Real-Time Streaming Spectral Convolution Filtering

These comments describe the algorithms utilized in the following c code that implements real–time filtering of PCM audio samples.

let $x[n]$ be a a vector of PCM audio samples at a given sampling frequency. Let $h[n]$ be the impulse response of the linear filter. The filter output $y[n]$ is then defined as the convolution of the input audio samples x and the impulse response of the filter h:

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]$$

where

$$\begin{array}{lll} x & \text{is length} & L \\ h & \text{is length} & P \\ y & \text{is length} & L+P-1 \end{array}$$

On of the key observations is that this time–domain convolution can be performed efficiently using the Fast Fourier Transform (FFT) by utilizing zero padding to $L+P-1$ of x and h.

MATLAB Pseudo-code convolution by FFT

$$\begin{aligned} x_{\mathrm{p}} &\leftarrow [x, \texttt{zeros}(\texttt{length}(P)-1)] \\ h_{\mathrm{p}} &\leftarrow [h, \texttt{zeros}(\texttt{length}(L)-1)] \\ X &\leftarrow \texttt{fft}(x_{\mathrm{p}}) \\ H &\leftarrow \texttt{fft}(h_{\mathrm{p}}) \\ Y &\leftarrow X \ \texttt{.*}\ H \\ y &\leftarrow \texttt{ifft}(Y) \end{aligned}$$

Note that even though $x[n]\,, h[n] \in \mathbb{R}$, $X[n], H[n] \in \mathbb{C}$ (i.e X and H are complex numbers).

An even further efficiency gain can be realize by noting that since the PCM input samples x are real, the corresponding FFT output has Hermitian symmetry. The FFT routine can then use about half the memory and operations to compute both the forward and inverse FFT. The FFT libraries used in this implementation exploit the Hermitian symmetry.

Another key observation is that it is possible to utilize block processing on convolution algorithms.

This is necessary for real–time processing of audio signals. This code utilizes the overlap–save method of block convolution, which is described on page 587 of the industry standard textbook Oppenheim and Schafer " This procedure is called the overlap–save method because the input segments overlap, so that each succeeding input section consists of $(L - P + 1)$ new points and $(P - 1)$ points saved from the previous section."

A key point to note here is that the "latency" of the digital filter is related to the chunk length $L - P + 1$, and not the length of the impulse response h. This allows us to use filters with very long impulse response lengths while still guaranteeing a low audio latency, which is important for professional audio applications. It also allows us to implement IIR (infinite impulse response) filters by truncating the impulse response when the impulse response decays to below the precision of the D/A converters. With this observation, it is possible to implement both IIR filters and linear–phase FIR filters in the same processing step.

The overlap-save block convolution utilizes rectangular windows, so when implemented properly, it is mathematically equivalent to a single convolution. Practically, this means there are no audio artifacts that arise from utilizing block processing to implement convolution

The highly–overlapped chunks are spaced to eliminate the circular convolution artifacts:

$$\begin{aligned} \texttt{chunklength} &\quad \text{is length} \quad L - P + 1 \\ \texttt{fftlength} &\quad \text{is length} \quad L + P - 1 \end{aligned}$$

$$P = \frac{-(\texttt{chunklength} - (\texttt{fftlength} + 2))}{2}\ , \quad L = \texttt{chunk} + (P - 1)$$

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]$$

$$\begin{aligned} x &\quad \text{is length} \quad N \\ x_r &\quad \text{is length} \quad L \\ h &\quad \text{is length} \quad P \end{aligned}$$

where

$$L > P$$

(which is necessary for a causal filter).), for $r = 0, 1, 2, \ldots$

$$x_r[n] = x[n + r(L - P + 1) - P + 1], \quad 0 \le n \le (L - 1)$$

this is the "blocks" or "chunks" of the PCM A/D input stream x

$$y_{rp}[n] = \sum_{k=-\infty}^{\infty} x_r[k]h[n-k]$$

The chunks are convolved with the impulse response which, as noted, can be performed using the FFT. Also note, the y_{rp} p reminds us that time aliasing has occurred due to circular convolution. The valid output samples y are then filtered out and placed into the output stream:

$$y_r[n] = \begin{cases} y_{rp}[n], & (P-1) \le n \le (L-1) \\ 0, & \text{otherwise} \end{cases}$$

$$y[n] = \sum_{r=0}^{\infty} y_r[n - r(L-P+1) + P - 1]$$

which is the real–time output of the streaming block convolution algorithm. In the algorithm implemented in the c code, it is noted that the x_r forms a FIFO Queue data structure which can be stored statically for greater efficiency.

Discrete Fourier Transform, FFT, and Hermetian Symmetry

[DTSP p. 543 8.10, 8.11, 8.12 (but with m instead of k as index variable) with W folded into equation (same as `fftw3`) but with different index variables]

$$H[m] = \sum_{n=0}^{\texttt{fftlength}-1} e^{-j\frac{2\pi mn}{N}} h_{\texttt{p}}[n] \tag{1}$$

$$X_{rp}[m] = \sum_{n=0}^{\texttt{fftlength}-1} e^{-j\frac{2\pi mn}{N}} x_{rp}[n] \tag{2}$$

$$Y_{rp} = \sum_{n=0}^{\texttt{fftlength}-1} X_{rp}[n] * H[n] \tag{3}$$

$$y_{rp}[n] = \frac{1}{N} \sum_{m=0}^{\texttt{fftlength}-1} e^{j\frac{2\pi mn}{N}} Y_{rp}[m] \tag{4}$$

where (1) and (2) are the forward Discrete Fourier Transform's (DFT) and (4) is the reverse DFT, and $j = \sqrt{-1}$. (3) is a complex vector multiplied elementwise by a complex vector. The $h_{\texttt{p}}$ notation reminds us that h is padded during `FFT` convolution. Also note that in the code x_{rp} is a zero–padded x_r, which is length(L+P-1) which is `fftlength`.

Since both the input x and the impulse response h are real, X and H have Hermitian Symmetry, so we only need to form $X_{\frac{1}{2}}$ and $H_{\frac{1}{2}}$, which are length $\left(\frac{\texttt{fftlength}}{2} + 1\right)$. If f_s is the PCM sampling frequency, then the Nyquist

frequency f_{ny} is $\frac{f_s}{2}$. This corresponds to continuous time domain frequencies

$$f_{ct}[n] = \frac{f_s\ n}{\texttt{fftlength}} \quad n = 0, 1, 2 \ldots \left(\frac{\texttt{fftlength}}{2}\right)$$

Because the PCM audio samples $x[n]$ are real numbers, the hermitian symmetry means that H is equal to

$$H[n] = \begin{cases} H_{\frac{1}{2}}[n], & n = 0, 1, 2 \ldots \left(\frac{\texttt{fftlength}}{2}\right) \\ H^*_{\frac{1}{2}}[\texttt{fftlength} - n], & n = \left(\frac{\texttt{fftlength}}{2} + 1\right), \ldots, \texttt{fftlength-1} \end{cases}$$

where $H^*_{\frac{1}{2}}$ means the complex conjugate of $H_{\frac{1}{2}}$ (i.e. if $z = a + b\,i$, $z^* = a - b\,i$). Note that `MATLAB` indexs vectors from one instead of zero, so the code looks slightly different.

```
%% Psuedo MATLAB code to form H from Hermitian symmetry
fs = 96e3; % PCM sampling frequency
fflength = 2^14;
n = 0:(fftlength/2);
n = n';
f2 = (fs * n) / fftlength;
H2 = zeros((fftlength/2)+1,1);
H2 = createH(f2);
H = zeros(fftlength,1);
H(1:fftlength/2 + 1) = H2;
H(fftlength/2 + 2:end) = conj(flipud(H2(2:fftlength/2)));
```

```
/*********************************************/
/************                 *****************/
/************ C CODE LISTING *****************/
/************                 *****************/
/*********************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <complex.h>
/* this needs to be included after complex.h, in order to  */
/* use c99 _Complex datatypes in fftw3 */
#include "fftw3.h"

void psmfilter(
               float * ychunk,              // Output:  PCM samples
               float * chunk,               // Input: PCM samples
               float _Complex * restrict H, // Input: Frequency Response - length( (fftlength /2) + 1) (Hermetian symmetry).
```

note: H is OVERWRITTEN, replaced by H/fftlength

```
               int chunklength,             // length of chunk and ychunk
               int fftlength,               // length of FFT
               int initializeflag           // 0 to initialize memory, 1 to process, -1 to cleanup
               ) {

  static float   * xrp;          // FIFO queue
  static float _Complex *  Xrp;
  static float _Complex *  Yrp;
  static float  *  yrp;
  static int L,P;
  static float fftlengthf;
  static fftwf_plan plan_forward;
  static fftwf_plan plan_backward;
  static int fftlengthhalf;

  int iter;
  int i;

  /* call one before starting processing to create FFTW plans, initialze xrp FIFO Queue to zero */
  if (initializeflag == 0) {

    fftlengthhalf = (fftlength / 2) + 1;

    xrp = (float *) fftwf_malloc(fftlength * sizeof(float));
    Xrp = (float _Complex *) fftwf_malloc(fftlengthhalf * sizeof(float _Complex));
    Yrp = (float _Complex *) fftwf_malloc(fftlengthhalf * sizeof(float _Complex));
    yrp = (float *) fftwf_malloc(fftlength * sizeof(float));

    P = (-1 * (chunklength - (fftlength + 2))) / 2;
    L = chunklength + (P - 1);

    fftlengthf = (float)fftlength;

    /* pre-normalize H (fftw does not do it for you */
    for (iter = 0 ; iter < fftlengthhalf ; iter++) H[iter] = H[iter] / fftlengthf;

    fftwf_cleanup(); // clear fftw plans out of memory

    /* create FFTW plans: SINGLE PRECISION FFT using SIMD */
    /* these exploite the Hermitian symmetry to use 1/2 the FLOPS and data storage */

    plan_forward = fftwf_plan_dft_r2c_1d(fftlength,xrp,Xrp,FFTW_MEASURE);
    plan_backward = fftwf_plan_dft_c2r_1d(fftlength,Yrp,yrp,FFTW_MEASURE);
```

```
  for (iter = 0 ; iter < fftlength ; iter++ ) xrp[iter] = 0.0f;

  return ;
}

/* clean up memeory before restart */
if (initializeflag < 0 ) {

  fftwf_free(xrp);
  fftwf_free(Xrp);
  fftwf_free(Yrp);
  fftwf_free(yrp);
  fftwf_destroy_plan(plan_forward);
  fftwf_destroy_plan(plan_backward);

  return ;
}
```

```
  /*************************************/
  /*********                   ***********/
  /**********  MAIN ALGORITHM ***********/
  /*********                   ***********/
  /*************************************/

  /* copy PCM input samples to FIFO Queue (its real in this version) */

  memcpy(&xrp[L - chunklength],chunk,chunklength * sizeof(float));

  /* forward FFT exploiting Hermitian Symmetry */

  fftwf_execute_dft_r2c(plan_forward,xrp,Xrp);

  /* Complex Multiplication of X and H, either using floating point unit or Altivec Unit */

  for ( i = 0 ; i < fftlengthhalf ; i++)   Yrp[i] = Xrp[i] * H[i] ;

  /*  note FFTW does not normalize for you , H is divided by fftlength in the initialization routine above */
  /* backward FFT exploiting Hermtian Symmetry */

  fftwf_execute_dft_c2r(plan_backward,Yrp,yrp);

  /* pick out the valid samples from the circular convolution */

  memcpy(ychunk,&yrp[(P-1)],chunklength * sizeof(float));

  /* overlaping memory move (tricky): update input sample  FIFO Queue */

  memmove(xrp,&xrp[chunklength],(L - chunklength ) * sizeof(float));

}
```

```
---------------------------------------------------------------
End of MSLI Literate TeX processing on file: psmfilter.c
Printed on: Mon Aug  7 16:25:02 PDT 2017
git info:
path: Spectral_Convolution/psmfilter.c
SHA1: 33236ba1d5b442d6903aa119e7cc96a484baa2bb
mode: 100644
url :
root@ellipticalcow.com:git/perrin2013.git

---------------------------------------------------------------
```