MATLAB Interactive Granular Convolution-based Synthesiser

Ricky Chu

Overview

- My program features the ability to load 2 audio files.
- It features the ability to see the spectrogram of the file(s).
- It allows playback of the files with pitch and tempo adjustment.
- It allows for a basic form of granular convolution.
- It lets you switch between audio layers.
- It allows you to edit both spectrograms separately.
- It allows you to perform ADSR enveloping.
- It allows you to apply the flanger and wah-wah audio effects.

As an extension I've added:

- The ability to perform equalisation
- Extra features for editing the STFT.
  - Reversal of sections
  - Cloning of sections
  - Convolution of sections with other sections

Plotting Spectrogram

To plot the spectrogram:

1. Find the STFT of the signal
2. Get the absolute value of the STFT and divide it by the frame size
3. Plot the values

The STFT function was taken from the labs.

```
stft_signal = my_stft(app, y, frame_size, frame_size, hop_size);
plotSignal(app, stft_signal)

function plotSignal(app, stft_signal)
    specy = abs(stft_signal)/app.frame_size;
    I = imshow(flipud(255*specy), 'Parent', app.UIAxes,
               'XData', [1 app.UIAxes.Position(3)],
               'YData', [1 app.UIAxes.Position(4)]);
    app.UIAxes.XLim = [0 I.XData(2)];
    app.UIAxes.YLim = [0 I.YData(2)];
    colormap(app.UIAxes, jet); %color display
end
```

Pitch and tempo changes

To do the pitch and tempo changes I used the `pvoc` and `resample` functions from the labs. The pitch and tempo values were on a slider, so to find a rational value for the resampling I used an algorithm from this website: https://www.johndcook.com/blog/2010/10/20/best-rational-approximation/

However, this algorithm only works for numbers between 0 to 1, whereas my range goes from 0 to 2.

So to find the rational approximation for the number n, where $1 < n \leq 2$.

I first find the approximation for $n - 1$ to get the values $a, b, \frac{a}{b} = n - 1$.

Then I set a to $a + b$, so now the value is $\frac{a+b}{b} = \frac{a}{b} + \frac{b}{b} = n$

```
pitch_value = max(app.PitchSlider.Value, 0.05);
tempo_value = max(app.TempoSilder.Value, 0.05);

%Get the ratio for pitch adjustment
if pitch_value == 1
    a = 1;
    b = 1;
elseif pitch_value > 1
    [a, b] = ratApprox(app, pitch_value - 1, 300);
    a = a + b;
else
    [a, b] = ratApprox(app, pitch_value, 300);
end

y = my_istft(app, stft_signal, app.frame_size, app.frame_size, app.hop_size);

% Pitch changes
ypvoc = pvoc(app, y, pitch_value, 1024);
ypitch = resample(ypvoc, a, b);

% Tempo changes
yslow = pvoc(app, ypitch, tempo_value, 1024);
```

Granular convolution

To implement granular convolution I performed these steps.

1. Get your first signal S_1.
2. Grab random grains of audio from the second signal S_2
3. Put these random grains together into a new signal S_2'.
4. Apply the FFT to S_2'.
5. Pad or truncate the S_2' to be the same length as S_1
6. Apply the FFT to S_1.
7. Perform element wise multiplication to the FFTs of S_1 and S_2' to get our convolved signal

The `granularise` function was taken from the Labs.

```
function y = graunlarise(app, x, fs)
    % Taken from lab5wk6
    nEv = 400;
    maxL = round(fs*0.02);
    minL = round(fs*0.01);
    Lw = round(fs*0.01);

    Ly=length(x);
    L = round((maxL - minL) * rand(nEv, 1) + minL);
    initIn = ceil((Ly - maxL) * rand(nEv, 1));
    initOut = ceil((Ly - maxL) * rand(nEv, 1));
    endOut = initOut + L - 1;
```

```
    y=zeros(Ly,1);
    % Synthesis
    for k=1:nEv
      grain=grainLn(app, x,initIn(k),L(k),Lw);
      y(initOut(k):endOut(k))= y(initOut(k):endOut(k)) + grain;
    end
end

% Apply granular synthesis to second signal
grains = graunlarise(app, signal_2, fs);

% Trunchate/Pad the 2nd signal to be same length as first
gLen = length(signal_1);
y = ones(1, gLen);
for i=1:gLen
    idx = max(1, mod(i, length(grains)));
    y(i) = grains(idx);
end

% get ffts
s1 = fft(signal_1);
s2 = fft(y);

% Do convolution with the current signal
Y = s1 .* s2';
Y = real(ifft(Y));
Y = Y/max(abs(Y));
```

Spectrogram editing

To implement spectrogram editing:

1. Run `ginput` to get 2 points on the plot
2. Check if either of those 2 points are out of bounds
3. Remap the values from plot space into spectrogram space
4. Create a mask from out values
5. Do element wise multiplication with our mask and our STFT

ADSR Enveloping

Take the Attack, Decay, Sustain and Release values from sliders. Each of these are points in 2D space, with the constrains.

$$A_y, D_y, S_y, R_y > 0$$
$$A_y, D_y, S_y, R_y \leq 1$$
$$A_x < D_x$$
$$D_x < S_x$$
$$S_x < R_x$$

Next we generate a piecewise curve going through the points $(0, 0), (A_x, A_y), (D_x, D_y), (S_x, S_y)$ and (R_x, R_y) with linear interpolation.

Then we generate a 1d mask that consists the y values of the curve and perform element-wise multiplication with our signal.

```
function mask = generateMask(app, len)
    x1 = 0;
    y1 = 0;

    x2 = floor(app.attack(1)*len);
    y2 = app.attack(2);

    x3 = floor(app.decay(1)*len);
    y3 = app.decay(2);

    x4 = floor(app.sustain(1)*len);
    y4 = app.sustain(2);

    x5 = floor(app.release(1)*len);
    y5 = app.release(2);

    attack_grad = y2/x2;
    attack_mask = 0 : x2;
    attack_mask = attack_mask * attack_grad;

    decay_grad = (y3 - y2)/(x3 - x2);
    decay_mask = 0:(x3 - x2);
    decay_mask = (decay_mask * decay_grad) + y2;

    sustain_grad = (y4 - y3)/(x4 - x3);
    sustain_mask = 0 : (x4 - x3);
    sustain_mask = (sustain_mask * sustain_grad) + y3;

    release_grad = (y5 - y4)/(x5 - x4);
    release_mask = 0 : (x5 - x4);
    release_mask = (release_mask * release_grad) + y4;

    %reshape the mask to match the size of original signal.
    %then transpose to make it able to do element wise
    %multiplication
    mask = [attack_mask decay_mask sustain_mask release_mask];
    [~, mlen] = size(mask);
    diff = abs(len - mlen);
    if mlen > len
        mask = mask(1:end - diff);
    else
        mask = [mask zeros(1, diff)];
    end
    mask = mask';
end
```

Flanger effect

For the flanger effect used a basic delay filter, but modulated the delay with a sine wave.

It's described by $y(n) = x(n) + gx[x - M(n)]$

$M(n)$ is a sine wave of the form $M_0 \cdot sin(2\pi nf)$.

We can use these parameters to modify the feeling of the effect.

- M_0 is the average length of the delay.

- f is the flanger frequency

The range of $sin(x)$ is $[-1, 1]$, so in order to prevent referencing negative samples we take $abs(sin(x))$ instead of $sin(x)$.

```
function y = flanger(app, x, Fs, freq, delay)
    %y(n) = x(n) + gx[x - M(n)]
    %M(n) = M0 * [1 + A*sin(2*pi*n*(f/Fs))]
    %f = flang freq (can be from 0.1 to 1hz)
    %A = sweep/excursion
    %M0 = max delay length
    %g = depth control (should be set to 1 for maximum effect)

    %delay is in milliseconds
    g = 0.8;
    M0 = round((delay/1000)*Fs);
    y = zeros(1, length(x) + M0);

    for n = M0:length(x)
        M = abs(M0 * sin(2*pi*n*(freq/Fs)));
        M = round(M);
        y(n) = g*x(n) + g*x(n - M);
    end
end
```

Wah-Wah effect

This was taken from the slides.

It involves creating a triangle wave and using that to modulate a sine wave.

Then pass that sine wave into a state variable filter to filter your audio signal to produce this effect.

Equalisation

For the equaliser, we combine 3 types of filters. A Low shelving filter, a peak filter and a high shelving filter. The code for the low, peak and high shelving filters were taken from DAFX.

We take in equalisation values from sliders, and pass it through the filters in a series

```
%low shelving
y1 = lowshelving(app, y, (2*32)/Fs, app.HzSlider.Value);

%peaking
y2 = peakfilt(app, y1, 2*(64)/Fs, (2*64)/Fs, app.HzSlider_2.Value);
y3 = peakfilt(app, y2, 2*(128)/Fs, (2*128)/Fs, app.HzSlider_3.Value);
y4 = peakfilt(app, y3, 2*(256)/Fs, (2*256)/Fs, app.HzSlider_4.Value);
y5 = peakfilt(app, y4, 2*(512)/Fs, (2*512)/Fs, app.HzSlider_5.Value);
y6 = peakfilt(app, y5, 2*(1024)/Fs, (2*1024)/Fs, app.KHzSlider.Value);
y7 = peakfilt(app, y6, 2*(2048)/Fs, (2*2048)/Fs, app.KHzSlider_2.Value);
y8 = peakfilt(app, y7, 2*(4096)/Fs, (2*4096)/Fs, app.KHzSlider_3.Value);
y9 = peakfilt(app, y8, 2*(8192)/Fs, (2*8192)/Fs, app.KHzSlider_4.Value);

%high shelving
y10 = highshelving(app, y9, (2*16348)/Fs, app.KHzSlider_5.Value);
```

Reversing sections of the STFT

To implement spectrogram reversal:

1. Run `ginput` to get 2 points on the plot
2. Check if either of those 2 points are out of bounds
3. Remap the values from plot space into spectrogram space
4. Create a mask from those values
5. Reverse the values that are masked out in the spectrogram in the x axis.

```
% Generate reversal area
[stft_y, stft_x] = size(stft_signal);
remap_x = @(x_val) max(round((x_val/axes_width) * stft_x), 1);
remap_y = @(y_val) max(round((y_val/axes_height) * stft_y), 1);

x1 = remap_x(min(x(1), x(2)));
x2 = remap_x(max(x(1), x(2)));

y1 = remap_y(min(y(1), y(2)));
y2 = remap_y(max(y(1), y(2)));

% Flip on x axis
reversed_block = stft_signal(y1:y2, x1:x2);
reversed_block = fliplr(reversed_block);
stft_signal(y1:y2, x1:x2) = reversed_block;
```

Cloning sections of the STFT

1. Run `ginput` to get 3 points on the plot.
2. Check if either of the first 2 points are out of bounds.
3. Remap the values from plot space into spectrogram space.
4. Create a mask from those values.
5. Create a copy of the selected region from your mask
6. Copy those values over to the 3rd point

```
w = int32(x2 - x1);
h = int32(y2 - y1);

center_x = remap_x(px);
center_y = remap_y(py);

half_w = idivide(w,2);
half_h = idivide(h,2);

left  = w - half_w;
right = half_w;
down  = h - half_h;
up    = half_h;

xp1 = center_x - left;
xp2 = center_x + right;
yp1 = center_y - down;
yp2 = center_y + up;

stft_signal(yp1:yp2, xp1:xp2) = cloned_block;
```

Convolution of sections of the STFT

1. Run `ginput` to get 3 points on the plot.
2. Check if either of the first 2 points are out of bounds.
3. Remap the values from plot space into spectrogram space.
4. Create a mask from those values.
5. Create a copy of the selected region from your mask
6. Put that mask over the 3rd point to get your second set of values.7. Convole the 2 set of values.

```
w = int32(x2 - x1);
h = int32(y2 - y1);

center_x = remap_x(px);
center_y = remap_y(py);

half_w = idivide(w,2);
half_h = idivide(h,2);

left  = w - half_w;
right = half_w;
down  = h - half_h;
up    = half_h;

xp1 = center_x - left;
xp2 = center_x + right;
yp1 = center_y - down;
yp2 = center_y + up;

stft_signal(yp1:yp2, xp1:xp2) = cloned_block;


conv_block = stft_signal(yp1:yp2, xp1:xp2);
new_block = conv_block .* orig_block;
stft_signal(yp1:yp2, xp1:xp2) = new_block;
```