Jarvis Voice Controlled Devices

John Costello, Jessica Louie, Edward McCloskey, and Kevin Pielacki
Rutgers Department of Electrical and Computer Engineering
Piscataway NJ, 08854

Keywords—dynamic time warping, distance matrix, averaging filter, socket programming

I. ABSTRACT

Jarvis is a system designed to respond to user issued commands to provide convenient control over a number of electronic devices. These devices can be lights, TV's, radios, stereos, etc. The system will be designed to work best within a moderate home with the convenience of a wireless router. Fig. 1.1 shows how each stage of the system will interact with each other. More details on this are provided below. The system will take a voice input from a user, match that input to an appropriate command within its library of recognized commands or reject the command if it's not recognized by the system and transmit an appropriate message via a router. The router will then send the operation to the proper device from so the operation can be performed.

The voice processing stage purpose of this stage is to convert a user voice recording into information convenient and usable for the rest of Jarvis. The initial phase of this step is to obtain a recording of what the user’s voice command. This will be accomplished through a user interface via a button press where upon initiation Jarvis will begin to record for a brief period. The period should be long enough to accommodate the entire voice message of the user which will contain multiple words. Once the message has been recorded Jarvis must determine make comparisons to which word in the bank best matches the received message.

A key component of Jarvis lies in its ability to recognize commands using only verbal input from the user; after a brief training period Jarvis will be able to match whatever commands are issued with templates it stores during the training period. In order for this to work Jarvis must be able to identify individual words within a three or four word instruction; this task is accomplished using an averaging filter.

The final step of the voice processor will be a comparison between each input word of the user to other templates that store cases of the user’s input command. If the input closely matches a stored template then it is likely the word input is the same as the template. The problem that arises is the user will not speak a word the same exact way every time. The word will likely be spoken at a different pitch or speed than what the template contains. To take these variations into account matching is done through a dynamic time warping function. The dynamic time warping function allows comparison between two signals that vary in time or speed. This function warps the input signal non-linearly to compare each signal at different points in time with respect to each other. From there a path finding algorithm can be used to calculate a distance of how well these signals represent one another. The lower the calculated distance the better each signal represents one another. This function will be applied to each word spoken as well as each template that the word has been categorized to and return the minimum value to decide which word was likely to be spoken. This finally generates a bit sequence output that is passed onto the communication channel stage.

The Jarvis communication channel will be how the processor will speak to the connected devices. Socket programming was used in Matlab to have the processor talk to each device. Socket programming has two parts a server and client. The client listens to a specific port on the server’s IP address. The server opens the port the client is listening to and sends data into it. The client will then take in the data the servers sends along the port. The processor in Jarvis is the server for the socket programming. The devices will be GUI’s which represent different devices and are the client for the socket programming. Each device will be listening to different ports and the processor will decide which device the command was for and will send it to the appropriate port. The GUI’s will then processes the message and perform the function within the message. For example if the user tells the TV channel to go up the GUI will display the channel going up. The devices, just like in real life, can have buttons change the status of its functions. The message is a 40 byte message. The first five bytes represent the devices meaning there is a max of 32 devices connected at once. Each device has a unique name. The next 30 bytes represent all the possible commands a device can receive. This means the total amount of unique commands can be $2^{30}$. The last five bytes represent the action for the device to take. So if the user inputs “TV Volume Up.” The first five bytes will represent the TV, the next 30 will represent Volume, and the last five will represent up. The socket programming did all of this correctly sending the command without error and properly changed the device’s
status. Future work that could be done with the socket programming is to have the devices constantly listen for messages from the port while also having the ability to have the devices changed using buttons. Matlab presented this limitation in the program since it cannot perform two operations at the same time. Another future work that can be done is to have the devices communicate back to the processor to let the processor know when something has been changed on the device by the user pressing a button.

II. INTRODUCTION

The method chosen for the voice recognition process of Jarvis is template matching. The motivation behind template matching is simple where a comparison of an input signal is made with a library of predefined templates. This method was chosen due to Jarvis is designed to contain a relatively narrow, in comparison to the commonly spoken words in the English language, library of words. The templates are obtained through the user giving a sample of each command before Jarvis is ready to process any user commands. The comparison of the template to the given command should return some measured result that decides how similar the two compared signals are to each other. The idea may be simple however Jarvis is expected to take a command input from a user's voice and find its best match from the library of commands. A more detailed discussion of this approach will be discussed in section 3.

A key component of Jarvis lies in its ability to recognize commands using only verbal input from the user; after a brief training period Jarvis is able to match whatever commands are issued with templates it stores during the training period. In order for this to work Jarvis must be able to identify individual words within a three or four word instruction; this task is accomplished using a moving average filter.

After the full string of three or four command words are issued the waveform generated by the users’ voice is stored in a vector; this vector holds the amplitude of each sample of the users’ voice. A sampling window is created in which laterally across the entire waveform, and if at any point a certain amount of samples contained in the window are above a predetermined threshold, defined by the noise floor of the received signal, the averaging filter will signal to Jarvis that it has found the beginning. When the average within the filter falls below the threshold again it marks this point in time as the end of the word. The window size must be large enough to overlook brief pauses within a user’s speech input as well as small enough to not mistake two closely spaced words for one large word.

The parent waveform has a large number of fluctuations, some minute and some huge. Upon inspection, it is a simple task for any person to mark the beginning and end of a word; the smaller changes in the graph are ignored and the larger are recognized as important information. In order for Jarvis to make these same distinctions the moving average filter is used; this filter smooths out the tiny imperfections and finds the changes it needs.

The filter tries to find large changes in amplitude, either increasing or decreasing; however, the bookends of each word are not the only amplitude changes present on the graph. Due to random noise and small pauses within the words themselves, a programmer cannot simply tell the computer to search for samples with an absolute value greater than zero. Jarvis has to search for a consistent chain of samples lying above the minimum amplitude. The window used by the filter slides from the first entry to the last, and at each iteration it checks each individual entry. If the entry examined lies above the threshold it is added to the sum which is then divided by the total number of entries checked.

Typically a moving average filter is used with a large sum of data to find large consistent changes within that data, so one would trying to find averages above fifty or sixty percent. Once the beginning or end of your trend has been identified more detailed analyses can be carried out. The primary goal of the moving average filter used in this project is to section out more specific bits of data so that an in-depth analysis can be performed.

The Jarvis communication channel will be how the processor will speak to the connected devices. This is an important part of the project because if the two parts do not communicate then nothing would happen. Since Jarvis is intended for within a home a method to have the devices and Jarvis communicate that would be sufficient is socket programming. Socket programming was used in Matlab to have the processor talk to each device. Socket programming has two parts a server and client which transmit data using TCP sockets. The client essentially listens to a specific port on the server’s IP address. This specific port corresponds to a specific device. The server opens the port the client is listening to and sends data into it. The client will then take in the data the servers send along the port. The processor in Jarvis is the server for the socket programming.

The devices will be GUI’s which represent different devices and are the client for the socket programming. Each device will be listening to different ports and the processor will decide which device the command was for and will send it to the appropriate port. The GUI’s will then processes the message and perform the function within the message. For example if the user tells the TV channel to go up the GUI will display the channel going up. The devices, just like in real life, can have buttons change the status of its functions. The message is a 40 byte message. The first five bytes represent the devices meaning there is a max of 32 devices connected at once. Each device has a unique name. The next 30 bytes represent all the possible commands a device can receive. This means the total amount of unique commands can be $2^{30}$. The last five bytes represent the action for the device to take. So if the user inputs “TV Volume Up.” The first five bytes will represent the TV, the next 30 will represent Volume, and the last five will represent up. The socket programming did
all of this correctly sending the command without error and properly changed the device’s status.

Future work that could be done with the socket programming is to have the devices constantly listen for messages from the port while also having the ability to have the devices changed using buttons. Matlab presented this limitation in the program since it cannot perform two operations at the same time. Another future work that can be done is to have the devices communicate back to the processor to let the processor know when something has been changed on the device by the user pressing a button. This could also be used for security devices to alert the system when someone is breaking into a house. Another issue with using socket programming is the fact that it would be taking up a chunk of bandwidth from the users Wifi, especially if the devices are changed to continually listening to the processor.

III. METHODS AND RESULTS

The method chosen for the two signal comparison is known as dynamic time warping or DTW. DTW is an algorithm used to measure how similar two signals that vary in time or speed are. When the two signals are passed through the DTW function a distance estimate is returned. One should not mistake the returned value as an actual distance because DTW is not always performed in a Euclidean metric. This metric is defined during the calculation of what is known as a distance matrix.

The type of distance calculation performed depends more upon the chosen application rather than exact measured values. Estimated distances can prove just as valuable as the exact values since the goal of DTW is to return how similar the two signals are to each other. The end result of the DTW function will be a number where the lower the value the more similar the two signals are. So an exact number isn’t as desired as a consistent quantity separation between similar and non similar signals. Listed below are distance formula equations that can be considered for the distance matrix calculation where T and R are vector representations of the signal:

Frame-by-Frame Distance Measure [1]:

\[ d(T, R) = \|T - R\| = \sum_{i=0}^{n} (T_i - R_i)^2 \]  

\[ (1) \]

Covariance Weighting [1]:

\[ d(T, R) = (T - R)^T \tau^{-1} (T - R) \]

\[ (2) \]

Where \( \tau^{-1} \) is the inverse of the covariance matrix.

Spectral distance [1]:

\[ d(T, R) = \int [\log[T(e^{j\omega})] - [\log[R(e^{j\omega})]]]^p d \omega \]

\[ (3) \]

Where “q” is usually an even integer. This assures that all difference values are positive which assures that the distance is returned as a magnitude. The integration is performed over the frequency range of interest. The range can be chosen as the range of a human voice since Jarvis is only interested in
human voice input however a more processing efficient method is to get a rough estimate of the band of frequencies excited by the voice command and limit the calculation to only that particular band. This does however introduce a new order of complexity due to background noise within the command recording may be mistaken as the band of interest.

LPC Log Likelihood Measure [1]:

\[
d(T, R) = \log\left(\frac{a_T V_T a_T^T}{a_R V_R a_R^T}\right)
\]

(4)

Where \( V_T \) is the matrix of autocorrelation coefficients, \( a_R \) and \( a_T \) are the LPC or linear predictive coding coefficient vectors.

Symmetrical Distance Measure [2]:

\[
d(T, R) = \sum_i (T_i - R_i)^2
\]

(5)

This is a simple difference calculation between the two vectors to be compared and a squaring the result. It also contains the benefit of maintaining positive. For the Jarvis voice processor this distance calculation method was chosen mainly due to low computation time since the operation only requires a subtraction and squaring operation. For simplicity, whenever any distance calculation is performed it will be done through equation (5) unless stated otherwise for the remainder of this document.

The distance matrix begins the process of quantifying the similarity of the two signals by performing a shift in time comparison of the reference and input signal. This process can be best demonstrated through a comparison of two discrete time signals that contain a single impulse at one value. Refer to Fig. 3.1 where the distance calculation is performed on these two signals.

![Fig. 3.1: Distance Matrix Calculation of Vectors T and R](image)

where:

\[
T_i = [0, 0, 0, 0, 1, 0] \quad \text{&} \quad R_j = [0, 0, 0, 1, 0, 0]
\]

Note that “i” and “j” are positive integer values that begin with a value of “1” and the boxed row r. The horizontally boxed values represent vector T while the vertically boxed values represent vector R. The resulting distance matrix is calculated the following equation:

\[
M_{(i,j)} = \sum_i \sum_j d(T_i, R_j)
\]

(6)

Where in this case “i” and “j” row and column indexes however “i” is incremented from the bottom of the matrix to the top. These unconventional indexing values are used because it is convenient to represent the path through the matrix as first quadrant on a coordinate plane. This path plays an important role on how these signals compare.

The diagonally boxed path contains a very interesting set of values. These values happen to be the distance calculation for when “i” is equal “j” or in other words the distance between the two signals without any shift in indexing. For example, if these vectors discrete indexes represented a moment in time then the diagonal represents the distance calculated at the same time difference from the start of each vector. Also notice that the bottom left value represents the distance calculation of the beginning entry of both vectors being compared while the top right represents the distance calculation for last entry of both vectors. Refer to Fig. 3.2 showing a comparison of the same vector that is:

\[ T = R = [4, 6, 15, 3, -4] \]

Note the zero value contained at the bottom right of the matrix is used as a placeholder so notice how when comparing the same vectors all the entries within the diagonal has a zero value. This diagonal is in fact the ideal path the signal can take and when the values contained in the path are all zero then it represents an exact match for the two signals. On the other hand when one signal contains more discrete values than the other or the signals are not a perfect match then a diagonal path alone cannot get to the endpoint at the top right of the matrix as shown in Fig. 3.3.

![Fig 3.2: Distance Matrix Calculation for the same Signal Comparison](image)

![Fig 3.3: Optimal Path when Comparing Vectors without a Matching Length](image)
Now take the sum of the values found in the boxed path in Fig 3.3. This value of 49 is actually the desired return value for the DTW function. This value returned is the quantity used to represent how similar the two signals are to each other. This quantity of 49 however is meaningless without some other values to compare to. In other words for the speech recognition process we do not wish to know that these two signals return a value of 49 when passed through the dynamic time warping function but rather comparing this value of 49 with other values found with different signals. Furthermore Fig 3.1 has a diagonal but this path does not yield the lowest optimal result for the DTW function. Actually, because of the zero values there are multiple paths that could be taken that would yield a result of zero. Fig 3.4 displays the region where these paths exist.

\[
T_i = [0,0,0,0,1,0] \quad \& \quad R_j = [0,0,1,0,0]
\]

It is obvious that these vectors are nearly identical. The only difference between the two is the index location of the value of one. So reviewing what was shown so far tells us that:

1) If the vectors are nearly identical the optimum path from the start and end point follow a diagonal
2) If the vector lengths to be compared don’t match then it is impossible to to reach the endpoint with only this diagonal path
3) The best path lies near to the diagonal. In other words, all other things being equal, a diagonal move going upwards and to the right should be favoured over simple upward or rightward moves. This constraint can be satisfied if we discount the cost of a diagonal move, i.e. we cost it at, say 50% of the cost of a move upwards or rightwards. [3]
4) The sum of the optimal path values represent how similar the vectors are to each other
5) Time moves forwards. This constraint can be satisfied by allowing just three kinds of move: up, right, or up-and-right. This means that there are only three ways of arriving at a particular cell: from below, from the left, or from below-left. [3]

This leads to the final step in the DTW function process, path finding. The previous paths were easy to visualize since most of the distance calculation revealed zero values within the distance matrix. However, for matrices that contains over 640,000 values and values of zero are rarely seen these paths are not as obvious. By following these ideas a path finding algorithm is formed. Point 5) is interesting because it limits the path to always move towards the end point of both vectors. The result is three possible ways travel from one index as illustrated in Fig 3.5. So the idea is to take the path that provides the minimum result when the two indexes are added whether you are moving from below, the left, or below-left of your current index. So performing this action will lead us one index closer to the endpoint but was the minimum really the optimum path to take? This is not answered until a path is found to the endpoint because performing this movement may lead to paths that contain higher values than other paths that would open if a less efficient index travel was taking previously. So traveling between indexes cannot be eliminated through just taking the minimum. Instead it must find the overall minimum which leads to the Jarvis code function labeled “dtw3.m”.

Within this code three matrices are calculated within the variables “B”, “L”, and “BL” which represent the below path, left path, and below-left path respectively. These matrices all depend on each other for the optimum path finding. Many iterations of the sum from two indexes are performed until each contain a value that hold a minimum distance traveled to that particular node where the final step was from below, left, or below-left. These matrices all call upon values from each other but during its iteration process one matrix cannot call a value for another if it is still not calculated. This leads to a very particular order of operations so no wrong values are retrieved during calculation. For example in Fig 3.5 the “B” value indexed within (2,3) cannot be calculated until the “L” value within index (1,3) is calculated. Further iterations reveals that eventually “B” will depend on “L”, “B”, and “BL” of the lower index. So a systematic order must be present to avoid wrong computation leading to a wrong DTW value.
Now refer back to point 3) of the path finding criterion stating the optimal path lies near the diagonal. Knowing this a typical restriction within path finding can be placed to reduce computation time. By only computing paths within a region of the diagonal the time it takes to compute all possible paths is severely reduced as well as the probability mismatch. This restriction is illustrated in Fig 3.7. Fig. 3.8 represents the restriction used on Jarvis’s DTW function. This region selection relaxes the start point recognition and is simpler to implement. Implementing the regions shown in Fig 3.7 and Fig 3.8 rely on understanding where below, left, and below-left paths are possible which isn’t clear until multiple iterations are complete. The amount of indexes the path can stray from the diagonal is taken as an input in the function “dtw3” as a variable named “w”. This is calculated on a ratio of the lengths of the input to assure that the end point can be reached.

Jarvis constrains each command to begin with the word “Jarvis” which will be the queue for Jarvis to realize that the user meant to deliver a command. The rest of the recording will be followed by two to three words that will progressively narrow down what the user wants which when process correctly will allow controlling an electronic device through voice. The following words after the queue “Jarvis” are from a library of recognized commands. The library’s content is decided on what devices Jarvis should control and some logical operations for that device. An example would be the command “Jarvis, light off” where light decides the device and off is the operation. To maintain an order of simplicity the commands will follow an order of queue, device, and operation. Queue as previously mentioned will be the word “Jarvis”. The device category will be a list of devices Jarvis will control and operation will determine what the device will do. To further simplify the library of words, operation will typically be narrowed to words that can apply for multiple devices such as “on” or “off”. At the same time if a word such as volume is detected it can be assumed that the user did not refer to the device light. By issuing commands in this way a user’s intentions can be better understood. This format leads to a simplistic use of the Hidden Markov Model, a statistical model where the process of selection of states is unknown however the output generated can give information to what the next state will be.

The final step of Jarvis’ speech recognition process is to appropriate the proper templates for each input signal. As mentioned earlier Jarvis relies on user discretion to input a three to four word phrase that formats into queue, device, operation1, and optionally operation2. This is to establish a system of minimizing required comparisons for a match result. This is a very simple version of the Hidden Markov Model where the output of the user’s command follows a format that is recognizable but the process of how the output is chosen isn’t known. The model essentially expects a zero chance of certain words appearing outside of its category. So Jarvis relies on depending the user said a device on the second word but it does not know what the word is or how the user decided on that particular word.

Jarvis uses the dynamic time warping function in addition to relying on user input format to detect a best match for a word. DTW takes a time frequency input of the two compared signals that is centered at the word in time of about a third of the vector length. The reason why the time period is at the center of the word is because this often yielded the best results.
because Jarvis relies on distinction between the words and the chosen vocabulary was most distinct during this time period. It is also passed through a peak finding function defined in Matlab to narrow region of computation. This is due to the unexcited frequencies during the recording will provide an alternative path that could lead to a mismatch as well as additional calculation. The peak finding function returns a vector with less contained data but the data lost is meaningless to the DTW function since many frequency bands remained unused during the user’s command input. The idea is to emphasize distinctions in each word for the DTW to return well separated values. Alternative methods that are proposed is to replace find peaks with and envelope detection to emphasize frequency excitations and narrower time period comparisons for time frequency analysis at the expense of processing time.

<table>
<thead>
<tr>
<th>Device</th>
<th>Match Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light</td>
<td>36.7%</td>
</tr>
<tr>
<td>TV</td>
<td>86.1%</td>
</tr>
<tr>
<td>Radio</td>
<td>74.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation 1</th>
<th>Match Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>On</td>
<td>66.2%</td>
</tr>
<tr>
<td>Off</td>
<td>61.6%</td>
</tr>
<tr>
<td>Channel</td>
<td>82.4%</td>
</tr>
<tr>
<td>Volume</td>
<td>68.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation 2</th>
<th>Match Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>72.2%</td>
</tr>
<tr>
<td>Down</td>
<td>83.2%</td>
</tr>
</tbody>
</table>

Table 2.1: Match Rates for Command Library

The process of voice recognition begins with obtaining the command phrase from the user. After the full string of three or four command words is issued, the waveform generated by the users’ voice is stored in a matrix; this matrix holds the amplitude of each sample of the users’ voice. A sampling window is created in the form of a 1-by-200 matrix. The sampling window slides laterally across the entire waveform, if at any point half of its samples are above a predetermined threshold the moving average filter will signal to Jarvis that it has found the beginning or end of a word.

The moving average filter has eight flags which it can place within a waveform, two for each possible word. As the filter examines the waveform containing the commands, it will place flags at the beginning and end of words. There are two truth bits to determine if the word found is beginning or ending, the start and end bits. If their values are equal the word has begun and if their values differ then the word has ended. A counter is also implemented to track the number of words found, each time an end flag is placed the counter is incremented.

The function which defines the moving average filter returns eight values to the parent function: four separate arrays which define the waveforms of each word found as well as four truth bits. The truth bits hold zero if no word is detected in the corresponding array and a one otherwise. The addition of the truth bits arose as a means to detect the length of the command as well as detect invalid command lengths, commands with less than three words. As of now there is no way to detect if a command exceeds the maximum command length. If more than four words are spoken, the first word is pushed out of the stack and the rest of the words shift down to the next variable. Jarvis will not realize the command is invalid until it checks and sees that the first word is not ‘Jarvis’.

![Fig 3.9: Time Domain Output of Input Waveform (‘Jarvis’ ‘Lights’ ‘On’)](image)

![Fig 3.10: Jarvis Word Separated](image)
The code for both the client and server are included below. [4]
Most of the code was pretty basic. The format was changed
so the message would be in the form of a double instead of a char
which is what it originally was. This code was tested
extensively with various computers and over computers using
different Wifi. One difficulty with the code is that in Matlab
only one function can run at a time. This means to have both
the client and server running at the same time it has to be done
on different instances of Matlab. The devices would call the
function client and input into the function the IP address of
Jarvis and port number for the device. The IP address would
be provided by the user in a text box so that it could change.
Jarvis would call the function server and input the message
from the processor and port number to the device the message
was intended for. The devices then had simple code which
broke down the message into parts to confirm that the first
five bytes represented that device, the next 30 bytes contained
the command for the device, and the final five bytes contained
the action for the device to perform. Other code found in the
devices was for the button controls of the device which
represented if a user directly changed the state of something
on the device by hand. It had to be programmed so that things
only changed if the state of the device was on, there was a
max volume for TV and radio, and that the channels looped
for the TV. These were also tested to determine if any of the
buttons failed to work correctly or if when the message was
received from Jarvis it would fail to perform the action. After
minor errors it worked successfully every time.

A user would have to first calibrate Jarvis’s system to their
voice. The GUI was set up so that two users could have stored
calibrations in the system and indicate to the user if that word
had been calibrated. To calibrate a word the user would click
calibrate and be prompted to say that word. To indicate the
word was successful a check mark appeared next to the word.
One issue that was discovered was that during calibration
Matlab tended to skip code if the user calibrated to fast. This
problem was overcome by adding pauses in the code to give
Matlab time to process everything. The GUI’s should also
display to the user what was received or sent and what Jarvis
was doing. To do this both Jarvis and the devices had static
text boxes which displayed to the user the message being sent
or received. Jarvis also had an additional static text box which
displayed prompts to the user and what the Jarvis was doing
during delays.

```matlab
% CLIENT connect to a server and read a message
% Usage - message = client(host, port, number_of_retries)
function message = client(host, port, number_of_retries)

import java.net.Socket
import java.io.*

if (nargin < 3)
    number_of_retries = 20; % set to -1 for infinite
end

retry = 0;
input_socket = [];
message = [];

while true
    retry = retry + 1;
    if ((number_of_retries > 0) &&
        (retry > number_of_retries))
        fprintf(1, 'Too many retries\n');
        break;
    end

    try
        fprintf(1, 'Retry %d connecting to %s:%d\n',...
                retry, host, port);

        % throws if unable to connect
        input_socket = Socket(host, port);
    end

    % get a buffered data input stream from the socket
    input_stream = input_socket.getInputStream;
    d_input_stream = DataInputStream(input_stream);

    fprintf(1, 'Connected to server\n');

    % read data from the socket -
    % wait a short time first
    pause(0.5);
```

```java
import java.net.Socket
import java.io.*

if (nargin < 3)
    number_of_retries = 20; % set to -1 for infinite
end

retry = 0;
input_socket = [];
message = [];

while true
    retry = retry + 1;
    if ((number_of_retries > 0) &&
        (retry > number_of_retries))
        fprintf(1, 'Too many retries\n');
        break;
    end

    try
        fprintf(1, 'Retry %d connecting to %s:%d\n',...
                retry, host, port);

        % throws if unable to connect
        input_socket = Socket(host, port);
    end

    % get a buffered data input stream from the socket
    input_stream = input_socket.getInputStream;
    d_input_stream = DataInputStream(input_stream);

    fprintf(1, 'Connected to server\n');

    % read data from the socket -
    % wait a short time first
    pause(0.5);
```
bytes_available = input_stream.available;
fprintf(1, 'Reading %d bytes\n', bytes_available);

message = zeros(1, bytes_available, 'uint8');
for i = 1:bytes_available
    message(i) = d_input_stream.readByte;
end
message = double(message);

% cleanup
input_socket.close;
break;

catch
    if ~isempty(input_socket)
        input_socket.close;
    end
end

% pause before retrying
pause(1);
end

% SERVER Write a message over the specified port
% Usage - server(message, output_port, number_of_retries)
function server(message, output_port, number_of_retries)

    import java.net.ServerSocket
    import java.io.*

    if (nargin < 3)
        number_of_retries = 20; % set to -1 for infinite
    end
    retry = 0;

    server_socket = [];
    output_socket = [];

    while true
        retry = retry + 1;

        try
            if ((number_of_retries > 0) && (retry > number_of_retries))
                fprintf(1, 'Too many retries\n');
                break;
            end
            fprintf(1, 'Try %d waiting for client to connect to this host on port %d\n', retry, output_port);
            % wait for 1 second for client to connect server socket
            server_socket = ServerSocket(output_port);
            server_socket.setSoTimeout(1000);
            output_socket = server_socket.accept;
            fprintf(1, 'Client connected\n');
            output_stream = output_socket.getOutputStream;
            d_output_stream = DataOutputStream(output_stream);
            % output the data over the DataOutputStream
            fprintf(1, 'Writing %d bytes\n', length(message));
            d_output_stream.writeBytes(char(message));
            d_output_stream.flush;
            % cleanup
            server_socket.close;
            output_socket.close;
            break;
        catch
            if ~isempty(server_socket)
                server_socket.close;
            end
            if ~isempty(output_socket)
                output_socket.close;
            end
            % pause before retrying
            pause(1);
        end
    end
IV. COST & SUSTAINABILITY ANALYSIS

Router = $10 (one-time fee most already own for other purposes)

Wifi = $100 (monthly fee most houses already pay but would need a larger bandwidth allocation if devices are changed to constantly listening to the processor)

Compatible devices (not yet created) = $25-$100 additional cost over regular device (could be done by placing adapters on walls for on/off devices or by getting new specially made devices)

<table>
<thead>
<tr>
<th>Device</th>
<th>Cost per Compatible Device</th>
<th>Cost over First Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wifi (average)</td>
<td></td>
<td>$75</td>
</tr>
<tr>
<td>Cost per Compatible Device</td>
<td></td>
<td>$15</td>
</tr>
<tr>
<td>Router</td>
<td></td>
<td>$10</td>
</tr>
<tr>
<td>Electricity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The cost associated with Wifi is for the average cost a user would need to spend to increase his/her bandwidth to handle a system that is constantly listening for the user to speak. The router is a one-time cost which most homes already have for their current Wifi. The electricity cost is the cost to have the devices and Jarvis constantly listening for the user’s input. This cost would fluctuate based on the price of electricity but it would not use much electricity to begin with. The additional cost of a compatible device would fluctuate based on the type of device being made. The lowest being a simple on/off light which would take only $25 to add a part in for the socket programming to occur. The highest cost would be a TV which would cost about as much extra as a smart TV does. So about $100 additional cost when compared to a regular TV. Jarvis software is estimated at $10.99 per copy however device bundles can range depending on devices included.

V. CONCLUSION

The return time of a voice recognized command varies depending on the computer’s processing time however performance has ranged for a 4 word command of approximately 20 to 45 seconds. The chosen method was meant to be more adaptable to future devices and the commands that would control them. Although the communication portion has allocated plenty of space for future expansion, adding more devices also increases the chance of wrong command reading. I believe Jarvis requires an additional step on deciding which regions of the returned signal yields the best results. An additional approach is to constrain the DTW function and moving averaging filter to very tight and consistent start and end points of the signal and template. Additionally, Jarvis should take each sample of a user’s word input for reference and build on top of it. This way the template can match the user’s variance on saying that particular word. Overall the voice processor when not challenged can make good estimates on the command processed but varies with the command issued.

The problem associated with this part of the project was to have the processor, Jarvis, communicate the message from the user to the device specified by the user. This problem was solved using socket programming which would piggy back the system most houses already have, Wifi. Since the processor was programmed using Matlab, the socket programming was as well, although in the future this does not need to be performed using Matlab. The client and server were written as functions, code provided above, which the devices and processor would call when ready to transmit or receive data. The user would provide the IP address of Jarvis to the devices. The port numbers of each device is hard programmed into the code of Jarvis and the device. The GUI’s, which represent Jarvis and the devices, were relatively easy to program. The devices could be controlled with buttons or by Jarvis, after prompted with a button. Jarvis was made so that multiple users could save calibrations to the system. It also had an activate button which would ask the user for the message and would automatically send the message after processing it. The devices and Jarvis would output the message received or sent in a static text box for the user to see. Jarvis also prompted the user using a static text box to display what Jarvis was doing or asking the user to do.

REFERENCES


