Problem 1: Automatic Gain Control  
Theory of Operation  
Implementation in Python for Simulation  
Implementation in C  
Part a  
Part b: Static Test  
The AGC data structure being debugged  
Part c: Dynamic Test  
Setup of the Analog Discovery Channel 1 function generator in Modulation mode.  
Setup of the Agilent 33250 to produce the dynamic waveform.  
I will expect a real-time demo of this in the lab. Have the debugger running with the AGC structure in a watch window.  

Problem 2: Two Options for ECE 4655, One Option for ECE 5655  
Option A (for ECE 4655): Signal Mean and Variance using CMSIS-DSP  
Part a  
Part b  
Part c  
Part d  
Part e  
Option B (for ECE 4655): Speech Diversity Combining  
Estimating the SNR  
Signal Combining Algorithms  
Speech Test Vector for Problem 2  
A 1 dB Steps Lookup Table for Use in C Code  
Building and Testing a Python Prototype  
Part a  
Part b  
Part c  
Part d  
Part e

%pylab inline
#%matplotlib qt
import sk_dsp_comm.sigsys as ss
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG

Populating the interactive namespace from numpy and matplotlib
Due No Later Than Friday April 5, 2019

The midterm exam is in the form of a take-home exam, meaning each student is to do his/her own work without consulting others. All problems involve working with the Cypress FM4 and also some use of Python. The due date is at the end of the day Friday April 5, 2019.

I have written the exam in an Jupyter notebook so it is available as a static PDF document and also as a live notebook document.

**What to Turn In:** Write your exam solutions in this Jupyter notebook. You can use markdown cells to document your C source code and you can insert screen shots as png graphics using something like:

```
Image('fname.png',width='90%')
```

or in a markdown cell you can enter an HTML image tag of the form

```
<img src="fname.png" width="90%">
```

I would like a paper copy of the notebook following conversion to LaTeX and then PDF. **Note:** The lab computers can do this conversion or you can do this if you add Pandoc and then MiKTeX to your Python setup. Also Email me a ZIP package containing the Jupyter notebook. A lab demo is required in Problem 1c.

**Note:** An alternative to setting up LaTeX is to just install Pandoc and then export as **HTML or Markdown**. With the HTML export you can print directly from the browser. With the Markdown export you can open the folder of files as edit and convert to PDF using the Editor **Typora**.

**Problem 1: Automatic Gain Control**

In this problem you will implement an automatic gain control (AGC) component [https://en.wikipedia.org/wiki/Automatic_gain_control](https://en.wikipedia.org/wiki/Automatic_gain_control). Ultimately the design will use a data structure similar to PN generator of Assignment 2 and also borrow a few ideas from [FIR_filters.c](FIR_filters.c). The system block diagram is shown below. The idea behind the AGC to take the uncertain amplitude level on the input signal and produce a constant level output signal.
Most AGC designs involve feedback as a means to regulate the output to a prescribed level. In searching the literature there are a few design approaches that use feedforward control as well. Here the focus is on a classical feedback design, but implemented in the discrete-time domain. The regulating aspect of the AGC has an associated time constant or response bandwidth. As a rule the AGC bandwidth is set below the bandwidth of amplitude fluctuations in the signal you want to retain. In a radio receiver application the AGC bandwidth is well below and desired amplitude modulation, but wide enough to track signal fading due radio wave propagation phenomena. In an audio application such as speech in a sportscast, the bandwidth is narrow or the time constant is large enough to ignore or smooth over the natural ups and downs of speech syllables. The bandwidth however has to be wide enough to track out changes in the announcer’s volume, i.e., as the head moves in front of the microphone. You would not for example use AGC on the audio produced by an orchestra, eliminating soft and loud passages destroys the character of music.

Theory of Operation

The envelope detector block, implemented in C using \( \text{fabs()} \), computes the absolute value of the output using the data type \( \text{float32_t} \) (in Python with Numpy we use \( \text{abs()} \)) in order to estimate AGC output signal level as a positive value. For a sinusoidal signal the absolute value produces a full-wave rectified signal. Filtering a full-wave rectified sinusoid gives the average value in continuous-time which is

\[
X_0 = \frac{A}{T_0} \int_0^{T_0} |A \sin(2\pi t/T_0)| \, dt
\]

\[
= \frac{A}{T_0} \int_0^{T_0/2} \sin(2\pi t/T_0) \, dt - \frac{A}{T_0} \int_{T_0/2}^{T_0} \sin(2\pi t/T_0) \, dt
\]

\[
= \frac{2A}{\pi} = A \cdot 0.6366
\]

See the Sympy calculation below, followed by a quick discrete-time simulation for the case \( A = 1 \).

```python
# Symbolic calculation for continuous time:
t, A = sym.symbols('t A')
f0, T0 = sym.symbols('f_0 T_0', positive = True)
signal = A*sym.sin(2*sym.pi*t/T0)
signal
```
It is desireable to have the ref_lvl refer to the input peak level, rather than the average value of the AGC output. The envelope detector block thus scales the output to refer to the peak value of a sinusoidal input. So if you want an output peak amplitude of say 10000, you set ref_lvl = 10000. The settling/step size parameter of the loop, alpha, controls the loop bandwidth. The larger alpha is the faster the response time. Making alpha too large will make it track/regulate amplitude features of the input waveform, and hence remove or distort desirable features. In general make alpha very small to insure the AGC just tracks/regulates the slowly varying waveform characteristics. For ref_lvl amplitudes in the range of a reasonable alpha value is 0.001. For ref_lvl in the range of [2000, 20000] is is useful to include a scaling factor such as ref_lvl = 0.001*1E-3. The two blocks to the left of the alpha scaling implement an accumulator with the output delayed by one sample, i.e., the input/output difference equation and system function are

\[
\begin{align*}
g[n] &= g[n - 1] + \alpha e[n] \\
G(z) &= z^{-1} G(z) + \alpha E(z) \\
z^{-1} \frac{G(z)}{E(z)} &= \frac{\alpha z^{-1}}{1 - z^{-1}}
\end{align*}
\]

One more practical detail is to include a maximum gain that the AGC can rise to. If no signal is present you do not want the gain to up without bound. The final block before the input waveform multiplier serves this purpose. If the feedback loop attempts to increase g[n - 1] above g_{max} the value of g[n - 1] is limited to g_{max}.

The C-implementation of this algorithm is the focus of this problem. Before heading in that direction we consider a simulation example in Python to motivate the C implementation.

---

**Implementation in Python for Simulation**

For testing and evaluation the algorithm is prototyped in Python via the class `AGC` given below:

```python
class AGC:
    
```
As a simple test set up test bed operating at $f_s = 48$ Ksps and input a sinusoidal signal at 1 kHz having input amplitude 5000 and $\text{rev lvl} = 20000$. Note: 5000 is representative of the integer peak values arriving as `int32_t` values for the audio codec. To add interest to this waveform a small amount of amplitude modulation (AM) at 10 Hz is added. Presently the modulation depth is 10%. To explore the impact of $\alpha$ the output for two value are given in the plots that follow.

```python
n = arange(100000)
fs = 48000
fc = 1000
fm = 10
x = 5000*(1+0.1*cos(2*pi*fm/fs*n))*cos(2*pi*n*fc/fs)
#ylim([0,1.5])
alpha = 0.0001
y1 = zeros_like(x)
AGC1 = AGC(20000.0, alpha*1e-4)
for k, xk in enumerate(x):
    y1[k] = AGC1.process(xk)

alpha = 0.001
y2 = zeros_like(x)
```

Implement a discrete-time Automatic gain control system

Mark Wickert March 2017

```python
def __init__(self, ref_lvl, alpha = 0.001, gain_max = 20):
    # Initialize the object used to encapsulate discrete-time AGC functionality.
    self.gain = 1.0  # initial condition
    self.gain_max = gain_max
    self.ref_lvl = ref_lvl
    self.alpha = alpha

def process(self, x_in):
    # Implement one iteration of the AGC algorithm
    # Apply AGC gain
    x_out = self.gain * x_in
    # Form the envelope
    z_env = abs(x_out)*pi/2
    # Form the error
    error = self.ref_lvl - z_env
    # Update the AGC_gain
    self.gain += self.alpha*error
    # Limit gain to gain_max
    if self.gain > self.gain_max:
        self.gain = self.gain_max
    return x_out
```
AGC1 = AGC(20000.0, alpha*1e-4)
for k, xk in enumerate(x):
    y2[k] = AGC1.process(xk)

subplot(211)
# plot(n/fs*1000, x)
plot(n/fs*1000, y1)
# plot(n/fs*1000, (10-(10-6)*exp(-n/fs/tau)))
title(r'In 5000, AGC Out 20000 with 10 Hz Env. Fluctuations')
ylabel(r'Output Amplitude')
xlabel(r'Time (ms)')
grid();
subplot(212)
# plot(n/fs*1000, x)
plot(n/fs*1000, y2)
# plot(n/fs*1000, (10-(10-6)*exp(-n/fs/tau)))
xlabel(r'Time (ms)')
grid();
tight_layout()
The results are reasonable for an input start at 5000 and begin gain leveled (AGC'd) to 20000. The AM modulation at 10 Hz is preserved. The 10 Hz can be thought of as a fading pattern.

**Implementation in C**

The desired approach for implementing the AGC on the FM is to use a data structure and write two functions: (1) initialize the structure and (2) a process function that runs the AGC algorithm over a frame of Nframe samples. Note: For sample-by-sample processing you set Nframe = 1. The frames is for future use in DMA code. The frames capability also follows the design pattern of the ARM CMSIS-DSP library.

The code base should be: defined in AGC.h and implemented in AGC.c. The AGC.h file is included in the midterm ZIP file and listed here:

```c
//AGC.h
// AGC header
 // Mark Wickert March 2017

#define ARM_MATH_CM4
#include <s6e2cc.h>
#include "arm_math.h"

//Used for scaling sinusoidal average value to peak value
```
You are responsible for writing `AGC.c`.

- **Part a**

Write the code module `AGC.c` so that it can be tested in the main module `fm4_loop_intr_AGC_GUI.c` as found in the midterm ZIP package. Snippets of this module are included below:

```c
#define PI_BY_TWO 1.570796

/*
Structure to hold AGC parameters and state information.
*/

struct AGC_struct_float32
{
    float32_t gain;
    float32_t gain_max;
    float32_t alpha;
    float32_t ref_lvl;
};

void AGC_init_float32(struct AGC_struct_float32 *AGC,
                       float32_t gain,
                       float32_t gain_max,
                       float32_t alpha,
                       float32_t ref_lvl);

void AGC_process_float32(struct AGC_struct_float32 *AGC,
                           float32_t *x_in,
                           float32_t *x_out,
                           int16_t Nframe);

// fm4_loop_intr_AGC_GUI.c

#include "fm4_wm8731_init.h"
#include "FM4_slider_interface.h"
#include "AGC.h"
...

// Create (instantiate) AGC data structure
struct AGC_struct_float32 AGC;

void PRGCRC_I2S_IRQHandler(void)
{
    ...
    x = FM4_GUI.P_vals[0] * sample.uint16bit[LEFT];
    xR = (int16_t) (FM4_GUI.P_vals[1] * sample.uint16bit[RIGHT]);
    // Do more processing on xL and xR
```
Note the GUI slider is configured to allow tuning of ref_lvl, alpha, and gain_max. Assume that the displayed value of alpha is further scaled by $10^{-3}$.

AGC_process_float32(&AGC, &x, &y, 1);

int main(void)
{
    ...
    init_slider_interface(&FM4_GUI, 460800,
                          1.0, 1.0, 10000, 0.001, 20.0, 0.0);

    // Init AGC structure
    AGC_init_float32(&AGC, 1.0, 20, 1e-6, 10000.0);

    ...

    while(1){
        // Update slider parameters
        update_slider_parameters(&FM4_GUI);
        if (FM4_GUI.P_idx == 2) {
            AGC.ref_lvl = FM4_GUI.P_vals[2];
        }
        else if (FM4_GUI.P_idx == 3) {
            AGC.alpha = FM4_GUI.P_vals[3]*1e-3f;
        }
        else if (FM4_GUI.P_idx == 4) {
            AGC.gain_max = FM4_GUI.P_vals[4];
        }
    }
}

Paste your final AGC code into the space below for report documentation of the working module:
Part b: Static Test

In this part you test the static gain leveling capabilities of the AGC function using a 1 kHz sinusoid. You vary the amplitude of the left channel input from 10 mv to 1 v peak. Set `ref_lvl = 10000`. In particular take measurements with an input level of 10, 100, 500, and 1000 mv peak. Record the value of `gain` required to level the output to 10000 in terms of `int16_t`, and the corresponding peak voltage from the CODEC left channel DAC. To observe `gain` configure a watch window in the debugger for the data structure `AGC`. See the figure below for details. If you need to increase `gain_max` above the initial set value of 20, make note of the new value. While taking these measurements keep the GUI gain slider `P_vals[0] = 1.0` at all times.

The AGC data structure being debugged

Values will change as the slider change and the waveform changes.

Part c: Dynamic Test

In this part a dynamic test of the AGC is performed using a waveform that is composed of a sinusoid at 10 kHz amplitude modulated by a 10 Hz squarewave. The test waveform can be generated easily using the Analog Discovery and the Agilent 33250A function at the lab bench. The next two figures show the configurations respectively.

Setup of the Analog Discovery Channel 1 function generator in Modulation mode.
Setup of the Agilent 33250 to produce the dynamic waveform.

Note to get 100 mV peak increase the amplitude to 200 mV p-p.

The waveform captured using the Analog Discovery is shown below:

```python
# Remove waveform DC bias and change the units.
plot(t_AD*1000,(Chan2_AD-mean(Chan2_AD))*1000)
xlim([-100,100])
title(r'10 kHz Sinusoid with 10 Hz Square-wave Modulation at 50% Depth')
ylabel(r'Amplitude (mv)')
xlabel(r'Time (ms)')
grid();
```
Your output waveform should look similar to the capture obtained using the Analog Discovery shown below:

```python
# Remove waveform DC bias and change the units.
plot(t_AD*1000, (Chan1_AD-mean(Chan1_AD))*1000)
plot(t_AD*1000, (Chan2_AD-mean(Chan2_AD))*1000)
xlim([-100,100])
title(r'Output Envelope Transients with 50% Modulation')
ylabel(r'Amplitude (mv)')
xlabel(r'Time (ms)')
legend((r'Output',r'Input'),loc='best')
grid();
```
Explain the detailed shape of the AGC output waveform as the input switches from the low to high amplitude. To aid your understanding and explanation, primarily vary `alpha`, but make changes to `agc_lvl` and `gain_max` to fully understand the interaction.

I will expect a real-time demo of this in the lab. Have the debugger running with the AGC structure in a watch window.
Problem 2: Two Options for ECE 4655, One Option for ECE 5655

If you are taking the course as ECE 4655 you have the choice between Option A Exploring the ARM CMSIS-DSP Functions arm_mean_f32 and arm_var_f32 or Option B Speech Diversity Combining. I have recently spent quite a few hours trying to get reasonable results for the speech diversity combiner using simple algorithms. I want the graduate students to have this added challenge and hence require that they choose Option B. For the undergraduate students, ECE 4655, I recommend Option A, but if you go for Option B I will consider up to 10% bonus points out of the 100% allocated for the midterm grade.

• Option A (for ECE 4655): Signal Mean and Variance using CMSIS-DSP

In this problem you will re-purpose the function `FIR_header()` found in `sk_dsp_comm.coeff2header` for creating a known test signal that can be imported into a FM4 project. In this problem again start with a copy of `fm4_loop_intr_GUI.c` as found in the ZIP package. This is not going to be a real-time project, so you need to comment out the line

```python
# You may want to change the variable h_FIR to x_sig or similar
# Similarly change #define M_FIR to #define N_xvec or similar

def FIR_header(fname_out, h):
    ""
Write FIR Filter Header Files
    ""

    M = len(h)
    N = 3 # Coefficients per line
    f = open(fname_out,'wt')
    f.write('//define a FIR coefficient Array\n\n')
    f.write('#include <stdint.h>\n\n')
    f.write('#ifndef M_FIR\n')
    f.write('#define M_FIR %d
' % M)
    f.write('#endif
')
    f.write('/************************************************************************/
    /*                         FIR Filter Coefficients
*/
')
    f.write('float32_t h_FIR[M_FIR] = {'
    kk = 0;
```
Part a

Use a header file to store a custom test signal generated in Python to be stored in the header file as float32_t values. The function that does this is listed above, `FIR_header(fname_out,h)`.

Create in Python the test signal
\[
x[n] = 3\cos\left(2\pi \cdot \frac{f_1}{f_s} \cdot n\right) + 2\sin\left(2\pi \cdot \frac{f_2}{f_s} \cdot n\right)
\]
where $f_s = 48000$ Hz, $f_1 = 1000$ Hz, and $f_2 = 400$ Hz. Create 1000 values, that is let $n \in [0,1000]$.

```python
# To get started create x[n]
f1 = 1000
f2 = 400
fs = 48000
n = arange(0,1000)
x = 3*cos(2*pi*f1/fs*n) + 2*sin(2*pi*f2/fs*n)
```

# Add Python code to create the header and then move on to part b

Part b

Using the CMSIS-DSP functions

```c
void arm_mean_f32 (float32_t *pSrc, uint32_t blockSize,
                   float32_t *pResult)
    /*Mean value of a floating-point vector. */

void arm_var_f32 (float32_t *pSrc, uint32_t blockSize,
                   float32_t *pResult)
    /*Variance of the elements of a floating-point vector. */
```
develop code for the Cypress FM4 to calculate the mean and variance of the test vector values read in from the header file you created in part (a). You may have to do some variable renaming in the header itself to make more sense in your code, that is change \texttt{h\_FIR[M\_FIR]} to \texttt{x[M\_FIR]} and maybe change the \texttt{#define M\_FIR} to \texttt{#define N\_xvec}. You can actually do the rewrite of the header writing function given above to be a test vector generation header file writer.

- **Part c**
  Compile and run the function and provide screen shots of the computed values.

- **Part d**
  Profile the calculation of the mean and the variance calculation using the Keil MDK profile clock. Optionally place GPIO \texttt{High/Low} calls around your code and capture the rising edge/falling edge timing event with the logic analyzer.

```c
gpio_set(TEST_PIN, HIGH);
/* your code here */
gpio_set(TEST_PIN, LOW);
```

- **Part e**
  Compare the values calculated via CMSIS-DSP with Python calculations obtained using:

```python
mx = mean(x)
varx = var(x)
print('Mean of x = %6.4f and var of x = %6.4f' % (mx, varx))
```

Also comment on what you expect theoretically from signals and systems theory, if the record of samples is infinite in duration. Note the \texttt{mean()} operator is like finding the average value and \texttt{var()} is like finding the average power.
Option B (for ECE 4655): Speech Diversity Combining

In this problem you explore how a two microphone audio system, such as found on a cell phone, can perform diversity combining to choose the best quality audio to pass on to the user. The system block diagram is shown below:

The idea of diversity combining is used with radio signal, think two antennas on your wireless router or two antennas per sector in a cell tower. With speech a similar need exists. When one or the other signal may be weaker and contain significant noise, the hope is the signal on the other antenna/microphone may have a higher signal-to-noise ratio (SNR). The two diversity combining schemes considered in this problem are: (1) threshold combining, where the signal with the highest SNR is used for the combiner output; (2) maximal ratio combining where a weighted sum of the signals is formed, with the combining gains related to the SNRs of the signals.

The SNR estimator is somewhat involved, as speech is bursty in nature. References used for this problem are:

1. Methods for Speech SNR estimation: Evaluation Tool and Analysis of VAD Dependency
2. Long-Term SNR Estimation of Speech Signals in Known and Unknown Channel Conditions

Formally, we estimate SNR in dB as

\[
\hat{\text{SNR}}_{\text{dB}} = 10 \log_{10} \left( \frac{\hat{\sigma}_r^2}{\hat{\sigma}_n^2} \right) = 10 \log_{10} \left( \frac{\hat{\sigma}_r^2}{\hat{\sigma}_n^2} - \frac{\sigma_n^2}{\hat{\sigma}_n^2} \right)
\]

where the second line shows that SNR can be estimated using the total received power in signal \( r_i[n] \), \( i = 1, 2 \) and an estimate of just the noise power alone, \( \sigma_n^2 \). To get access to just the noise component alone we need to further estimate the voice activity factor (VAD), which measures when speech is present.
With the VAD signal it is now possible to process just the noise segments of the noise speech waveform and estimate \( \hat{\sigma}_n^2 \). The entire noisy speech samples can be used to estimate \( \hat{\sigma}_s^2 \). We can again use exponential averaging to estimate the variances:

\[
\begin{align*}
\text{Always} & \quad \hat{\sigma}_x^2[n] = \rho_{\text{var}} \hat{\sigma}_x^2[n-1] + (1 - \rho_{\text{var}}) r^2[n] \\
\text{If VAD} = 0 & \quad \hat{\sigma}_n^2[n] = \rho_{\text{var}} \hat{\sigma}_n^2[n-1] + (1 - \rho_{\text{var}}) r^2[n]
\end{align*}
\]  

### Estimating the SNR

#### Signal Combining Algorithms

**Threshold Combining:** This scheme passes to the output the signal having the highest SNR. The figure-of-merit is the ratio \( \text{SNR}_1 / \text{SNR}_2 \). The processing algorithm is simply:

\[
x_{\text{comb}}[n] = \begin{cases} 
  r_1[n], & \frac{\text{SNR}_1}{\text{SNR}_2} \geq 1 \\
  r_2[n], & \frac{\text{SNR}_1}{\text{SNR}_2} < 1 
\end{cases}
\]  

**Maximal Ratio Combining:** Assuming an accurate SNR estimate is available for \( r_1[n] \) and \( r_2[n] \), the maximal ratio combining algorithm combines the two noisy signals such that \( x_{\text{comb}}[n] \) has maximum SNR. The processing algorithm is:

\[
x_{\text{comb}}[n] = \frac{1}{1 + \frac{\text{SNR}_2}{\text{SNR}_1}} \cdot r_1[n] + \frac{1}{1 + \frac{\text{SNR}_1}{\text{SNR}_2}} \cdot r_2[n]
\]  

### Speech Test Vector for Problem 2
Play the speech vector (in the Python folder of the ZIP package) into the FM4 line in. Since this is an 8 ksp file, the sampling rate on the FM4 should be reduced to 8 kHz. You code is going to require quite a few `float32_t` divisions, and the lower sampling rate will help make this a non-issue. All-in-all this reduction will allow for more processing in the ISR.

```python
fs, x = ss.from_wav('OSR_us_000_0030_8k.wav')
# Gain normalize to (-1,1)
x_norm = x/max(abs(x))
```

For testing purposes a double length speech array, that that can be manipulated using the Windows audio player, `speech_test_2x.wav` is convenient for testing. An alternative of course is to use `pyaudio_helper` and the looping object to allow for continuous play. If you decide to go this route, you can add true Gaussian noise in a `pyaudio_helper` callback and making it in stereo will allow all of the signal generation to be done in Python. Note the current C-code implements the noise generation using the simple `uniform` random number generator `rand_int32()`.

```python
ss.to_wav('speech_test_2x.wav',fs,hstack((x_norm,x_norm)))
```

Using the `Audio` control for playback:

```python
Audio(x_norm[:],rate=fs) # play a single/mono wave file
#Audio([x[:,0],x[:,1]],rate=fs) # play a stereo wavefile
```

## - A 1 dB Steps Lookup Table for Use in C Code

The LUT belows allows the gain sliders used to add noise in 1 dB steps, as shown in the above block diagram, in a fast and efficient manner. This is included in the sample file `fm4_loop_intr_Speech_Div_Comb_GUI.c` in the `src` folder of the ZIP package. The noise summing operation is also implemented with GUI sliders `P0` and `P1` controlling the noise level.

```python
GdB = arange(-50,1,1)
10**(GdB/20)
```
Building and Testing a Python Prototype

Before getting into the details of Python prototype, we adopt some simplifications based on the assumptions that the signal level is the same in signals $r_1[n_1]$ and $r_2[n_1]$, only the noise is different. This is significant as we can see by considering the ratio of SNR's:

$$\frac{\text{SNR}_1}{\text{SNR}_2} = \frac{\hat{\sigma}_1^2 / \hat{\sigma}_{n_1}^2}{\hat{\sigma}_2^2 / \hat{\sigma}_{n_2}^2} = \frac{\hat{\sigma}_{n_2}^2}{\hat{\sigma}_{n_1}^2}$$  \hspace{1cm} (13)

All we have to do is estimate the noise-alone variances for channels 1 and 2. Note VAD is still required, but the only variance estimator is required per channel. The block diagram below provides the final details of what needs to be implemented to run on the FM4:
The code below builds up a model of the speech combining system. To better reflect the issues of working with speech samples brought in as `int16_t` values from the FM ADC, we scale the wave file speech values by 2^{15}.

```python
# Add noise to speech vector
r_norm = x_norm + 0.01*randn(len(x_norm))

t = arange(len(r_norm))/8000
plot(t, r_norm)
title(r'US Male Test Speech Vector at $f_s = 8$ kHz')
ylabel(r'Amplitude')
xlabel(r'Time (s)')
grid();
```
# VAD Sensing Using Average Energy Detection
# with Hysteresis thresholding

# Add noise to the single length speech vector
r_norm1 = 0.001*randn(len(x_norm)) + x_norm
r_norm2 = 0.04*randn(len(x_norm)) + x_norm

# Initialize VAD related variables
VAD1 = zeros(len(r_norm1))
VAD2 = zeros(len(r_norm2))
e_avg_v1 = zeros(len(r_norm1))
e_avg_v2 = zeros(len(r_norm2))
VAD_thresh = 200

# Scale the (-1,1) noisy Speech samples to int16_t Levels
r_norm_fp1 = r_norm1*(2**15)
r_norm_fp2 = r_norm2*(2**15)

# Exponential smoother
for n in range(len(r_norm_fp1)):
    e_avg1 = alpha_e*e_avg_old1 + (1-alpha_e)*r_norm_fp1[n]*r_norm_fp1[n]*scale
    e_avg_old1 = e_avg1
e_avg_v1[n] = e_avg1

for n in range(len(r_norm_fp2)):
    e_avg2 = alpha_e*e_avg_old2 + (1-alpha_e)*r_norm_fp2[n]*r_norm_fp2[n]*scale
    e_avg_old2 = e_avg2
\[ e_{avg\_old2} = e_{avg2} \]
\[ e_{avg\_v2[n]} = e_{avg2} \]

# Standard Threshold
if \( e_{avg1} \geq VAD\_thresh \):
    \( VAD1[n] = 1 \)

# Include some hysteresis
elif \( e_{avg\_old1} > VAD\_thresh / 2 \) and \( e_{avg1} > VAD\_thresh / 2 \):
    \( VAD1[n] = 1 \)
if \( e_{avg2} \geq VAD\_thresh \):
    \( VAD2[n] = 1 \)

# Include some hysteresis
elif \( e_{avg\_old2} > VAD\_thresh / 2 \) and \( e_{avg2} > VAD\_thresh / 2 \):
    \( VAD2[n] = 1 \)

Nstart = 50000
Nspan = 50000
t = arange(len(VAD1))/8000
subplot(211)
plot(t[Nstart:Nstart+Nspan], VAD1[Nstart:Nstart+Nspan])
plot(t[Nstart:Nstart+Nspan], e_avg_v1[Nstart:Nstart+Nspan]/2**10)
title(r'US Male Test Speech Vector VAD with $e_{avg}$ Overlay')
ylabel(r'Logic & Energy')
xlabel(r'Time (s)')
legend((r'VAD1', r'Scaled $e_{avg1}$'))
grid();
subplot(212)
plot(t[Nstart:Nstart+Nspan], VAD2[Nstart:Nstart+Nspan])
plot(t[Nstart:Nstart+Nspan], e_avg_v2[Nstart:Nstart+Nspan]/2**10)
title(r'US Male Test Speech Vector VAD with $e_{avg}$ Overlay')
ylabel(r'Logic & Energy')
xlabel(r'Time (s)')
legend((r'VAD2', r'Scaled $e_{avg2}$'))
grid();
tight_layout()
```python
t = arange(len(r_norm1))/8000
Nstart = 50000
Nspan = 50000
subplot(211)
noise_alone_fp1 = r_norm_fp1*(1-VAD1)*(1-VAD2)
plot(t[Nstart:Nstart+Nspan],noise_alone_fp1[Nstart:Nstart+Nspan])
title(r'Noise Alone Estimate (int16_t Levels)')
ylabel(r'Amplitude')
xlabel(r'Time (s)')
grid();
subplot(212)
noise_alone_fp2 = r_norm_fp2*(1-VAD2)*(1-VAD1)
plot(t[Nstart:Nstart+Nspan],noise_alone_fp2[Nstart:Nstart+Nspan])
title(r'Noise Alone Estimate (int16_t Levels)')
ylabel(r'Amplitude')
xlabel(r'Time (s)')
grid();
tight_layout()
```
Part a

Develop real-time C-code for the FM4 that implements the System Block Diagram ... in addition to the additive noise injection that uses the dB LUT described earlier. The starting point will be `fm4_loop_intr_Speech_Div_Comb_GUI.c` as found in the ZIP package `src` folder.

```c
# Recursively estimate the variances
rho_x = 0.9999
var_n1 = 0
var_n_old1 = 0
var_n2 = 0
var_n_old2 = 0

SNR1_by_SNR2 = zeros(len(r_norm_fp1))
for n in range(len(r_norm_fp1)):
    if (1-VAD1[n])*(1-VAD2[n]) == 1:
        # Exponential smoother
        var_n1 = rho_x*var_n_old1 + (1 - rho_x)*r_norm_fp1[n]*r_norm_fp1[n]*scale 
        var_n_old1 = var_n1
        # Exponential smoother
        var_n2 = rho_x*var_n_old2 + (1 - rho_x)*r_norm_fp2[n]*r_norm_fp2[n]*scale 
        var_n_old2 = var_n2

SNR1_by_SNR2[n] = var_n2/var_n1
```
Part b

Verify that VAD is working for Channels 1 and 2 by setting channels 1 and 2 GUI slider dB settings to to -50 dB and -13 dB respectively, then capture a plot similar to the one shown below. The capture will need to be made while playing the speech file `speech_test_2x.wav` into the FM4 line input.

![VAD plot](image)

The effectiveness of VAD in extracting noise only speech samples

Be sure to indicate the threshold value you are using, e.g., `VAD_thres`.

Part c

Continue in the code development to implement the SNR ratio estimator and prepare for a demo. For the demo be ready to play the 2x speech file into your system and have the GUI slider configured as shown below and have watch variables in the Keil debugger to `watch` what is happening with the algorithm as noise levels are changed which speech is playing through the system.
Part d

Audio demo using headphones and/or PC speakers in the lab so that the performance of the two combining schemes can be evaluated. I don't expect perfection with MRC, as these are are challenges in estimating the true SNR ratio accurately when using exponential smoothing. Be creative.