To appear in "The Phonetician" 111 (2016).

Using web audio to deliver interactive speech tools in the browser

Mark Huckvale

Speech, Hearing and Phonetic Sciences

University College London

m.huckvale@ucl.ac.uk

Correspondence: Chandler House, 2 Wakefield Street, London WC1N 1PF, U.K.

Abstract

In 2014 the number of web pages delivered to tablets and smartphones overtook the number delivered to laptop and desktop computers, with a majority of users saying they prefer these new portable platforms over conventional computers for many tasks. This shift in device use provides both opportunities and challenges for providers of speech analysis tools, phonetic demonstrations and language teaching aids. It is an opportunity because web standards mean we can make our applications available to a wide audience through a single consistent programming architecture rather than writing for one particular computing platform. It is a challenge because tablets and smartphones are less powerful, require different programming skills and have different limitations in terms of user interface.

In this article I will show how interactive applications in Phonetics and Speech Science can be written to run in web browsers on any computing platform. These are native web applications, written in HTML, CSS and JavaScript that can capture, replay, display, process, and analyze audio using the Web Audio API without needing any plug-ins. I will describe - and give the URLs of - some demonstration applications. I will discuss some future opportunities in the area of collaborative research and some remaining challenges that arise from incompatibilities across browsers. My audience is teachers and students with intermediate web programming skills wanting to build custom speech displays, perform custom speech analysis or run speech audio experiments over the web

Keywords

Speech audio, speech analysis, internet, web, programming

1. Introduction

There have been many changes in the field of computing since I started writing speech analysis software in the 1980s. The first Speech Filing System (SFS) tools (Huckvale et al, 1987) were written for the Unix operating system running on engineering workstations only available in scientific laboratories. But as personal computing grew, I developed and ported them to mainstream computing platforms: first to MS-DOS and then to Windows 3, 95, NT, XP, Vista, 7, 8 and now 10. By targeting one platform, my goal was to make the tools available to the largest number of people for the lowest cost in support. Other authors of speech tools have targeted Windows, Mac OS, Linux or the Java VM, but all have primarily addressed users of desktop and laptop computers which were the descendants of those engineering workstations.

Recently however, the landscape of personal computing has changed radically. In 2014, it is said, more web pages were delivered to tablets and smartphones than were delivered to laptop and desktop computers. When asked, users say they prefer these new portable devices over conventional computing devices for a number of activities, including accessing the web[1], managing communications and consuming entertainment media. That preference is probably to do with portability, permanent network connectivity, and significantly better ease-of-use compared to laptops and desktops. In this landscape, our speech analysis tools look out of place, not only in terms of their restriction to particular desktop computing platforms, but because of their old-fashioned user interface and their need for installation and configuration.

There are gains to be had if we were able to make our tools compatible with modern tablets and smartphones by converting them to web applications. Our tools would become more widely available to a broader range of users; distribution would be simplified with our applications sitting on web pages and no longer needing installation, and by exploiting web standards we would be programming for a single environment compatible with all computing platforms.

There are challenges too, of course. Our tools will need a user interface that doesn't require a mouse or keyboard which may involve re-thinking how they are operated – but the result may be tools which are more intuitive and easier to use by non-technical people. The available computational power and storage in tablets is less than in desktops (although improving every year) – but this can be addressed through the use of cloud computing, which also allows for more collaborative work. The personalization of tools with scripts might be more difficult for users – but we have the opportunity for an open plug-in architecture for analysis algorithms too.

In this paper I look towards one approach to putting our speech analysis tools into the hands of modern users of tablets and smartphones: that of exploiting the industry standard programming development environment for audio processing available within web browsers. Web browser applications are different to smartphone and tablet "apps" in that typically they do not need installation or special privileges to operate and they can be delivered in the same way as ordinary web pages. Web applications are good for the novice developer in that the only tools needed to write them are a text editor and a browser. Also because all the program sources are available by default this environment is more open to the sharing of code and algorithms. My goal is to provide practical information on how to build speech audio applications for the teacher or student wanting to build custom speech displays, perform custom speech analysis or run speech audio experiments over the web. My audience is intermediate level developers who have already come to terms with basic elements of web programming.

2. The web software development environment

The browser application environment has special characteristics which provide a number of challenges for software development. The first is the separation between client and server: the client being the browser application running on the user's computer, while the server being the remote system that delivers web services, see Figure 1. Applications can be programmed

[1] http://www.statista.com/statistics/326100/most-important-device-for-connecting-to-the-internet-uk/

to run solely on the client, solely on the server or on a mixture of the two. Typically security constraints limit what services an application can call on either server or client. Notably, the application has very limited access to data stored on the client or to the local hardware. This is to prevent remote applications taking control of the client's computer, such as recording audio or accessing personal information without permission. Additionally the communication between client and server can be unreliable – particularly in mobile networks – so applications need to be robust to slow network transfer speeds. In practice, this means that communications between client and server must be performed in the background, with applications still functional while data is being transferred, and they have to be written with this asynchrony in mind.

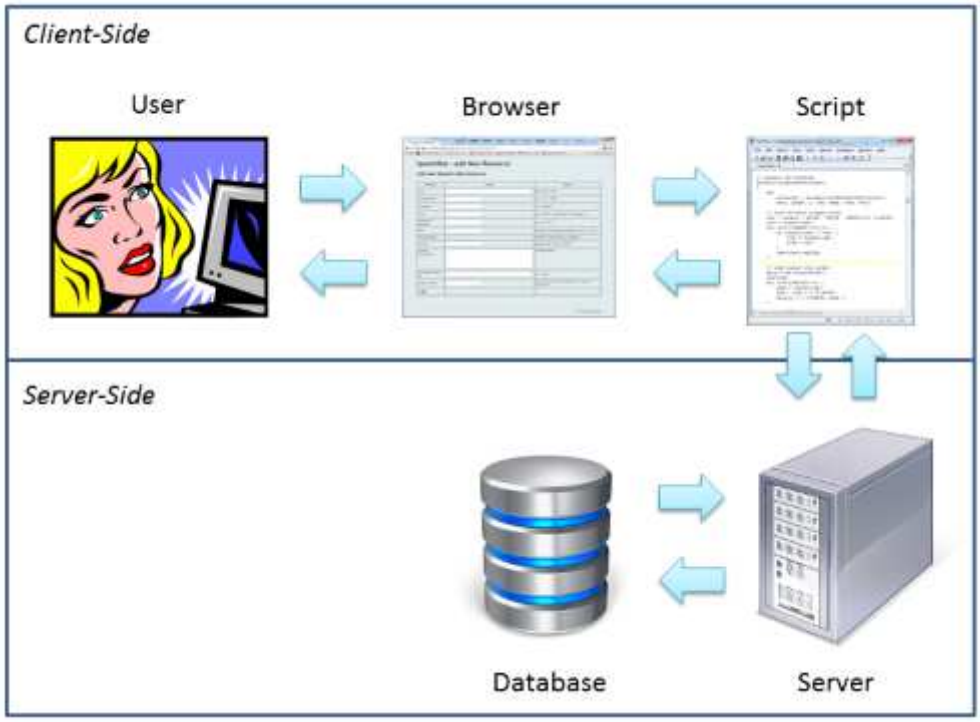


Figure 1. Anatomy of a web application

On the client side, the dominant programming framework involves HTML5, CSS and JavaScript. HTML5 is the mature content mark-up language for web pages, which gives structure to the information displayed in the browser. CSS is the styling language which controls the layout and typography for that information as well as controlling other graphical elements aspects of the page. JavaScript is a programming language which is able to manipulate elements of the web page, as well as performing general purpose programming tasks on the client, communicate with the server, and facilitate access to many other services provided by the browser (such as audio). The combination of HTML, CSS & JavaScript is also becoming the framework of choice for the development of smartphone and tablet "apps", so knowledge of these is now even more important for the modern programmer.

On the server side, scripts may be written in a wide variety of languages, including C++, Python, Perl, and PHP as well as JavaScript. Typically server-side scripts are used to mediate access to databases – providing permanent data storage for transient client-side applications. In contrast to client-side scripts which are distributed in source form, server side scripts are

not generally available to users, and this difference can be used to enforce security and ownership of intellectual property.

In the following sections I will focus on the novel aspects of writing web applications that manipulate audio using the web audio API (application programming interface) [2]. In section 3 I give a complete simple demonstration of a web audio application, while in section 4 I introduce some more advanced capabilities.

3. Web audio demonstration

In this section I give the source listing of a complete web audio application. This application loads an audio file from the client's computer, displays the signal as a waveform and allows the user to replay the audio. It exploits the Flotr graphing library which is described in section 5. Figure 2 shows the application running.

```
<html>
<head>
<meta charset="utf-8">
<title>WebAudio Demonstration</title>

<!-- flotr graphics library from http://www.humblesoftware.com/flotr2/ -->
<script type="text/javascript" src="flotr2.min.js"></script>

<script>

// audio context
var  context=null;

// storage for signal
var signal=[];

// create audio context
function createContext()
{
     if (context==null) {
          // create the audio context
          try {
               context = new window.AudioContext();
          }
          catch(e) {
               alert('Web Audio API is not supported in this browser.');
          }
     }
}

// display the waveform
function displayAudio()
{
     // get target container on page
     var container = document.getElementById("waveform");

     // squeeze signal into 2000 points for efficient plotting
     var  factor=Math.ceil(signal.length/2000);

     // load signal into graph
     var data = [];
     for (var i=0;i<signal.length;i+=factor) {
          var min=signal[i];
          var max=signal[i];
          for (var j=1;j<factor;j++) {
               if (signal[i+j] < min) min=signal[i+j];
               if (signal[i+j] > max) max=signal[i+j];
```

[2] https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html

```
            }
            data.push([ i/context.sampleRate, min ]);
            data.push([ i/context.sampleRate, max ]);
      }

      // Draw Graph using Flotr library
      graph = Flotr.draw(container, [ data ], {
            title : "Waveform",
            shadowSize : 0,
            xaxis : {
                  title : "Time (s)"
            },
            yaxis : {
                  title : "Amplitude",
                  titleAngle : 90
            },
            HtmlText : false
      } );

}

// load a file from client
function loadAudio()
{
      // get the filename
      var file = document.getElementById('filechoice').files[0];
      var filename = file.name;

      createContext();

      // set up a file reader
      var reader = new FileReader();

      reader.onload = function(e) {
            var filedata = e.target.result;
            context.decodeAudioData(
                  filedata,
                  function onSuccess(buffer) {
                        // OK, take a copy of the samples
                        signal = new Array(buffer.length);
                        var srcbuf = buffer.getChannelData(0);
                        for (i=0;i<buffer.length;i++) signal[i] = srcbuf[i];
                        // display waveform
                        displayAudio();
                  },
                  function onFailure() {
                        // load did not succeed
                        alert("decodeAudioData failed on "+filename);
                  }
            );
      };

      reader.readAsArrayBuffer(file);
}

// play some audio
function playAudio()
{
      createContext();

      // create audio buffer source node
      sendsrc = context.createBufferSource();
      sendbuf = context.createBuffer(1,signal.length,context.sampleRate);

      // copy in the signal
      senddat = sendbuf.getChannelData(0);
      for (i=0;i<signal.length;i++) senddat[i] = signal[i];
```

```
      // kick off replay
      sendsrc.buffer = sendbuf;
      sendsrc.loop = false;
      sendsrc.connect(context.destination);
      sendsrc.start(context.currentTime);
}

</script>
</head>
<body>

<h1>WebAudio Demonstration</h1>

<div style="height:1cm;width:100%;background-color:lightgray;display:flex;
align-items:center;justify-content:center;margin-bottom:5mm;">

<input type="file" id="filechoice">
<button onclick="loadAudio()">Load Audio</button>
<button onclick="playAudio()">Play Audio</button>
</div>

<div style="height:10cm;width:100%;" id="waveform">
</div>

</body>
</html>
```

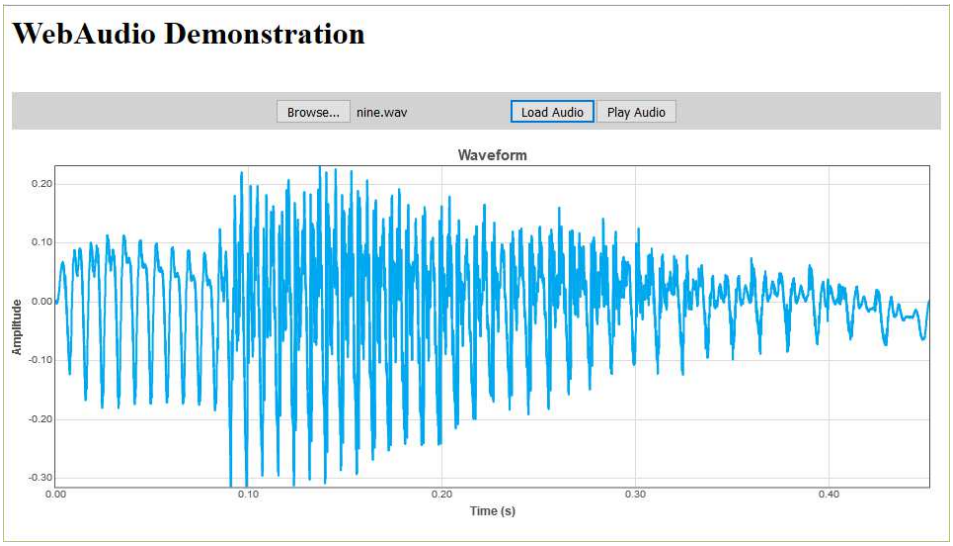


Figure 2. Web audio demonstration

Here is a brief commentary on some of the important elements of the demonstration.

At the heart of the audio functionality in modern web browsers is the AudioContext object. To access any of the web audio methods, it is necessary to first create an audio context object using the window.AudioContext method, as this code demonstrates:

```
try {
      context = new window.AudioContext();
      alert("context.sampleRate="+context.sampleRate);
}
catch(e) {
      alert('Web Audio API is not supported in this browser.');
}
```

The sampleRate property of the AudioContext object gives the sampling rate for all audio operations in the browser. This is typically 44100 or 48000 samples per second. This cannot be changed, and the script must be written to work with the supplied rate.

To load an audio file from the client, a file input element needs to be placed on the web page for the user to select a particular file. For security reasons, scripts are not able to load files by pathname from the client machine. The file input HTML might look like this:

```
<input type="file" id="filechoice">
```

We can get access to the chosen file though the input element's files property.

To read the client file into the application, we can use a FileReader object in conjunction with the AudioContext decodeAudioData method. Reading and decoding takes place in the background, and success and failure are indicated by which of two callback functions are executed, as this code demonstrates:

```
// load a file from client
function openaudio()
{
     var file = document.getElementById('filechoice').files[0];
     var filename = file.name;

     var reader = new FileReader();

     reader.onload = function(e) {
          var filedata = e.target.result;
          context.decodeAudioData(
               filedata,
               function onSuccess(buffer) {
                    signal = new Array(buffer.length);
                    var srcbuf = buffer.getChannelData(0);
                    for (i=0;i<buffer.length;i++) signal[i] = srcbuf[i];
               },
               function onFailure() {
                    trace("decodeAudioData failed on "+filename);
               }
          );
     };

     reader.readAsArrayBuffer(file);
}
```

To play an audio signal, we create a processing chain using AudioContext methods then run the chain through once. The createBufferSource method creates an element in the chain where we can inject audio samples. We create a buffer to hold our signal and pass it to the BufferSource. We then connect the BufferSource object to the output channel (context.destination), and kick off replay with its start() method.

```
// play some audio
var  sendsrc;
function playaudio(sig)
{
     var nsamp = sig.length;

     // create audio buffer source node
     sendsrc = context.createBufferSource();
     sendbuf = context.createBuffer(1,nsamp,context.sampleRate);

     // copy in the signal
     senddat = sendbuf.getChannelData(0);
     for (i=0;i<nsamp;i++) senddat[i] = sig[i];
```

```
     // kick it off
     sendsrc.buffer = sendbuf;
     sendsrc.loop = false;
     sendsrc.connect(context.destination);
     sendsrc.start(context.currentTime);
}
```

To stop the audio playing, it is possible to call the BufferSource stop method:

```
sendsrc.stop()
```

4. Advanced web audio functionality

In this section we highlight additional JavaScript objects and functions available through the web audio API which allow us to load audio from the server, to save audio to the client machine, to record audio and to process audio signals.

To load an audio file from the server, the XMLHttpRequest object can be used to transfer the file to the browser, then the decodeAudioData method of the AudioContext object is used to create an array of sample values. In the code below, note how the loading of the file is conducted in the background and the loadaudio functions returns before the data is actually available.

```
// load an audio file from server
var signal=[];
function loadaudio(aname)
{
     // Note: this loads asynchronously
     var request = new XMLHttpRequest();
     request.open("GET", aname, true);
     request.responseType = "arraybuffer";

     // callback loads signal into global buffer
     request.onload = function() {
          context.decodeAudioData(
               request.response,
               function onSuccess(buffer) {
                    signal = new Array(buffer.length);
                    var srcbuf = buffer.getChannelData(0);
                    for (var i=0;i<buffer.length;i++) signal[i] = srcbuf[i];
               },
               function onFailure() {
                    alert("decodeAudioData failed");
               }
          );
     };

     // get file
     request.send();
}
```

Samples are stored as floats in the range -1.0 to +1.0 and converted to the AudioContext sampling rate. The decodeAudioData method supports a number of audio file formats, including MP3.

To save a signal back to the client machine, we create a WAV file in memory then trigger a download request by faking a click to a hyperlink. We use methods of a DataView object to gain access to a byte buffer and write a 16-bit version of the audio signal to the buffer complete with a WAV file header:

```
// set bytes in a buffer
function writeUTFBytes(view, offset, string)
{
     var lng = string.length;
```

```
     for (var i = 0; i < lng; i++) {
          view.setUint8(offset + i, string.charCodeAt(i));
     }
}

// make a WAV file from signal (16-bit mono)
function makeWAV(signal)
{
     var buffer = new ArrayBuffer(44 + signal.length * 2);
     var view = new DataView(buffer);

     // RIFF chunk descriptor
     writeUTFBytes(view, 0, 'RIFF');
     view.setUint32(4, 44 + signal.length * 2, true);
     writeUTFBytes(view, 8, 'WAVE');
     // FMT sub-chunk
     writeUTFBytes(view, 12, 'fmt ');
     view.setUint32(16, 16, true);
     view.setUint16(20, 1, true);
     view.setUint16(22, 1, true);
     view.setUint32(24, context.sampleRate, true);
     view.setUint32(28, context.sampleRate * 2, true);
     view.setUint16(32, 2, true);
     view.setUint16(34, 16, true);
     // data sub-chunk
     writeUTFBytes(view, 36, 'data');
     view.setUint32(40, signal.length * 2, true);

     // write the PCM samples
     var lng = signal.length;
     var index = 44;
     for (var i = 0; i < lng; i++) {
          view.setInt16(index, signal[i] * 30000, true);
          index += 2;
     }

     // make final binary blob
     var blob = new Blob ( [ view ], { type : 'audio/wav' } );
     return blob;
}

// save file
function saveaudio(sig)
{
     // create a hyperlink and fake a mouse click on it
     var a = document.createElement('a');
     a.href = window.URL.createObjectURL(makeWAV(sig));
     a.download = 'download.wav';
     var event = document.createEvent("MouseEvents");
     event.initMouseEvent(
          "click", true, false, window, 0, 0, 0, 0, 0,
          false, false, false, false, 0, null
     );
     a.dispatchEvent(event);
}
```

To make a recording using the microphone on the client machine, we first make use of the navigator.getUserMedia method to gain access to the microphone, then we use the AudioContext object set up a processing chain from the microphone to a script which siphons off the data passing through it into a global buffer. For security reasons, the getUserMedia function pops up a dialog to the user requesting confirmation that the script may access the microphone.

```
// start audio processing
var micsource=null;
var capturenode=null;
```

```
var recording=0;
function startrecording(stream)
{
     // create the microphone source
     micsource = context.createMediaStreamSource(stream);

     // create a processing node to capture the data
     capturenode = context.createScriptProcessor(8192, 1, 1);
     capturenode.onaudioprocess = function(e) {
          if (recording) {
               // only save data if recording flag is set
               var buf=e.inputBuffer.getChannelData(0);
               for (i=0;i<buf.length;i++) signal.push(buf[i]);
          }
     };

     // connect microphone to processing node and to output.
     micsource.connect(capturenode);
     capturenode.connect(context.destination);
}

// start/pause recording
function recordpause()
{
     // restart acquisition after pause
     if (!recording) {
          signal = new Array();
     }

     // first time only request use of microphone
     if (micsource==null) {
          // accommodate different names in different browsers
          navigator.getMedia = ( navigator.getUserMedia ||
                         navigator.webkitGetUserMedia ||
                         navigator.mozGetUserMedia ||
                         navigator.msGetUserMedia);
          navigator.getMedia(
               {audio:true},
               startrecording,
               function() { alert('getUserMedia() failed'); }
          );
     }

     // start/pause function
     recording = 1 - recording;
}
```

The first time the recordpause function is called, the recorded signal buffer is reset and the microphone is acquired. The second time, the recording is paused. In this code the recording is never actually stopped, merely halted from adding to the captured signal. This means that recording may be restarted without re-acquiring the microphone which would have caused another screen confirmation.

To demonstrate how some signal processing might be applied to a signal, we implement below a non-recursive low-pass filter at 1000Hz using the window method, then apply it to the audio signal through convolution:

```
// sinc function sinc(x) = sin(x) / x
function sinc(x)
{
     return Math.abs(x)<1.0E-10 ? 1 : Math.sin(x)/x;
}

// build non-recursive low-pass filter.
function nrlowpass(freq,ncoeff)
```

```
{
     // create symmetric buffer
     var nhalf=Math.floor(ncoeff/2);
     var filt=new Float32Array(2*nhalf+1);
     // calculate sinc function
     var omega=2*Math.PI*freq;
     for (var i=0;i<=nhalf;i++) {
          filt[nhalf+i]=filt[nhalf-i]=omega*sinc(i*omega)/Math.PI;
     }
     // Hamming window
     for (var i=0;i<=2*nhalf;i++) {
          filt[i] = filt[i] * (0.54-0.46*Math.cos(i*Math.PI/nhalf));
     }
     return filt;
}

// apply a filter to audio
function filteraudio()
{
     var lpfilt=nrlowpass(1000/context.sampleRate,31);
     var fsignal=new Float32Array(signal.length);
     // convolution
     for (var i=0;i<signal.length;i++) {
          var sum=0;
          for (var j=0;j<lpfilt.length;j++) {
               if ((i-j)>=0) sum += signal[i-j]*lpfilt[j];
          }
          fsignal[i]=sum;
     }
     signal=fsignal;
}
```

Other aspects of JavaScript programming are important for building software analysis tools, but are outside the scope of this article. In particular, worker threads are useful mechanisms for performing long calculations in the background without tying up the user interface; and the window.requestAnimationFrame() function is useful in building animations which synchronize to the display refresh rate.

5. Web audio software development

It is not always necessary to program web applications from scratch, since there are an increasing number of freely available libraries of standard functions to reduce development time. Perhaps the most well-known is JQuery[3], but we mention a few libraries directly relevant to speech analysis below.

5.1 Graphing Libraries

Web applications can create graphical elements as well as text. Modern web browsers support both pixel-based and vector-based drawing in 2 and 3 dimensions. For speech signal analysis applications, a common requirement is to produce mathematical graphs and charts, and a library of graph drawing functions provides a simple means for creating graphs without the need to build them from primitives such as lines and dots.

The Flotr2 graph plotting library[4] is a set of JavaScript objects and functions for plotting simple data plots and charts. It is open source and free to use. The Flotr2 library supports all

[3] https://en.wikipedia.org/wiki/JQuery
[4] http://www.humblesoftware.com/flotr2/

major browsers including mobile, and can produce scatter plots, line plots, bar plots and pie charts.

The Highcharts graph plotting library[5] is another pure JavaScript library for plotting graphs. It has more options than Flotr2 and is a little more complex to use. Highcharts is a commercial product, but is free for personal use. The Highcharts library supports all major browsers including mobile, and can produce scatter plots, line plots, bar plots, pie charts, boxplots and many specialised plots.

5.2 Mathematical Libraries

Although JavaScript comes with a standard set of mathematical functions, it is often useful to be able to call on existing libraries of mathematical functions that support signal processing or statistics.

DSP.js is a comprehensive digital signal processing library for JavaScript[6]. It includes many functions for signal analysis and generation, including Oscillators (sine, saw, square, triangle), Window functions (Hann, Hamming, etc), Envelopes (ADSR), IIR Filters (lowpass, highpass, bandpass, notch), FFT and DFT transforms, Delays and Reverb.

SimpleStatistics is a library of basic statistical functions[7] for performing descriptive and inferential statistics, including regression.

6. Examples

The following example web applications were written by the author and chosen to demonstrate the functionality that can be achieved using only HTML, CSS and JavaScript within a web browser.

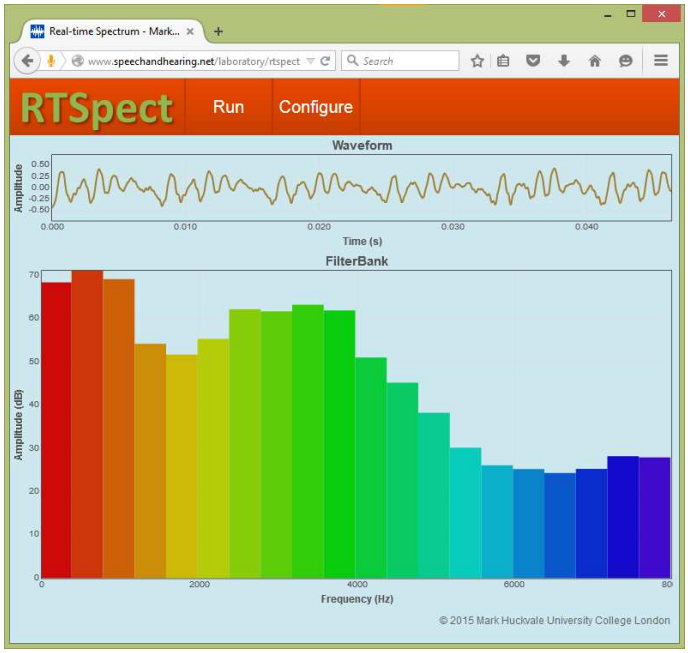
[5] http://www.highcharts.com/
[6] https://github.com/corbanbrook/dsp.js/
[7] http://simplestatistics.org/

RTSPECT

RTSpect provides a real-time spectrum display from the user's microphone with waveform, spectrum and filterbank graphs. The application implements a real-time discrete fourier transform and performs graphical animation using the Flotr2 library.

www.speechandhearing.net/laboratory/rtspect

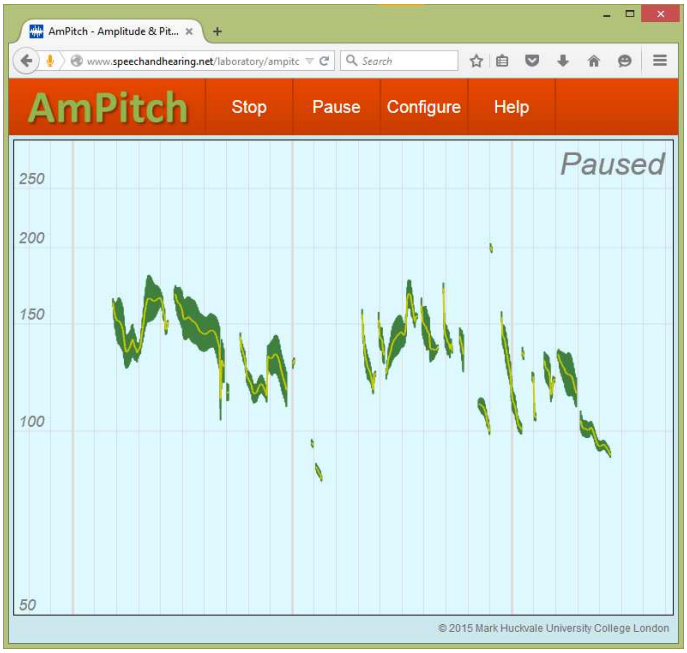
.



AMPITCH

AmPitch provides a real-time amplitude and pitch track display from the user's microphone. The application implements an autocorrelation based fundamental frequency estimation algorithm and scrolling animation using the JavaScript animation methods.

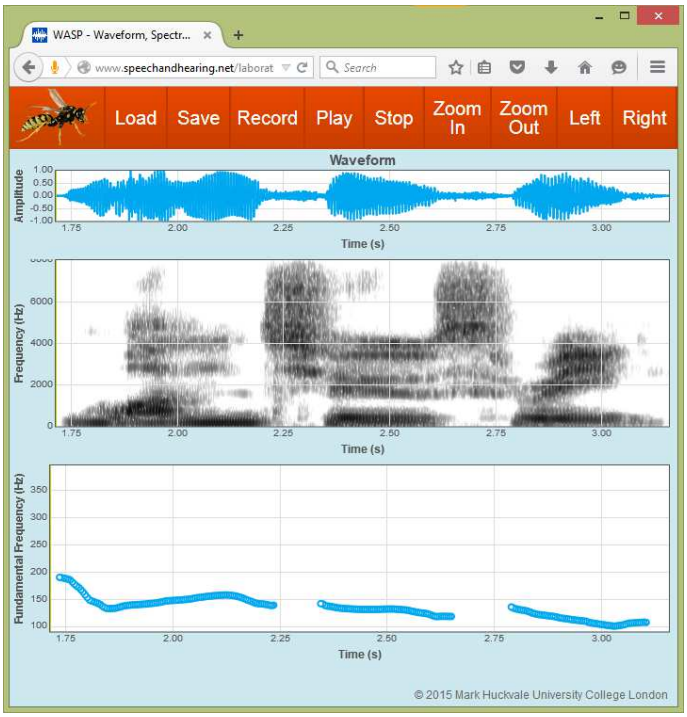
www.speechandhearing.net/laboratory/ampitch



WASP

WASP allows the user to record speech from the microphone and to display its waveform, spectrogram and pitch track. The application implements the SWIPE pitch estimator (Camacho & Harris, 2008) and spectrogram calculation. These run in worker threads since neither work in real time on most devices.

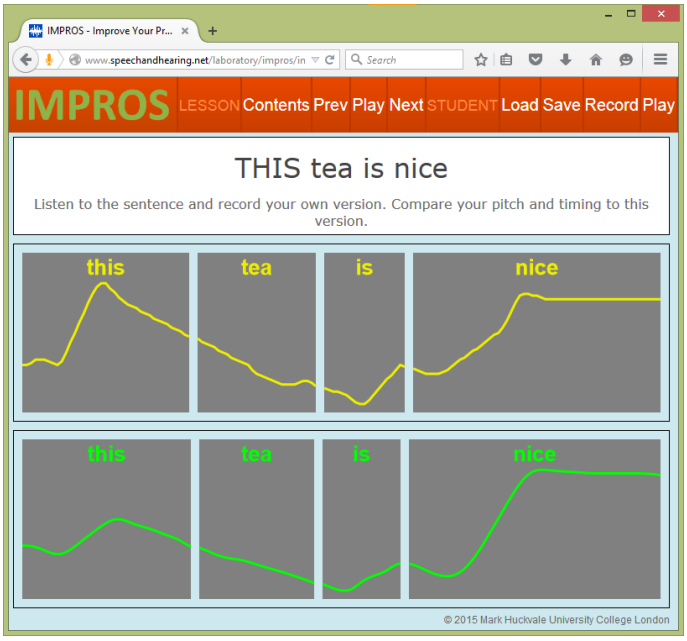
www.speechandhearing.net/laboratory/wasp



IMPROS

ImPros is designed as a tool to improve the prosody of language learners. The user can record a sentence and compare its prosody with a teacher's version. The application implements mel-frequency cepstral coefficient (MFCC) calculation together with the SWIPE pitch estimation algorithm and dynamic programming time alignment.

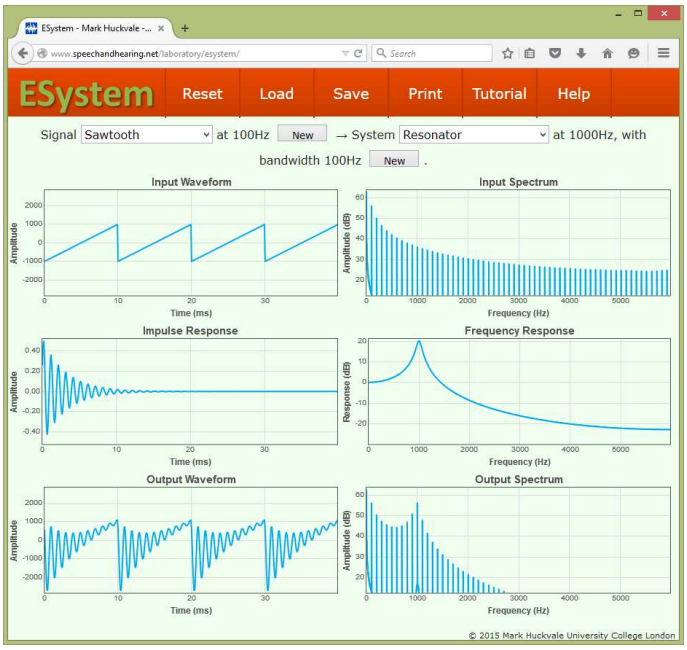
www.speechandhearing.net/laboratory/impros



ESYSTEM

ESystem is a tool for teaching and learning signal and systems theory. The application implements a general-purpose filtering library and fourier analysis. It uses the Flotr2 graph library.

www.speechandhearing.net/laboratory/esystem



7. Discussion and the future

Tablet computers may never fully replace conventional laptop and desktop computers for some applications. But their increasing number, power and ubiquity mean that software developers cannot shy away from making their tools and applications available on these platforms. This article has shown that at least some speech analysis tools originally developed for the Windows platform can be made to run fairly well as web applications within the browser on tablets thanks to the web audio API.

Some incompatibilities between computing platforms remain, particularly in the area of the web audio API which is still quite new. Apple iOS seems to put more constraints on how the AudioContext object is used compared to Android. These problems will be overcome in time, and the future will surely see more web audio applications like the ones described in this article.

In the future there is scope for more sophisticated use of the web application environment, particularly through the exploitation of cloud computing and social networking. The interconnectedness of tablet computing allows for new kinds of collaborative work in which data may be collected and analyzed, and the results shared. We are beginning to see applications for the collaborative construction and labelling of speech corpora, the exploitation of native language speakers across the globe for phonetic analysis and pronunciation training, or the running of experiments in production and perception with hundreds of subjects on their own phones.

The open nature of web programming could be exploited to help advance the field of speech tools if authors are willing to share implementations of state-of-the-art algorithms within the web application framework. I am hopeful that libraries of speech analysis algorithms will be made available in the same way as the graphics and mathematical libraries mentioned above.

8. References

- M.A.Huckvale, D.M.Brookes, L.T.Dworkin, M.E.Johnson, D.J.Pearce, L.Whitaker, "The SPAR Speech Filing System", European Conference on Speech Technology, Edinburgh, 1987.
- Carmacho, J.G. Harris, "A sawtooth waveform inspired pitch estimator for speech and music", J.Acoust.Soc.Am. 124 (2008) 1638-52.