Sound Sensor TDR

Raraa (Remote Audio Recorder and Analyzer)

Name: Cameron S. Embree
Project: Capstone
Semester: Fall 2014
Date: December 8, 2014

v1.0.0

Table of Contents

1. Problem Definition

1.a Problem Scope

This project solves the problem of recording audio, extracting "features" that characterize that audio, and packaged this data in a way that can be easily parsed.

1.b Technical Review

The field of audio recognition is growing and useful for many things. Higher quality cheaper microphones now record more accurately and more powerful computers allow for analysis of this audio. Applications include speech recognition, noise recognition, noise classification, and more. The iPhone's *Siri* is perhaps the most visible current use of audio (voice) recognition and classification. We are interested in creating a cheap and usable infrastructure for researchers to record large samples of audio remotely.

CSU Channel Islands has a new research base on the Santa Rosa Island. This project is the first version of a sound sensor that can be deployed on the Santa Rosa Island to record and extract features from that audio to be sent to the mainland for research purposes. Quality audio recording is a significant data asset for research because it can run all the time, is not limited by visibility, and is omnidirectional while other data like video are only unidirectional and are limited by visibility. Further, audio data is useful for studies in animal migration, bald eagle reestablishment, etc.

1.c Design Requirements

1. Record Audio on *Raspberry Pi*
2. Extract features from Audio
3. Allow for filtering of audio based on features extracted
4. Generate Meta Data File in JSON for features and Audio
5. Optionally wrap features in JSON metadata
6. Not be dependent on ANY CI Rainbow infrastructure

2. Design Description

2.a Overview

Audio is recorded is done using a *Raspberry Pi*, a USB audio card, and a 3.5mm audio microphone. The open source Linux OS *Raspbian* is run on the *Pi*'s and the command line package *arecord* (native to *Raspbian* OS) is used to record audio.

2.b Detailed Description

To understand the *Raara* infrastructure, we first study the basic steps it follows:

1. Read configuration File
2. Start a recording
3. Apply optional filtering to audio
4. Apply optional feature extraction to audio
5. Generate Meta data file for recording & analyses wrapped results
6. Move meta data and user requested data to the data deployment directory

The following a high level explanation of what happens at each step of the previous:

(1 of 6) Read configuration File
A configuration file handler is used to read the state of the config file once at the start and then use these details for subsequent operations. The default config file is $SOUND_BASE/soundsettings.conf unless $CIRAINBOW/cirainbow.conf exists. The cirainbow.conf file always takes precedence. After reading the configuration file once, every state variable is set for the duration of the run.

Config files must end with the ".conf" extension and be of the form:

```
[NODE_INFO]
...
Optional Node info line by line
...

[SOUNDS]
...
sound config details line by line
...
```

There are a minimum number of [SOUNDS] configuration state variables that need to be set. Possible sound settings include:

```
 recordingduration   - (REQ) Duration of each recoding in seconds

 recordingnumber     - (REQ) Number of recordings to make
```

```
samplerate          - (REQ) Sample rate in Hz of each recording

recordingextention  - (REQ) recording extension.
                      DEFAULT is ".wav"

outputform          - (REQ) Either output is plan Yaafe using "FILES" or
                      output is a JSON file using "WRAPPED".
                      DEFAULT: string "FILES"

saverecording       - (OPTIONAL) Save audio that is recorded
                      DEAFULT: string "on" - recordings are saved

analysis            - (OPTIONAL) Turn audio analysis (audio feature
                      extraction via Yaafe) off.
                      DEAFULT: string "off" - filtering is performed

background          - (OPTIONAL) Turn Raraa into a Daemon with "on"
                      DEAFULT: string "off" - prints to terminal

filter              - (OPTIONAL) Turn filtering on/off
                      DEAFULT: string "on" - filtering is performed

recordingprefix     - (OPTIONAL) Recording filename prefix.
                      DEFAULT: 'rec_' at start of each audio file

featureplanpath     - (OPTIONAL) Path to a feature plan file
                      DEFAULT: local sound directory

featureplanname     - (OPTIONAL) Name of a feature plan
                      DEFAULT: string "featureplan"

filterplanpath      - (OPTIONAL) Path to a filter plan file
                      DEFAULT: local sound directory

filterplanname      - (OPTIONAL) Name of a filter plan file
                      DEFAULT: string "featureplan_filter"

recordinglocation   - (OPTIONAL) Directory path where recordings
                      are saved. Analysis still in 'analysis' dir
                      DEFAULT: local 'analysis' dir

datalocation        - (OPTIONAL) Directory where data finished
                      being recorded and analysis goes.
                      DEFAULT: local 'data' directory

analysislocation    - (OPTIONAL) Work space for audio recording
                      analysis and filtering.
                      DEFAULT: local "analysis" directory

latitude            - (OPTIONAL) Latitude loc, DEFAULT is zero
                      DEAFULT: string "0.0000 W"
```

```
longitude           - (OPTIONAL) Longitude location
                      DEAFULT: string "0.0000 N"

rasberrypiid        - (OPTIONAL) Custom string raspberry pi id
                      DEFAULT: string "-1"

simulate            - (OPTIONAL) flag to turn Raraa into sim mode
                      where it copies simulated output files from
                      'simulateiondirectory' to 'data' directory
                      DEFAULT = string "off"

simulationdirectory - (OPTIONAL) Directory where sim output files
                      are found. Copied to "data" directory if
                      simulateion is "on".
                      DEFAULT: durectory "/test/data_yaafe/"

debug               - (OPTIONAL) Turn on/off debug statements
```

An example of a config file might be:

```
[NODE_INFO]
latitude = 17.2343432
longitude = -119.23423423
raspberrypiid = 14

[SOUNDS]
background = on
analysis = on
filter = off
recordingextention = .wav
recordingduration = 3
recordingnumber = 2
recordingprefix = rec_
samplerate = 44100
datalocation = /home/pi/data/
outputform = YAAFE
```

In the above example, two recordings of duration 3 seconds at a sample rate of 44100Hz will be made. Each recording will be saved as a ".wav" file with the prefix "rec_". Filtering is “off”, analysis is “on” so extracted audio data from *YAAFE*, (directed by the local 'featureplan' file) are saved to $CIRAINBOW/data or $SOUND_BASE/data if the first does not exist. Recordings are also saved to “$CIRAINBOW/data/media" or “$SOUND_BASE/data/media”. All other program output is sent to dev/null because “background” is "on".

Consider another example of a config file:

```
[SOUNDS]
recordingextention = .wav
recordingduration = 10
```

```
recordingnumber = 0
recordingprefix = rec_
samplerate = 44100
```

In the above example, an infinite number (noted as zero recordings) of recordings of duration 10 seconds at a sample rate of 44100Hz will be made. The recording will be saved as a ".wav" file with the prefix "rec_". These recordings are saved to the default “$CIRAINBOW/data/media” directory or “$SOUND_BASE/data/media” if the first does not exist. Audio analysis, sometimes called feature extraction, is NOT performed. Further, filtering is NOT performed as well. All program text output is sent to stdout because “background” is not specified. In short, the above config file just makes recordings and saves them to “$CIRAINBOW/data/media” and generates a meta data file to “$CIRAINBOW/data/”.

(2 of 6) Start a recording

First check the config file’s number of recordings. Make a recording and keep track of the number of recordings that have been made. If the number of recordings is less than the number of recordings we are told to make, then continue. *NOTE: If 0 recording are requested, then that means make an infinite number of recordings.*

Recordings are currently performed using the *arecord* command line tool native to *Raspbian*. These recordings are requested by *system()* commands, which starts a sub bash process that generates the audio. The temporary results of extraction are saved to $SOUND_BASE/analysis by default (Can be overloaded in config file).

(3 of 6) Apply optional filtering to audio

Filtering is available so the user can choose to only “mark” specific audio as important based on interpretation of features extracted from that audio. Consider if we marked EVERY piece of audio recorded as interesting. We might overload our network, run out of memory, or simply record audio that contains nothing worth listening to. For example, if we recorded audio near a stream of water, we could construct a filter that only marks audio as interesting if it is above a minimum threshold of noise generated by the stream. Further, if we were only looking for a bird’s cry, we can characterize our particular bird’s cry through testing and generate a filter that will only “mark” audio that contains that particular cry. *NOTE: The particulars of what features from YAAFE are used and how these characterize a particular animals noises is not discusses here.*

If the config file says to perform filtering, then we do the following:

Get a path to a *YAAFE* formatted featureplan file (See *Ref 1* for format details). By convention, the filter features are in a file titled “featureplan_filter” is in *$SOUND_BASE* directory. The path and file name of a *YAAFE* formatted featureplan can be changed to anything in the configuration file (refer to config settings section previous).

Audio filter feature extractions are requested by *system()* commands, which starts a sub bash process that extracted the features from the audio file generated previously. The temporary results of filter feature extraction are saved to "$SOUND_BASE/analysis".

The $SOUND_BASE/src/filters.h file is where "filtering" is performed. Simply put, a filter is a method that returns a True or False indicating if that filter determined the features extracted from some audio to be "Interesting". What determines if a piece of audio's features are "Interesting" is up to the filter.

WARNING: If filtering is activated and the filter used determines that the data extracted from audio is NOT interesting, then NONE of the following steps are performed. In other words, NO additional features are extracted from the audio and NO meta data file is created. Thus, nothing will be sent to the server because the audio, and features extracted for the filter are "not interesting".

(4 of 6) Apply optional feature extraction to audio

Features are extracted using the *YAAFE* (Yet Another Feature Extraction) library. *YAAFE* was chosen for feature extraction because it contained a robust number of different features, the most comprehensive documentation and setup steps. Extracting features is optional because the user could choose to only record audio, or record audio and features.

If the config file says to perform feature extraction, then we do the following:

Get a path to a *YAAFE* formatted featureplan file (See *Ref 1* for format details). By convention, the features are in a file titled "featureplan" located is in *$SOUND_BASE* directory by default. The path and file name of a *YAAFE* formatted featureplan can be changed to anything by configuration options (refer to config settings section previous).

Audio feature extractions are requested by *system()* commands, which starts a sub bash process that extracted the features from the audio file generated previously. The temporary results of feature extraction are saved to $SOUND_BASE/analysis.

(5 of 6) Generate Meta data file for recording & analyses wrapped results

Meta data about the state of the machine and configuration file are save with paths to the audio file used (if it was requested to be saved) and any feature extractions performed (if they were requested). The meta data file generated is JSON formatted and ends with a ".dat" extension.

The JSON wrapped meta data file follows the format:

{

"date": <INTEGER FOR DATE IN milliseconds>

"type": <INTEGER FOR SOUND ID TYPE (Can be set in config)>

```
        “files”: [ “audio_file_name.wav”, “feature_1.csv”, “feature_2.csv”, … ]
        “data”: {
                // SOUND SENSOR META DATA STATE
        }
}
```

The “files” key is the only optional piece of meta data, as the user can request that no audio or features be saved… if they really wanted to. Or the user could choose to “WRAP” their Json.

The following is an example of a feature extracted “WRAPPED” JSON meta data file:


```
{
  "date": "2014-12-06_18:48:56",
  "type": 5,
  "files": [ "rec_D-6-11-114_T-46-48-18.wav" ],
  "data": {
    "sampleRate": 44100,
    "recordingNumber": 1,
    "recordingDuration": 1,
    "latitude": "17.2343432",
    "longitude": "-119.23423423",
    "rpid": "-1",
    "datetimeOfRecording": "D-6-11-114_T-46-48-18",
    "simulation": "no",
    "filter": "yes",
    "analysis": "yes",
    "dataPath": "/home/pi/sounds/data/",
    "configPath": "/home/pi/sounds/sound_settings.conf",
    "mediaPath": "/home/pi/sounds/data/media/rec_D-6-11-114_T-46-48-18.wav",
    "featureNames": [ "MFCC", "PerceptualSharpness", "SpectralVariation", "Loudness"  ],
    "features": {
      "MFCC": {
        "normalize": "-1",
        "resample": "no",
        "samplerate": "44100",
        "version": "0.64",
        "featurename": "MFCC",
        "blockSize": "512",
        "stepSize": "256",
        "CepsNbCoeffs": "11",
        "yaafedefinition": "MFCC blockSize=512 stepSize=256 CepsNbCoeffs=11",
        "data": [ 2.64863, 0.4194, 0.461774 ]
      },
      "PerceptualSharpness": {
        "normalize": "-1",
        "resample": "no",
        "samplerate": "44100",
        "version": "0.64",
        "featurename": "PerceptualSharpness",
        "yaafedefinition": "PerceptualSharpness",
        "data": [ 1.68952, 1.71186, 1.72784, 1.61998 ]
      },
      "SpectralVariation": {
```

```
          "normalize": "-1",
          "resample": "no",
          "samplerate": "44100",
          "version": "0.64",
          "featurename": "SpectralVariation",
          "yaafedefinition": "SpectralVariation",
          "data": [ 0, 0.0368179, 0.299728, 0.239304 ]
        },
        "Loudness": {
          "normalize": "-1",
          "resample": "no",
          "samplerate": "44100",
          "version": "0.64",
          "featurename": "Loudness",
          "yaafedefinition": "Loudness",
          "data": [ 0.165051, 0.0267617, 0.0243643, 0.0244145, 0.0252698, 0.025465, 0.0287419 ]
        }
    }
  }
}
```


The following is an example of a “FILES” formatted JSON meta data file:


```
{
 "date": "2014-12-06_18:45:39",
 "type": 5,
 "files": [ "rec_D-6-11-114_T-33-45-18.wav", "rec_D-6-11-114_T-33-45-18.wav.sv.csv", "rec_D-6-11-114_T-33-45-18.wav.psh.csv", "rec_D-6-11-114_T-33-45-18.wav.lpc.csv", "rec_D-6-11-114_T-33-45-18.wav.mfcc.csv", "rec_D-6-11-114_T-33-45-18.wav.lx.csv" ],
 "data": {
  "sampleRate": 44100,
  "recordingNumber": 1,
  "recordingDuration": 1,
  "latitude": "17.2343432",
  "longitude": "-119.23423423",
  "rpid": "-1",
  "datetimeOfRecording": "D-6-11-114_T-33-45-18",
  "simulation": "no",
  "filter": "yes",
  "analysis": "yes",
  "dataPath": "/home/pi/sounds/data/",
  "configPath": "/home/pi/sounds/sound_settings.conf",
  "mediaPath": "/home/pi/sounds/data/media/rec_D-6-11-114_T-33-45-18.wav",
  "featureNames": [ "SpectralVariation", "PerceptualSharpness", "LPC", "MFCC", "Loudness", "SpectralSlope" ],
  "featureFileNames": {
   "SpectralVariation": "rec_D-6-11-114_T-33-45-18.wav.sv.csv",
   "PerceptualSharpness": "rec_D-6-11-114_T-33-45-18.wav.psh.csv",
   "LPC": "rec_D-6-11-114_T-33-45-18.wav.lpc.csv",
   "MFCC": "rec_D-6-11-114_T-33-45-18.wav.mfcc.csv",
   "Loudness": "rec_D-6-11-114_T-33-45-18.wav.lx.csv",
   "SpectralSlope": "rec_D-6-11-114_T-33-45-18.wav.ss.csv"
  }
 }
```

}

As can be seen from these examples, meta data files tell us about the state of the config file and the results of ONE recording and analysis run.

(***6 of 6) Move meta data and user requested data to the data deployment directory***
Meta data files must end in the *.dat extetention and are moved to the "$CIRAINBOW/data/" directory and any files saved like audio or YAAFE features extracted are moved to the "$CI_RAINBOW/data/media/" directory. If the env var $CIRAINBOW does not exist, we try "$SOUND_BASE/data/" and $SOUND_BASE/data/media/".

2.c Use
Assuming a $CIRAINBOW/cirainbow.conf file exists that is properly formatted and *YAAFE* exists either locally or in the standard execution path, then to install and run the code is as simple as:

```
./install.sh
./start_sound.sh
```

The install script has various optional arguments. The DEFAULT install script does the following:

1. Update any dependent libraries
2. Check for YAAFE installation
    a. If the required env var's for YAAFE does not exist (See R2), then YAAFE is attempted to be install locally. *NOTE: The CIRAINBOW image should have YAAFE installed already, so this step should be skipped.*
3. Ensure that a sound card exists with an microphone
4. Compile *Raara*
5. Generate start and stop scripts called *start_sound.sh* and *stop_sound.sh*
    a. These are copied to $CIRAINBOW/bin if the env var exists

One can customize the install by the following format:

```
./install.sh [-l|--local]
             [-nu|--noupdate]
             [-nc|--nocompile]
             …
```

The following arguments change the install script behavior as follows:

-l|--local)
Only generate start and stop sound scripts locally. Does not attempt to move to the $HOME/bin/

-u|--update)
    Turns ON checking for system updates and dependent library existence/updates
-nu|--noupdate)
    Turns off DEFAULT installation job of checking for system updates and dependent library existence/updates
-c|--compile)
    Turns off DEFAULT installation job of cleaning and remaking the local *Raara* code.
-nc|--nocompile)
    Turns off DEFAULT installation job of cleaning and remaking the local *Raara* code.

For example, if we performed the following install:

```
./install –nu –l
```

Then the above would NOT UPDATE any of the dependent libraries and NOT copy any of the generated start and stop scripts to "$CIRAINBOW/bin". In short, the above only compiles *Raara* (if changes have been made or if it has never been compiled) then creates start and stop scripts locally.

For another example, if we performed the following install:

```
./install –nu
```

Then the above would NOT UPDATE any of the dependent libraries, compile *Raara* (if changes have been made or if it has never been compiled), create start and stop scripts locally, and copy those scripts to "$CIRAINBOW/bin".

3. Conventions

3.a Motivations

We have attempted to be internally consistent and make everything user friendly, no matter how complicated it gets under the hood. For implementation, everything has been done to get code working before concerning ourselves with small changes to efficiency. For example: *YAAFE* can be called directly using C++ but we currently generate command line string for *system()* calls to the *YAAFE* python code. This is slower than direct *YAAFE* compiled c++ libraries, but this can be improved in the future.

In terms of usability, the goal is to make *Raara* (the sound sensor) work independent of the CIRAINBOW infrastructure such that it can be installed and used similar to any linux software package.

3.b Environment Variables

Internally to *Raara* we assume the existence of two environment variables:

- SOUND_BASE
- HOME.

$HOME should result in the path to the user's home folder. This is standard for linux.

$SOUND_BASE should result in the path to the base project directory for the sound code. If the sound script code was located at "home/pi/scripts/sound/" then SOUND_BASE= home/pi/scripts/sound/. The SOUND_BASE env var is generated by the installation script and saved to the sound start script called "start_sound.sh". So long as *Raraa* is started using the install script "start_sound.sh", then $SOUND_BASE will exist.

As part of the CI RAINBOW infrastructure, the env var CIRAINBOW should exist as well. $CIRAINBOW should result in the path to the root of the project where the bin, data, share, etc. folder exist.

3.c Coding standards

Follow the sub sectioned coding standards that follow.

3.c.i Variables

Variables as arguments to methods and in methods are lower camel case. All internal private class variables are all lower case and word separated by underscores. All internal variables must be private and only mutable by mutator methods.

Every class must have a private string called “n” that is the name of the class followed by two colons (used by convention before every print out to user).

For example: A simple bank account class would have the following variable structure.

```
Class bank_account {

private:
        private string n; // Class name, set during init()
        private double account_balance;
        private double interest_rate;
        bool init(); // Init function ALWAYS called at construction. Sets defaults.

public:
        // Methods here
}
```

3.c.ii Methods

Method names are always lower case with underscores between words and usually spelled out fully. For example: “generate_random_number” instead of “gen_rand_num”. Arguments start and end with spaces within the argument parentheses. Open curly braces for methods start on the line of the method signature. If user output is displayed, always preface each output with the name of the class and name of the method in the form “classname::method name: ” (This helps the developer follow program execution). Each class must name a global variable “n” which is a string representing the class name followed by two colons. Each method that shows output must have a variable “mn” at the start of the method that is a string of the method name followed by one colon. (This is done with “cout<<n<<mn<<” Some user message here”)

Constructor always call the “init()” function to set default state for each class. The “init” function returns a bool indicating that the default state was set without failure.

Mutator methods always start their method name with “set_” followed by the name of the variable they are changing. If the private internal variable was “student_age”

then the mutator method name would be “set_student_age”. Mutators return a Boolean indicating their successful change.

Accessor methods always start their method name with “get_” followed by the name of the variable they are accessing. If the variable was “student_age” then the mutator name would be “get_student_age”. Accessors return the type of the variable they are accessing.

For example: A business method to compute tax would be:

```
double compute_tax( double taxRate, double amount ) {
        string mn = “compute_tax::”;
        double res = 0.0;

        cout<<n<<mn<<“Computing tax … ”;
        //some work is done

        cout<<n<<mn<<The tax rate is: ‘”<<res<<”””<<endl;

        return res;
}
```

For example: A mutator method to set the tax rate would be:

```
bool set_tax_rate( double taxRate ) {
        double res = true; // success or not

        if ( taxRate > 0) {
                tax_rate = taxRate;
        } else {
                res = false;
        }

        return res;
}
```

For example: An accessor method to get the tax rate would be:

```
double get_tax() {
        return tax_rate;
}
```

3.c.iii Classes

Class names are always lower case and word separated by an underscore. Every class has an “init()” function called during the constructor. All private internal class states states are changed by Accessors and Mutators only. Variable are declared before functions in each public and private section.

The $SOUND_BASE/src/utils.h file should be a home for helper functions that are cross class applicable.

3.c.iv Comments

Comment code where it helps the understanding of the reader. It would be nice to have a method summer above each method but at least leave comments describing why we are doing each next step.

3.d Error Handling

All cases where *Raraa* could crash should be covered by an error handler. Errors should never crash the program without (at least) FIRST reporting what they think the error was using one of the many JSON formatted error output message functions found in $SOUND_BASE/srs/utils.h and in the *Raraa* code itself.

3.e Ease of Use

Ensure that any output messages are meaningful and user input relegated to the configuration file only.

Update the CHANGELOG file located at $SOUND_BASE whenever you make changes to remind yourself of previous work.

Update the README file immediately after you make changes.

4. *Raraa* Directory Structure

The basic directory structure for the sound script is as follows:

```
sound_script/
        analysis/

        data/
                media/
        logs/
        src/
        test/
                data_fv/
                data_yaafe/
        README
        CHANGELOG
        featureplan
        featureplan_filter
        sound_settings.conf
        install.sh
        makefile
        raraa.cpp
```

The following is the use of each element of the above structure:

sound_script/
: Directory containing all the code.

analysis/
: Working directory for temporary recording and analysis before they are wrapped up to be sent to the backend.

data/
: Local directory for data to be sent to the backend. Only used if another data directory

data/media/
: Directory for media files (audio, *YAAFE* features extracted from audio, etc)

logs/
: A place for log files.

src/
: The home for all files except raraa.cpp itself.

test/
: Any test code but also the two directories data_fv and data_yaafe which are directories for example outputs for the config file simulation mode.

test/data_fv/
: Example output from *Raraa* with the config file option:

"outputform = WRAPPED".

test/data_yaafe/

Example output from *Raraa* with the config file option:
"outputform = FILES".

README

Readme for how to install and use *Raraa*.

CHANGELOG

Keep track of any changes you make here with your name and date. Append to the top of the document.

featureplan

YAAFE formatted feature plan. This is the default file for features if no other is pointed to in the config file.

featureplan_filter

YAAFE formatted filter feature plan. This is the default file for features if no other is pointed to in the config file.

sound_settings.conf

Default configuration file that is used if $CIRAINBOW/cirainbow.conf is not found.

install.sh

Installation script.

makefile

makefile used by the installation script.

raraa.cpp

The core *Raraa* code.

5. Evaluation

5.a Results

Installation and use of *Raraa* is simple and many output formats are supported. For example, we can change any of the following options to generate unique output:

- SaveAudio = on/off
- Filtering = on/off
- FeatureExtraction = on/off
- OutputForm = FILES/WRAPPED

Simply editing the config file should accommodate any person's needs for output formatted in JSON.

Installing and starting the sound script is as easy as:

```
./install.sh
./start_sound.sh
```

The sound script can be stopped, edited, and started as easily as:

```
./stop_sound.sh
   // edit configuration file now
./start_sound.sh
```

5.b Future Work

1. Change *system()* calls for recording audio to be internal method that do not use system calls
2. Change *system()* calls for *YAAFE's* python library to use *YAAFE's* compiles cpp library. This will improve speed of analysis
3. Generate a few more robust examples of filter use.
    a. This could be a project in and of itself

6. References

Ref 1) *YAAFE* Feature vector file format
http://yaafe.sourceforge.net/manual/quickstart.html#feature-definition-format

Ref 2) *YAAFE* project
http://yaafe.sourceforge.net/

Ref 3) *YAAFE* user manual
http://yaafe.sourceforge.net/manual.html