagram

Here, in figure 4, we have our final circuit design. The diagram shows the flow of our circuit logic except for the last step, where the MCP3008 connects to the Raspberry Pi via the Serial Peripheral Interfacing (SPI) protocol.

![Circuit Diagram](image)

**Figure 4:** Final circuit diagram. Vcc = 3.3 V and ground provided by the Raspberry Pi. The low pass anti-aliasing filter (at the output to each microphone, with R = 4.3kΩ and C = 0.01μF) has a cutoff frequency of 3700 Hz. Also, the op amps are biased around Vcc/2, thus the voltage divider circuitry in the middle of the diagram.

We initially built this circuit on a breadboard and then soldered it onto a compact perma-protoboard. Ultimately, the circuit would be best on a custom PCB to avoid any soldering mistakes or noise from wires. As displayed by the circuit diagram, there are many analog components in this circuit, both passive and active. Though we designed our prototype to be as digital as possible, these analog components were necessary to get decent results from the analog microphones and to turn those results into digital signals.

The following tables outline the pin connections between the Raspberry Pi and the protoboard circuit (Table 1) and the connections on the MCP3008, which connects the protoboard circuit to the Raspberry Pi over a hardware SPI connection (Table 2).
Table 1: The connections between the Raspberry Pi and the protoboard in our final design.

<table>
<thead>
<tr>
<th>Raspberry Pi Pin Number &amp; Description</th>
<th>Connection to Protoboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 3.3 Volt Power Source</td>
<td>Power rail</td>
</tr>
<tr>
<td>6. Ground</td>
<td>Ground rail</td>
</tr>
<tr>
<td>19. SPI MOSI (GPIO10)</td>
<td>MCP3008 pin 11 (Din for SPI)</td>
</tr>
<tr>
<td>21. SPI MISO (GPIO9)</td>
<td>MCP3008 pin 12 (Dout for SPI)</td>
</tr>
<tr>
<td>23. SPI SCLK (GPIO11)</td>
<td>MCP3008 pin 13 (CLK for SPI)</td>
</tr>
<tr>
<td>24. SPI CEO (GPIO8)</td>
<td>MCP3008 pin 10 (CS for SPI)</td>
</tr>
<tr>
<td>33. GPIO13</td>
<td>LED voltage source</td>
</tr>
</tbody>
</table>

Table 2: The circuit connections to the MCP3008 in our final design.

<table>
<thead>
<tr>
<th>MCP3008 Pin Number &amp; Description</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. Channel 4</td>
<td>Op amp output 1 (TLV2772 pin 1)</td>
</tr>
<tr>
<td>7. Channel 6</td>
<td>Op amp output 2 (TLV2772 pin 7)</td>
</tr>
<tr>
<td>9. DGND, digital ground</td>
<td>Protoboard ground rail</td>
</tr>
<tr>
<td>10. CS/SHDN, chip select</td>
<td>RPi pin 24</td>
</tr>
<tr>
<td>11. Din, digital input</td>
<td>RPi pin 19</td>
</tr>
<tr>
<td>12. Dout, digital output</td>
<td>RPi pin 21</td>
</tr>
<tr>
<td>13. CLK, clock</td>
<td>RPi pin 23</td>
</tr>
<tr>
<td>14. AGND, analog ground</td>
<td>Tied to pin 9</td>
</tr>
<tr>
<td>15. Vref, reference voltage</td>
<td>Tied to pin 16</td>
</tr>
<tr>
<td>16. Vdd, voltage source</td>
<td>Protoboard power rail</td>
</tr>
</tbody>
</table>

c. **Serial Interfacing**

Though our final circuit uses the analog-output INMP411 microphone, we initially planned to use the digital-output INMP441, as previously mentioned. Once we had the Raspberry Pi working, our next step was to interface it with the INMP441 microphone so that we could read in the data directly; however, setting up this serial communication interface was more complicated than we anticipated. The INMP441 outputs digital data using the Inter-IC Sound (I2S) protocol which is a serial communication protocol developed specifically for audio applications. It has a fairly complicated and nonstandard timing rule in that the most significant bit is read on the second rising edge of the clock signal, rather than the first. Unfortunately, the Raspberry Pi 2 Model B does not natively support I2S. However, it does natively support Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C). SPI and I2C are two generalized popular communication protocols, and both are more widely used and accessible than I2S. By natively support, we mean that the Raspberry Pi makes it easy to interface with devices that use these protocols by providing dedicated GPIO pins and storing libraries to control communications over these interfaces. The Raspberry Pi has hardware GPIO pins that are already configured to communicate over SPI or I2S if enabled. Confusingly, the Raspberry Pi also seems at first glance to support I2S if enabled, and we were surprised that such a flexible computer with so many GPIO pins would not support this protocol. I2S is even mentioned in a configuration file in the Raspberry Pi’s home directory, but we found that this may have been an artifact from an older revision of the board we do not have, one which made I2S more plausible. Ultimately, we determined that there was no clear way to enable I2S on the Raspberry Pi 2.
Model B. We also considered the MSP430G2553, a microcontroller we are familiar with, to see if that supported I2S. Since it did not, we decided to carry on with the Raspberry Pi.

Without native hardware support, the only way to interface with the INMP441 would be to bit-bang the I2S protocol on the Raspberry Pi side, which would involve setting the timing specifications using software rather than hardware. We briefly considered this, but the protocol is too fast to easily bit-bang and would have taken more time than was appropriate for the project. We decided that we would have to communicate with the Raspberry Pi using a protocol other than I2S. We decided to do an initial test of the SPI bus by connecting an SPI-enabled sensor. We chose the ADXL362, which is a MEMS accelerometer that communicates over SPI and was available in the Engineering Department. We connected the ADXL362 to the SPI-dedicated hardware pins on the Raspberry Pi header (pins 19, 21, 23, and 24). Then we needed code to pull the data. After some research, we decided to install SPIdev, a library for using SPI on the Pi (“SPI”). We also had to enable SPI on the Pi itself (Hawkins). We found sample code that tested the SPIdev library and were success in reading data from the ADXL362 (Vorontsov). This gave us confidence that we could communicate with the Raspberry Pi with an SPI-enabled device.

d. Analog Microphones

Since we decided to continue with the Raspberry Pi, we could no longer use the I2S-output INMP441, and we had to get a different microphone. Since our research revealed no microphones with digital SPI output, we decided that we could use an analog output microphone and ultimately convert that analog signal to a digital signal using an SPI-enabled Analog to Digital Converter (ADC) that would communicate with the Raspberry Pi. Since the Raspberry Pi is a fully digital system, it does not accept analog inputs because it has no native ADC capability. To connect an analog output, we needed to feed that output into an ADC that could easily communicate with the Raspberry Pi over a supported protocol. We discuss this more in the following subsection, Analog to Digital Conversion. The most readily available analog output microphone was an electret microphone owned by the Engineering Department, so we tried this first. We tested its frequency response using an oscilloscope. Once we figured out the correct component values for the surrounding circuitry, the microphone worked and we could see a clear response to whistling or speaking into it on the oscilloscope (figure 5).

![Figure 5: Breadboard circuit for electret microphone ("capsule"). We set it up with a 2.2 kΩ resistor to V+, a 0.1 µF capacitor to the output, and a power supply of V+ ≈ 3.00 V. At this point, the output was connected to an oscilloscope.](image)
Unfortunately, the electret microphone signal was noisy. The quality was too low for our project. We also tried connecting the electret microphone circuitry to the Raspberry Pi via the MCP3008 ADC over the Raspberry Pi SPI pins, adding in a TLV2772 op-amp to deal with the noise (figure 6). Here, we were more successful. The Python script on Raspberry Pi reported ADC values around 500, the midpoint. The ADC has values from 0-1023 corresponding to an input range of 0-3.3 V. Since the microphone output was biased around 1.67 V, as shown in the figure 6, a “zero” output would correspond to 512 on the ADC. With the microphone output connected to both the ADC and an oscilloscope, we whistled into the microphone. The oscilloscope clearly showed sine waves in response, and ADC channel changed value by about 30-40. It was at this step in the design process that we began including the TLV2772 op amp in every circuit.

![Figure 6: Electret Microphone circuit diagram. Here, the analog electret is connected to the Raspberry Pi via an op amp and the MCP3008 ADC. Vcc and ground for each component came from the 3.3V power supply and ground pins on the Raspberry Pi. The closed loop gain of the op amp is 10. Though not shown in this diagram, the ADC is connected to the Raspberry Pi.](image)

Since we were successful in reading values from an analog microphone with the Raspberry Pi, we decided to order an analog microphone that was higher-quality than the electret. Since we had already worked with MEMS, we decided to get the analog output MEMS version of the digital one we had been working with before: the INMP411, as opposed to the original INMP441. The INMP411 has the same high performance specs and tiny form factor as its I2S alternative.

![Figure 7: Side-by-side comparison of the electret microphone and INMP411 microphone (not to scale). The electret has two leads (combining the power line and output) and ~1cm diameter.](image)
The INMP411 has 3 leads (ground, power, and output) and has a 4.72mm x 3.76mm footprint. Here you can see the surface-mount part connected to the evaluation flex board which separates out the 3 pins.

For another comparison between the electret and the INMP411, we took data using the example Adafruit ADC code running on the Raspberry Pi. We saved the digital ADC output into csv files so that we could graph the results. We did this first for the electret microphone when we had it and then a couple months later once we had successfully connected the INMP411. Below is a side-by-side comparison of the output graphs for each microphone corresponding to whistle input.

**Figure 8, left:** Data from 2/20/16. We whistled into the electret microphone with code running on the Raspberry Pi. This graph came from the file generated by the Raspberry Pi code.

**Figure 9, right:** Data from 4/10/16. We whistled into one of the INMP411 microphones on the protoboard with code running on the Raspberry Pi. This graph came from the file generated by the Raspberry Pi code.

As evident in these graphs, the ADC output for the INMP411 covers a greater range than the output for the electret (about 450-600 for the INMP411 rather than ~460-500 for the electret). This means that the INMP411 is more sensitive to acoustic changes than the electret microphone, which made it the better choice for our project. Since whistles are much louder than water flowing through a pipe, we knew that we needed a very sensitive microphone.

We were able to run the code with the INMP411 soldered into a simple circuit, which is described in the section beginning on the next page.
e. Testing the INMP411

Once the INMP411 analog output MEMS microphone arrived, we decided to test it with a simple circuit. We soldered it to a small protoboard that allowed us to connect the microphone output initially to an oscilloscope input channel and eventually to a Zoom recorder as well (see schematic below). The INMP411 pinout is very simple, with a power line, ground line, and output line. While the INMP411 is only available as a surface mount part, we got the evaluation flex board, EV_INMP411-FX, which allowed us to easily access these pins. We placed a 180Ω resistor in parallel with a 0.1μF capacitor between the power line and the power source, as well as a DC-blocking capacitor at the output, as recommended by the datasheet. We also included a 3.0 V lithium ion battery so that we could avoid including the Raspberry Pi right away.

Once we saw a response from the microphone on an oscilloscope, we decided to test it with real water data using a Zoom recorder. At the beginning of this project, one of our biggest unknowns was the characteristics of water flow through a pipe; we did not have any intuitions about the volumes or frequencies that might be present. Thus we did this test both to explore the sensitivity of the INMP411 and to develop intuition about what our water data would look like. We decided to several recordings against a pipe at various water flow rates. The Zoom recorder enabled us to record data with the INMP411 at a 96000 Hz sampling rate and save the results as a .wav file. Later, we downloaded the file to a computer for analysis with MATLAB.

For the test, we brought the microphone-Zoom setup to Hicks basement, where we recorded real water flow through a pipe whose water flow we were able to control. We held the microphone up to the outside of the pipe while adjusting the flow through it with a valve. Over the course of 4.5 minutes, and starting at no flow, we increased the flow rate every 10 seconds. This resulting in 27 different flow rates total, ending with the maximum amount of flow (the valve was opened all the way). We then saved the resulting .wav file and uploaded it to MATLAB, where we made the graphs below.

Figure 10: Schematic of the battery-powered circuit in which the INMP411 output is fed into the Zoom recorder. This circuit was soldered to a protoboard.
You can see from the above graph that increased flow does not linearly correspond with amplitude. The third flow rate is by far the loudest, and as the flow rate increases, the volume decreases until it hits a steady value for most of the tests. This surprised us, as we had generally been expecting a linear relationship. However, the graph still shows that all of the flow rates result in higher amplitudes than the initial value (the amplitude resulting from zero water flowing through the pipe). We also analyzed the frequency spectrum of this data using a spectrogram. This revealed little to no frequencies above 20 kHz, with the majority of the frequency content in the 0-5 kHz range. This confirmed the findings from the analog prototype, which attenuated signals outside of the 2.5-3.5 kHz frequency band. Also, this test was successful in demonstrating that our chosen microphone would pick up the sort of data we were interested in (various different water flow rates as recorded from the outside of a pipe).

Confident that we wanted to use the INMP411 in our final design, we ordered a second evaluation board for this microphone. We tested this second microphone in a more complete version of our final circuit, connecting it to a TLV2772 op-amp that outputs to one of the channels of the MCP3008 ADC, which connects to the Raspberry Pi. Once we had this circuit working, and once the Zoom recorder tests were done, we desoldered that INMP411 from the Zoom protoboard (above image) and made a circuit using both mics. This is the circuit in Figure X (our final circuit diagram).

f. Analog to Digital Conversion

After the successful Zoom recorder test, we needed to get the INMP411 talking to the Raspberry Pi. As previously mentioned, the decision to use an analog-output microphone also meant that we needed an analog to digital converter, as the Raspberry Pi programs would all require digital signals. We decided to use the 10-bit MCP3008 ADC because it is well-documented; has Adafruit tutorials with sample code for use with the Raspberry Pi; has multiple channels (we need one for each microphone); and communicates over SPI, which we
got working with the ADXL362 accelerometer (DiCola, “Raspberry”). Also, a MCP3008 was already available in the Engineering Department.

To acquire data from the microphone and view it on the Raspberry Pi, we ran Python code on the Raspberry Pi that frequently checked the ADC values coming in over the hardware SPI configuration (Raspberry Pi GPIO pins 8, 9, 10, and 11). In our final design, we read from the ADC in our record.py code, which is described in the Software Design section. This code uses the Adafruit MCP3008 Python library available from Github and builds off the sample code provided with this library (DiCola, “Adafruit”). This example code is designed to communicate with the MCP3008 over SPI, so it was perfect as a starting point for our project. The code has a while loop that reads all channels of the ADC every X seconds, where we can adjust the value of X. We tested the code without any inputs to the ADC to confirm that it was working and then added both INMP411 microphones as inputs to separate channels.

Before our database was up and running, our “record” code saved the ADC values into a csv file. From the Raspberry Pi interface, we could email these csv files to ourselves and then open them as spreadsheets in Google Sheets. From there, we could graph the data over time. This allowed us to run simple tests on our setup to see whether or not the code was working. We would run the code, whistle into the microphone, stop the code, and observe graph resulting from the saved csv file. These simple graphs gave us an idea of what sort of data our database would ultimately store.

Finally, we measured the amount of time it took to read one value from the MCP3008 ADC using the Adafruit library code. While the MCP3008 datasheet lists a sampling rate of 75 kHz, this value is much lower in practice when running on the Raspberry Pi with Python code. We found that one read takes 0.000135 seconds, which corresponds to a data acquisition rate of 7500 Hz. By the Nyquist-Shannon sampling theorem, this allows us to read signals with a maximum frequency of 7500/2 = 3750 Hz. While the empirical sampling rate of the MCP3008 is far below the Zoom recorder sampling rate of 96000 Hz, it is enough to get the range of frequencies we are interested in, as previously stated. However, this step also taught us that we needed to include a low-pass filter for anti-aliasing. Since we can only accurately read signals with a maximum frequency of 3750 Hz, we must attenuate all frequencies about that value at the analog microphone output, before amplifying and digitizing the signals. Thus, at this step we added the low-pass anti-aliasing filters after each INMP411 output, each with cutoff frequency of 3700 Hz (see final circuit diagram for RC values).
2. Software Design

We have six Python scripts on the Raspberry Pi that comprise the software program. We chose to keep the scripts modular for easy adjustment and design alterations as we worked. The gray box on the left represents the hardware circuitry. The red box represents the Raspberry Pi. The programs within are Python scripts run by the Raspberry Pi. First, we have the installation.py program that runs once to initialize the system during installation. Then we have water_main.py, the blue box in figure X, which calls the other Python programs when it runs every 30 minutes. This main program calls record.py to measure the data and get the signals from the two microphones, filter.py to remove the noise and recover the true water flow signal, threshold.py to determine if that water flow represents the presence or absence of flow, and send_data.py to store the data in a MySQL database. The MySQL database lives on Professor Carr Everbach’s Fubini server and holds the tables to store the necessary data.

![Block diagram of the software components for our prototype.](image)

Figure 12: Block diagram of the software components for our prototype.

a. record.py

The first script in the main program is record.py. This recording program is based on started code provided by Adafruit for interfacing with the MCP3008 ADC as described above (DiCola, “Raspberry”). This provided the libraries necessary to send data from the ADC to the Raspberry Pi via SPI. It also provided some starter code that we edited to write the program we needed for our prototype. We only had two microphones, so we needed to use and read from just two of the MCP3008’s 8 output channels. See the appendix for this code. The general structure of the script is a while loop that repeats and reads from the two desired ADC channels in every iteration to build up the data that is the signal from each of the two microphones. This program takes 100000 data points from each microphone. At the sampling rate of 3750, this program just under 30 seconds to run and gather data. For ease of use while testing, we added an LED onto the protoboard that would turn on while the record.py program runs to indicate when data is being recorded.
b. `filter.py`

By choosing to implement a digital design, we then needed to program a filter algorithm as well. As described previously, our design centers around the two microphones that in combination can detect water flow. The first microphone that is attached to the pipe will record background noise in addition to any sound produced by water flowing through the pipe. The other microphone directed away from the pipe will record the ambient background noise present in the surroundings. In order to determine whether there is water present in a pipe, we need to find the difference between the signal containing both pipe and noise and the signal for the background noise. We cannot simply subtract the background signal from the pipe signal for several reasons. First, there will be a time delay between the two signals because the microphones are not in the same position, so a certain sound signal will reach the microphones at different times. Second, the background noise recorded within the pipe signal may not include the exact same sounds as the background recording. Again, due to the separation between the microphones, signals appear differently at each.

To accomplish this digital filtering, we chose to implement a Wiener filter. The Wiener filter works on a signal with additive noise by minimizing the mean squared error between the estimate and the desired signal. Often, uses of the Wiener filter involve removing noise to determine a desired unknown signal (Shimkin). Figure 13 displays how the inputs and outputs interact with the filter to result in an estimate for the desired signal, where table 3 relates the signal variable names in figure 13 with corresponding signals in our system. For our filter, the `y` signal is the pipe recording, the `s` signal is the water sounds, and the `n` signal is the noise within the pipe recording. The `x` signal is the background noise recording made by the other microphone. We feed the background noise signal into the Wiener filter, which then outputs an estimate of the background noise signal. This noise estimate can then be subtracted from the pipe recording to give an estimate of the water flow signal. To get this noise estimate, we must multiply the noise signal from the background microphone by some matrix of coefficients, or weights. The optimal matrix of weights will give us the best estimate of the noise within the pipe microphone’s signal.

![Figure 13: The diagram of the Wiener filter.](image-url)
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Signal in our System</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>Input signal from pipe microphone</td>
</tr>
<tr>
<td>( x )</td>
<td>Input signal from background microphone</td>
</tr>
<tr>
<td>( s )</td>
<td>Water flow signal</td>
</tr>
<tr>
<td>( n )</td>
<td>Ambient noise</td>
</tr>
<tr>
<td>( n_{est} )</td>
<td>Estimate of the ambient noise</td>
</tr>
<tr>
<td>( s_{est} )</td>
<td>Estimate of the water flow signal</td>
</tr>
</tbody>
</table>

**Table 3.** The corresponding signals for the abbreviations above.

The following equations walk through the process of finding the optimal filter weights (Ifeachor). We try to find an estimate of the desired signal \( s \) in equation 1 where the estimate of the noise signal is as shown in equation 2. Equation 3 is the equation for mean square error where \( P \) is the cross-correlation vector and \( R \) is the autocorrelation matrix for \( x \). The Wiener-Hopf Equation is equation 5 and displays how to calculate the optimal filter coefficients to get the best estimate for the noise. The optimal filter weights are at the bottom of the mean square error surface in the figure where the gradient is equal to zero. Figure 14 shows what the MSE surface looks like for a system with two filter weights \( W(0) \) and \( W(1) \). To find the \( W(0) \) and \( W(1) \) that minimize the error, we set the gradient of the error \( J \) equal to 0 as shown in equation 4. We can then solve this equation to get the optimal weights.

\[
\begin{align*}
    s_k^\hat{} &= y_k - n_k^\hat{} = s_k + n_k - n_k^\hat{} & \text{(Equation 1)} \\
    n_k^\hat{} &= \sum_{i=0}^{N-1} w_k(i)x_{k-i} & \text{(Equation 2)} \\
    J &= \sigma^2 + 2P^T W + W^T RW & \text{(Equation 3)} \\
    \frac{dJ}{dW} &= -2P + 2RW = 0 & \text{(Equation 4)} \\
    W_{opt} &= R^{-1}P & \text{(Equation 5)}
\end{align*}
\]
Figure 14: Surface of the mean-squared error $J$ shown for a 2D system with two filter weights (Ifeachor).

Programming this algorithm proved to be rather simple. We chose to code the initial program in MATLAB because matrix operations are often simpler in MATLAB than in Python's numpy. We chose to solve for the optimal weights using a least squares solution to the problem. Alternatively, we could have solved for the optimal weights using gradient descent which is less computationally demanding. However, we had the benefit of using the Raspberry Pi and Python, so we chose to use the least squares approach to the problem because it does not require any extra data gathering to tune the filter weights on while the system is running like gradient descent does. We also needed to account for the number of filter weights needed for this system. We used the sampling rate of 3.75 kHz to do so. For this prototype, that only required three filter weights. Thus, we programmed the optimal filter with the pseudoinverse and saw that the filter performed well on test data. Given a voice recording with additive noise of a vacuum cleaner, the filter recovered the initial voice signal closely as seen in figure 15. As the legend shows, the blue signal is the original voice and noise signal ($y$). The green signal is the original noise signal ($x$). The red signal is the recovered estimate of the desired voice signal ($s_{est}$). After seeing this success, we transferred the algorithm into Python to be able to run it directly on the Raspberry Pi after data collection. The filter takes in two numpy arrays that each store the signal from each of the microphones. It then outputs a numpy array that is the estimate of the water flow signal itself by doing the filtering process described above.
c. threshold.py

Given the water flow signal output from the Wiener filter program, we need to then determine if that signal displays the presence or absence of water flow. To begin, we simply want to make a binary decision to indicate if water is flowing through the pipe or not. At first, we considered just seeing if the signal every drops below a determined threshold for no flow and reporting that the signal has the absence of water flow. However, for a signal like the one shown in figure 16, this method loses information about the signal that could be beneficial to store.

To store more of the signal’s information, we chose to implement a more informative filter. For each point, we determine if it is above or below the threshold and increase a counter for values above the threshold or a counter for values below the threshold respectively. Thus, once we have analyzed the whole signal, we know how much of the signal was above as compared to below the threshold. We can then return a fraction that represents the number of data points above the threshold out of the length of the signal. This number corresponds to the fraction of the signal for which water flow is detected. To make this metric more robust, we chose to use the root mean square which represents the power of the signal at a given point. For each point in the signal, we calculated its RMS using the surrounding 150 data points. This
measurement is less affected by outliers in the data, making it more robust for comparisons. The desired value to use as our threshold may vary slightly from pipe to pipe or from room to room, so we wanted to find a way to learn the threshold for each installed device rather than to just detect the signal. Thus, we added the installation.py script to account for this and to find the desired threshold for no flow in the given location, also using an RMS measurement.

d. installation.py

As mentioned earlier, installation.py is outside the loop of water_main.py. A key element of our design is this installation code, which must be run when setting up the system. It was crucial to us that we did not empirically determine the threshold value against which we test our system in the water_main.py loop. We wanted to find a value that accounts for the noise inherent to our system for the estimated water flow signal when there is no water flowing through the pipe. Then, every recorded value above that value can be considered above the threshold of “no flow” and thus evidence of water flow. Since threshold.py compares the RMS values of our signals against the threshold value, we must find that threshold value using RMS. The installation.py code runs record.py while no water is flowing through the pipe, filters the signal with the optimal_filter.py code, and then analyzes the output. The threshold value is then the power (the RMS) of the resulting signal estimate from the filter for these no flow signals. Once installation.py finds the threshold, it writes it to a file. From that point onward, the water_main.py code reads from that file each time it runs and uses the stored value in the threshold.py step.

This installation.py code makes our system a flexible solution, in that installation.py is run for each pipe that gets a sensor. Thus we account for differences amongst the many possible pipes one may encounter in an apartment building, determining a new customized threshold for each pipe. Also, the installation.py code helps achieve our design goal of providing an easy-to-install system. This code must be run when there is no water flow through the desired pipe. While it can be inconvenient to shut off the water supply to a pipe, it is only necessary for one minute, which is the runtime of installation.py. This is much easier than shutting off the water supply long enough to cut open a pipe and install an internal flow meter, and then sealing the pipe again before water is restored. Running installation.py requires no user input either and is initialized to run on the reboot of the Raspberry Pi, which will occur during the installation of the device onto the pipe. Furthermore, we would like to expand the system so that installation.py can be called remotely over the WiFi connection to make this process even easier, and repeatable in case a less-than-desirable threshold results from the first run of installation.py.

e. send_data.py

Another task associated with this project is the setup for a database to store the collected data. In order for this device to be useful, people must be able to track the water flow through pipes over time to determine when and where leaks may be present. We plan to allow for this by sending data from the Raspberry Pi over WiFi to the database. Each recording will be filtered and thresholded. The resulting decision as well as the time will be sent to our database.
for reference later. To make this database, we are using a MySQL database hosted on Professor Carr Everbach’s Fubini server. We have one table which stores three values: a pipe id indicating which pipe the data is recorded from, a time stamp, and the decimal that represents the fraction of the signal for which water flow was detected as an float. We later added several other categories to the database, including the number of points in the signal above and below the threshold as well as the basic threshold output determined by seeing if any point in the signal falls below the threshold. After processing, this entry will be added to the database through the MySQL API Python protocol. From this data, we can easily make a user interface like a website on which users can track the water flow through various pipes over time. This tool will allow users to visualize the water flow and notice which pipes might be leaking. Figure 17 demonstrates what an app or website might display to the user. In this example, there are 4 devices on risers going to the basement, 1st floor, 2nd floor, and 3rd floor. The blue dots indicate water flow, meaning that the decimal returned from our system was high enough to be the presence of water flow in that pipe. Thus, a user can quickly glance at this and see that because the first floor has water flow at every time interval, there may be a leak on the first floor.

![Figure 17: Visualization of how a user might view the output of our system to help them identify the location of leaks within a large building.](image)

f. Task Scheduling

We use the native Raspberry Pi task scheduler, Cron, to schedule our code. This allows us to modularize our system rather than having to log into the Raspberry Pi with a workstation every time we want to run code and calling it from a terminal. Cron allows users to create “Cron jobs” that run automatically based on a set schedule. We set up Cron to call the water_main.py code once every half hour, every day of the week. We also added installation.py as a Cron job that is called at reboot, which is the only “run one time” option provided by Cron. Installers can cause installation.py to be called by forcing a Raspberry Pi reboot (disconnecting and reconnecting the power source). Thereafter, water_main.py will be called every half hour, record data, and send it to the database. Unfortunately, due to permissions, getting Cron to interface with MySQL proved to be difficult. One alternative method is to call water_main.py from within an infinite while loop using Python time.sleep to choose the recording interval of 30 minutes.
3. Modularization

In addition to modularizing the code with Cron, we aimed to make the Pi more portable, which is essential for an embedded sensor that will sit on a pipe. We started working with the Raspberry Pi with a busy workstation in Hicks with a monitor, keyboard, mouse, and power cable to the wall. Cron replaces the monitor, keyboard, and mouse, but we still need a way to power the Raspberry Pi. We decided to use a simple, portable smartphone charger, which provides the right voltage and plugs right into the Raspberry Pi without needing extra circuitry. Also, these chargers are cheap and easily available.

We also designed a 3D-printed case for the Raspberry Pi (figure 18). This keeps our components together and allows us to secure the prototype to a pipe. There is room in the case for the Raspberry Pi, the portable charger (power source), and our custom protoboard housing our final circuit. We would add a foam insert between the Raspberry Pi and the protoboard to avoid shorting any sensitive circuit parts together and to keep them from moving around and getting damaged. There are holes for each of the two microphones to stick out of the case, one of which sticking down to be close to the pipe and one that is oriented away from the pipe out into the surroundings. Finally, there are loops on the bottom of the case that allow for zip ties or velcro strips to pass through and tighten around the pipe for secure fastening. The front of the case is removable to be able to access the prototype and is screwed into place when closed.

Figure 18: Two renderings of the 3D-printed case design. This shows the fastening loops on the bottom and the hole for the pipe microphone.

Results

The following section will show the data we took with our prototype. We tested the device on a pipe in the basement of Hicks on the Swarthmore College campus. We held the prototype against the pipe for testing as shown below in figure 19. As figure 19 displays, our testing involved having the microphones oriented correctly, one lying just next to the pipe and the other microphone about 5 inches away from the pipe. While running tests, we watched the LED to know when the prototype was recording. The final prototype would be held in a plastic box which would be secured to the pipe as described in the Modularization section above.
Figure 19: An image of our prototype during testing against a pipe in the basement of Hicks.

<table>
<thead>
<tr>
<th>Flow Rate</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
<th>Trial 6</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>No flow</td>
<td>0.6411</td>
<td>0.5227</td>
<td>0.6643</td>
<td>0.5675</td>
<td>0.6768</td>
<td>0.5210</td>
<td>0.599</td>
</tr>
<tr>
<td>Stage 1 flow (low)</td>
<td>0.5310</td>
<td>0.5417</td>
<td>0.5899</td>
<td>0.6577</td>
<td>0.5969</td>
<td>0.8250</td>
<td>0.624</td>
</tr>
<tr>
<td>Stage 2 flow (med)</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.00</td>
</tr>
<tr>
<td>Stage 3 flow (high)</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9999</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>Variable flow</td>
<td>0.6286</td>
<td>0.6690</td>
<td>0.5230</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.607</td>
</tr>
</tbody>
</table>

Table 4: Our results from testing our system on the pipe in Hicks basement. Running installation.py at the start of this test resulted in a threshold of 3.303 for all the trials. For the 6th trial, we would add additional background noise (more than ambient) by talking close to the microphones. The values in the table cells represent the percentage of flow detected.

As shown in Table 4, our flow rates are not strict values, but relative. We kept the flow rate constant for each set of trials and then increased it for each subsequent set of trials. We did not measure the exact flow rates for each trial because our system differentiates between the presence and absence of water flow rather than finding values of the flow rates. Thus, when testing, we were curious about the prototype’s ability to detect flow at various conditions rather than precise flow conditions. We tried three types of “variable flow.” For variable flow trial 1, we recorded data with no water flowing through the pipe for the first five seconds, and then steadily increased and then decreased the amount of flow. The second variable flow trial was similar; we started with five seconds of no flow and then did an increasing ramp with a slight decrease at
the end. Finally, the third trial of variable flow started recording during the maximum water flow through the pipe. Then we monotonically decreased the flow until it stopped, with 7-10 seconds of silence at the end of the recording. Overall, we made each variable flow signal have water flowing at some point and no water flowing at other points. Also, we added louder background noise during the sixth trial to ensure that the optimal filter handles loud background noise the way we expect it to. Since basements in cities may hear loud noises from streets or driveways, the filtering of loud noises as well as constant quieter ones is essential for our system to work.

Figures 20 to 27 show the detected flow signal where the original signal is the signal from the microphone against the pipe, original noise is the signal from the microphone away from the pipe, and recovered is the estimate for the water flow signal. The y axis is a digital value read by the Raspberry Pi from the ADC on the scale of -512 to +512. The x axis is time in seconds. The figures (20 and 21) depicting trials from the no water flow cases show a range of about -25 to 35. Figure 21, when the no flow case was recorded with loud background noise, shows the same trends as figure 20 but with one noise spike that is filtered from about 60 down to 50 through the filter. Figure 22 for the low flow case demonstrates the slightly louder signal as expected although with a similar range of -25 to 40. Both trials for the medium flow case show a range of -30 to 40 even with loud background noise (Figures 23 and 24). Figures 25 and 26 for the high flow case also have ranges of -30 to 40 for both quiet and loud background noises. For Figure 27 as the representative figure of the variable flow cases, the signal stays between -25 and 30 like with low flow and no flow cases but when the flow increases, the signal also increases to between -30 and 40 as we expect given the range of values for the high flow trials.

![Figure 20](image1.png)

**Figure 20:** Filter output for no flow trial 3.

![Figure 21](image2.png)

**Figure 21:** Filter output for no flow trial 6.
Figure 22: Filter output for stage 1 (low) flow trial 4.

Figure 23: Filter output for stage 2 (medium) flow trial 1.

Figure 24: Filter output for stage 2 (medium) flow trial 6.
We also decided to test how our filtering system compares to a simple subtraction of the noise signal from the pipe signal. During the same experiment, we ran our same program without changing the threshold. Instead of using the filter script, we simply subtracted the background noise microphone’s signal from the pipe microphone’s signal. The rest of the setup and software run was identical to the above trials. The results for this are in the table below:
### Table 5: Results from the test using subtraction as the filtering method. During Trial 2, we added additional background noise by talking near the mics.

<table>
<thead>
<tr>
<th>Flow Rate</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>No flow</td>
<td>0.9997</td>
<td>0.9995</td>
<td>0.9996</td>
</tr>
<tr>
<td>Stage 3 flow (high; same level as stage 3 flow in table 3)</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

### Discussion

In this section, we first discuss the results from our trials of the final system that were presented in the previous section. Then, we discuss our component choice, and how the final elements of our hardware have both pros and cons for this particular application. We also address the general takeaways we have as well as our ideas for extensions of this project.

#### 1. Analysis of Results

Overall, the prototype performs well. There is a clear difference between the results of the system using a pure subtraction of the background microphone’s signal from the pipe microphone’s signal and the results using our optimal filter method. When this subtraction is used to find the estimate of the true water flow signal, the no flow trials detect water flow to be present 99.96% of the time, and the high flow (stage 3) trials detect 100% flow regardless of the level of background noise (table 5). When run with the optimal filter code, our prototype provides more sensible results. Our system detects the presence of water flow 59.9% of the time on average in the signal for the no flow trials, whereas it detects water flow in the signal 100% of the time for the high flow (stage 3) cases (table 4). Thus, the optimal filtering is better than pure subtraction because the subtraction method results in a reading of 100% presence of water flow regardless of whether water flow is present or absent. Our filter code, however, does return different results depending on whether water flow is present or absent during the recording.

Our final design, with the optimal filter code, proved to perform well when detecting the various cases of water flow we hoped to capture. For the trials with no water flow, the prototype detected water flow for 59.9% of the signal on average for the six trials (table 4). This result means that our system detected that 59.9% of the signal has an RMS above the threshold determined at installation. As discussed previously, the program considers being above the threshold as the presence of water flow. This number may seem high, but looking at figures X and X shows that the water flow signal actually hovers above the threshold value of 3.03 for about half of the signal. Thus, this high percentage may actually be due to the variation of the signal itself rather than the noise introduced by our system.

The most important feature of this prototype is its ability to differentiate between no water flow at all and some amount of water flow. When we ran trials on stage one flow, which was just a trickle of water, we got a similar percentage of detected water flow at about 62.4% (table 4). Thus, the system is not high-resolution enough to differentiate between the absence of water
and a slow trickle. With adjustments that could include increasing the sampling rate or getting equipment dedicated to audio, the filter may perform better and help the system differentiate between no flow and low flow. However, the system performs very well on medium (stage two) and high (stage three) flow rates, detecting an average of 100% water flow for both flow rates. For the three trials with variable flow, we had sections of several seconds with no flow and then constantly changing flow rates for the rest of the measurement. Our system detects 60.7% presence of water flow within a signal on average for the variable flow trials. While we have some sections of no flow during each trial, we expected this percentage to be higher. Figure 27 shows that the variable case looks similar to the no and low flow cases as the numbers indicate. This result shows that our system has trouble differentiating no flow from this variable flow.

Examining all the flow rates tested, our system does perform differently between cases of no to low flow and cases of medium to high flow. Thus, we can simply normalize our prototype to be able to determine what we will consider the presence or absence of flow. We can advise people using the system to consider results below 65% detected presence of water flow to be no flow. As previously stated, this will also cause very low trickles to be considered “no flow.” Thus, we can separate the results determined from the various flow cases to correctly identify them. Additionally, we see in figures 20-27 that the ranges of values recorded in the signal cover about 70 values of the possible 1024. This low resolution may contribute to some of the problems in differentiating between different flow cases. Figure 20 displays a no flow range of -25 to 35 whereas the high flow range is about -30 to 40 in figure 25. This difference is only a change of about 10 in range. This is small given that the digital values are on the scale of -512 to +512. If the system had a higher resolution without losing definition, we may be able to detect the differences between no flow and low stage one flow cases better.

When there is significant background noise added, the system still performs well. Figures 21, 24, and 26 when compared respectively to figures 20, 23, and 25 show that the recovered signal estimate of the water flow does not change much for the trials when there is loud background noise present during a recording. The no flow, stage two flow, and stage three flow cases do not exhibit significant change in flow detection due to loud background noise (table 4). However, the lowest flow case for stage one flow has water flow presence detected on average for 62.4% of the signal but with loud background noise it jumps up to 82.5% presence of water flow in the signal. Despite this increase meaning that the recording would be considered to have water flow present, the real conclusions we can draw are limited. The other trials for the same stage one flow rate are below what we would consider to be no flow (65% as explained above). Therefore, this trial’s higher percentage of detected flow may in fact be due to a worse filtering of the loud background noise out of the pipe signal. Our prototype is not adequate to detect very low or variable flow rates, though it does differentiate between cases of no flow and cases of medium or high flow.

2. General Design Analysis

Throughout the project, we learned the pros and cons of what options are available for this application. We tried both electret and analog MEMS microphones, and we found the MEMS to provide a cleaner signal, as expected. However, the electret had simpler circuitry and a lower dynamic range. While this is usually a negative in audio applications, our E90 deals with
a small dynamic range. The range of amplitudes produced by water flowing is actually quite small, so we would rather decrease dynamic range while increasing resolution. The InvenSense MEMS microphones advertise their large dynamic range - up to 131 dB - which is probably far higher than necessary for this project. While the MEMS microphone was preferable to the electret for our project in terms of its signal to noise ratio (SNR), we may have had more success (and perhaps avoided the use of an op-amp) with a microphone with a lower range and higher resolution.

In terms of microphones overall, we still would have liked to use one with a digital output over I2S. If we had found a way to interface with the Raspberry Pi, or another microcontroller, our circuit would have been free of analog components, which would have reduced several sources of error and additive noise. As mentioned earlier, we had to adjust our circuit to include analog components once we began using analog-output microphones. We only saw useful data once we added an op amp in an inverting configuration, which provided the necessary gain to amplify the signal. This is in addition to a decent number of resistors and capacitors, which increased the number of analog elements needed. Unfortunately, analog components provide a source of error as they have non-trivial tolerances. Also, avoiding analog components would simplify the design by removing the need to decide what resistor and capacitor values to use, how to set the op amp gain, and so on. However, a low-pass anti-aliasing filter would still have been necessary to account for the system’s sampling rate.

In another version of this prototype, we might switch out the Raspberry Pi rather than the microphone. We could have found a microcontroller specifically designed for audio applications, something that would support the I2S processor and high sampling rates necessary for recording audio. However, for this project we stuck with the Raspberry Pi because we wanted to learn the system and have the ease of writing our algorithm in Python. A dedicated audio microcontroller may have provided better results but would have been less documented and more difficult to learn during the short timeframe of this project.

In addition to the presence of analog components, a source of noise may be the amount of wires and hand-soldering in our protoboard. If we had the time to order a custom printed circuit board (PCB) after solidifying our final circuit design, we would have used the surface-mount INMP411 rather than the evaluation flex board. Our protoboard is closer to a breadboard circuit than a board with surface-mounted parts as it uses entirely hand-soldered through-hole parts. However, it was crucial for our design process to use the evaluation flex boards for the microphones as we kept making changes throughout the semester. Also, we needed the ability to control the location of the microphones during testing. If we were to mass-produce this sensor, however, we would use the surface-mount INMP411 because it is much cheaper than the $80 flex board INMP411. Also, if we used surface-mount parts overall, we would reduce the need for wires and thus reduce noise and possible error resulting from soldering by hand.

Our current system uploads data to a server over WiFi, providing one-way communication between the sensor and the server. However, we realized that we could use this wireless capability for more than we initially intended. We could build a website and allow two-way communication so that users (such as the building’s superintendent) could look at the data being sent from the sensor and make edits to the system. Most importantly, we would allow
for the ability to change the threshold value or trigger another run of installation.py from the website. This eliminates the need for the superintendent to go down to the physical basement and force a reboot of the Raspberry Pi to run installation.py again and reset the threshold. In case the first run of installation.py results in a less-than-desirable threshold, allowing users to remotely run the code again allows for easy re-calibration. Also, we could send battery data to the website so that it could be monitored remotely as well.

Another idea for the future is mass-producing this product to allow superintendents to install a system with many of these devices, one for each pipe they are interested in monitoring. However, there are some issues we would need to address to scale up the project. As previously mentioned, we would have a lower-cost PCB to save time and money, rather than recreating the hand-soldered board we have now. Also, we would need to consider how to enable the WiFi for each unit. Our unit is currently connected to Swarthmore College campus WiFi, but part of setting up the system in a new building would involve installing routers to make sure all monitoring devices have a reliable WiFi connection with the appropriate credentials.

We would need to test the battery life of the smartphone charger, too. While it powers the Raspberry Pi well, it might not last long enough to power the Raspberry Pi for days without a recharge. In a full-scale system, we could use a smaller, lower power microcontroller that could survive on less power than the Raspberry Pi uses. As described above, using a microcontroller may also improve the performance of the system as well as the battery life. Choosing a microcontroller with a robust low power mode and interrupt routines would prove best for deploying these devices.

Finally, it might be interesting to test this system with a different filtering or thresholding method. Our prototype uses an optimal filter, which we have tested to be more effective than pure subtraction. Another possible choice would be an adaptive filter, which is a “digital filter with self-adjusting characteristics” (Ifeachor, 645). Adaptive filters are usually built with least mean square or recursive least squares algorithms. Using an adaptive filter would allow us to vary the filter characteristics (its frequency band), which is useful for this application as the interference (noise signal) may vary in frequency range with different pipes or over different time periods. The main difference between our optimal filter and an adaptive filter is the feedback configuration of an adaptive filter, which allows the filter to adjust its coefficients during processing. Another way to threshold would be by frequency. Our threshold.py code currently thresholds based on the power of the signal in the time domain. We could also try to threshold by the frequency of the signal. This would probably be most effective on a microcontroller designed for audio processing, however, with a dedicated DSP chip for fast Fourier transforms. As with the testing done for this prototype, we would have to confirm that we pick up different frequencies when there is no water flow and when there is water flow.

Realistic Design Constraints

While we have extensions for this project, there were realistic design constraints that restricted the scope of the project. We only had one semester to complete the design and testing, so time was a constraint. Although we have ideas to improve our design, we do not have the time to continue and implement these changes. We were also constrained by our budget of $500, limiting the types of equipment we could incorporate into this design. Our
design goals overall were to build a small, cheap, and easily-installable device, so these factors also limited our design in terms of components and cost. Though the device would use energy, we hope that the device would work well enough to offset its energy cost with the amount of water it would save when installed to monitor water leakage.

Conclusion

In high-rise apartment buildings in cities like New York City, internal leaks cost money and waste potable water. Our goal for our final E90 project was to build a system to monitor pipes in such buildings to detect these leaks. Throughout the course of the semester, we designed and built a prototype that demonstrates our end-to-end system design. Using two MEMS microphones, we record the sounds through a pipe and process that data to detect the presence or absence of water flow in the pipe during that recording. With this information, we can track the presence of water flow in a pipe over time to ultimately determine whether or not a leak might be present in a certain location within the building, which would be indicated by consecutive outputs indicating the presence of water flow. Ideally, this sensor would be installed on every pipe in a high-rise building to localize the cause of the leak if one is found. Our results show that our system clearly differentiates between readings of zero flow and readings of positive flow, so our design works for this application. The prototype struggles to differentiate between variable flow, low flow and no flow. While we have built a proof-of-concept prototype, our final circuit design could be developed on a custom printed circuit board for mass-production. Also, we have tested that our prototype improves on the previous analog prototype, as our optimal filter clearly performs better than pure subtraction of the noise signal from the pipe signal. We think that with a microcontroller designed for audio capture, we could have even better results for detecting water flow and perhaps increase the resolution of our results to differentiate between different levels of water flow (low, medium, and high).

Finally, the Raspberry Pi, the microphones, and the protoboard with our final circuit will stay in the Engineering Department for future use. Learning how to use a MEMS microphone with the Raspberry Pi in this way could be applied to future projects.

Acknowledgments

We would like to thank first and foremost our advisors for this E90, Professors Erik Cheever and Carr Everbach. Both were instrumental in guiding us through this process and helping us with various design decisions and hardware issues along the way. We would also like to thank Ed Jaoudi of the Engineering Department for his endless patience supplying us with electronic components and helping us make the hardware work. We thank Professor Matt Zucker for helping us port the optimal filter algorithm from MATLAB to Python. Finally, we want to thank Tom Sahagia ’74 for providing the initial idea for this project and providing consultation and support throughout the semester.
Bibliography


Shimkin, N. The Wiener Filter. N.p.: Israel Institute of Technology, 2009. PDF.


Appendix: Code

installation.py
from optimal_filter import ofilter
from record import record
import numpy

def main():
    f = open('threshold.csv','w')
    # record the no flow data
    ch4, ch6 = record()
    ch4 = ch4 - 512
    ch6 = ch6 - 512
# filter this
signal = ofilter(ch4, ch6)
len = signal.shape[0]
rmsls = numpy.zeros(len-149)

for i in range(75, len - 74):
a = signal[(i-75):(i+75)]
rms = numpy.sqrt(numpy.mean(numpy.square(a)))
rmsls[i-75] = rms

thresh = numpy.mean(rmsls)
f.write(str(thresh))
print "value to threshold with: ", thresh
f.close()

main()

water_main.py
import sys
import datetime
import numpy as np
from optimal_filter import ofilter
from send_data import send_data
from threshold import threshold
from record import record
from basic_threshold import basic_threshold

def main():
f = open('mic_data2.csv','a')
# set pipe id
pipeid = 'pipe1'

# get date and time here
now = datetime.datetime.now()
print "today is ", now

# record signals here
ch4, ch6 = record()
print "record elapsed time is", datetime.datetime.now()-now
ch4 = ch4 - 512
ch6 = ch6 - 512
# filter signals here
signal_est = ofilter(ch4, ch6)

# signal_est = ch4 - ch6 # for when we want to test pure subtraction

# threshold here
flow, thresh, num_ones, num_zeros = threshold(signal_est)

# "flow" is the fraction of time that we register the pipe as having
# flow through it. So if flow = .90, there is flow through the pipe
# 90% of the time we're looking at it (though this does not distinguish
# between medium and high flow states). If flow is greater than .90,
# perhaps there is a leak.

# send to database here
send_data(pipeid, now, thresh, num_ones, num_zeros, flow, basic_flow)

print flow
main()

record.py
import numpy
import time
from readadc import readadc
import RPi.GPIO as GPIO
import math
import Adafruit_GPIO.SPI as SPI
import Adafruit_MCP3008

def record():
    mcp = Adafruit_MCP3008.MCP3008(spi=SPI.SpiDev(0,0))
    GPIO.setmode(GPIO.BOARD) #turn on indicator LED
    GPIO.setup(33,GPIO.OUT)
    GPIO.output(33,GPIO.HIGH)

    iterator = 100000 # takes about 20s. sampling rate=3700 Hz.
    i = 0
    ch4 = []
    ch6 = []

    while(i<iterator):

        # collect ADC data
# The read_adc function will get the value of the specified channel (0-7)

v4 = mcp.read_adc(4)
v6 = mcp.read_adc(6)
ch4.append(v4)
ch6.append(v6)

i = i + 1

# return array of values
np4 = numpy.array(ch4)
np6 = numpy.array(ch6)

GPIO.output(33,GPIO.LOW)  # turn off indicator LED

GPIO.cleanup()
return (np4, np6)

optimal_filter.py

import numpy as np
import matplotlib.pyplot as plt
import scipy.io.wavfile
from numpy.lib.stride_tricks import as_strided

def ofilter(signal, noise):  # ch4 is signal (short wires)

    # Get sampling rate
    rate = 3700

    # really really don't want to deal with integer division issues
    # anywhere below
    signal = signal.astype(float)
    noise = 0.1*noise.astype(float)

    ####################################################################
    # Make up a kernel

    kernel_dur = 0.001
    nw = int(rate * kernel_dur)

    rc = 0.0001
    tw = np.arange(nw,dtype=float)/rate
    kernel = np.exp(-tw/rc)
kernel /= kernel.sum()
kernel = kernel[::-1] #reverse

print "max: ", signal.max(), " ", noise.max()
print "min: ", signal.min(), " ", noise.min()

#################################################
# stupid numpy tricks:
# this will make X the matrix
#
# google as_strided
#
# [ x_0 x_1 x_2 ... x_nw   ]
# [ x_1 x_2 x_3 ... x_(nw+1) ]
# [ etc... ]

#################################################
# Now convolve noise with filter

# In matlab you would produce this with corrmxt or something
X = as_strided(noise, shape=(len(signal), nw), strides=(8, 8))

combined = signal

# 100 seems reasonable rather than zero
lambda_coeff = combined.var() * 100
print lambda_coeff

# solve our regularized least squares problem
#
# pseudoinverse of X is (X'X)^(-1) X'
# so solution looks like w = (X'X)^(-1) X' y
#
# we regularize by adding to the diagonal of X'X some constant which
# is within a couple orders of magnitude of the variance of the
# signal.
A = np.dot(X.T, X) + lambda_coeff * np.eye(nw)
b = np.dot(X.T, combined)

# w is kernel weights
w = np.linalg.solve(A, b)

# we are now re-filtering our noise, this time with the discovered
# weights w.

n_est = np.dot(X, w)

# recover the original signal by subtracting out the reconstructed
# filtered noise.
recovered = combined - n_est
print "recovered min and max: ", recovered.min(), recovered.max()

# write out our solutions
scipy.io.wavfile.write('combined.wav', rate, combined.astype('int16'))
scipy.io.wavfile.write('recovered.wav', rate, recovered.astype('int16'))

# plot some stuff

# threshold.py
import numpy

def threshold(signal):
    num0 = 0
    num1 = 0
    f = open('threshold.csv','r')
    no_flow = f.read()
    f.close()
    print "no flow is ",no_flow
    len = signal.shape[0]

    for i in range(75, len - 74):
a = signal[(i-75):(i+75)]
rms = numpy.sqrt(numpy.mean(numpy.square(a)))
if rms > float(no_flow):
    num1 += 1
else:
    num0 += 1
print "num0: ", num0
print "num1: ", num1

num_sum = float(num1)+float(num0)
percent_flow = float(float(num1)/num_sum)
return percent_flow, float(no_flow), num1, num0

**send_data.py**
import MySQLdb as db
import datetime
import random
import warnings

def send_data(pipeid, time_of, thresh, num_ones, num_zeros, flow, basic_flow):
    host = 'fubini'
    user = 'sbrakem1'
    password = 'e90pipe'
    dbname = 'E90PipeData'
    warnings.filterwarnings('ignore')
    con = db.connect(host, user, password, dbname)
    cur = con.cursor()
    try:
        create_table = ("CREATE TABLE IF NOT EXISTS pipetest "
                        "(pipe_id VARCHAR(10) NOT NULL, "
                        " time_of_entry DATETIME NOT NULL, "
                        " threshold FLOAT(10,6) NOT NULL, "
                        " num_ones INT(8) NOT NULL, "
                        " num_zeros INT(8) NOT NULL, "
                        " water_flow FLOAT(8,6) NOT NULL, "
                        " water_flow_basic INT(1) NOT NULL)")
        cur.execute(create_table)
    except:
        query = ("INSERT INTO failures (time) VALUES (%s")
cur.execute(query)

query = ("INSERT INTO pipetest (pipe_id, time_of_entry, threshold, num_ones, num_zeros, water_flow, water_flow_basic) "
    "VALUES (%s, %s, %s, %s, %s, %s, %s)")
print "pipe: ", pipeid
print "time_of: ", time_of
print "threshold: ", thresh
print "num_ones: ", num_ones
print "num_zeros: ", num_zeros
print "flow: ", flow
print "basic_flow: ", basic_flow

try:
cur.execute(query, (pipeid, time_of, thresh, num_ones, num_zeros, flow, basic_flow))
    con.commit()
except:
    con.rollback()
    query = ("INSERT INTO failures (time_of) VALUES (%s)")
    cur.execute(query, (time_of))
    con.commit()
    print "rolling back on (", pipeid, ",", time_of, ",", flow, ",", basic_flow, ")"

con.close()