ECE 3640 - Discrete-Time Signals and Systems
Convolution and Filtering in C

Jake Gunther

Spring 2015

UtahStateUniversity

Department of Electrical & Computer Engineering

outline

- processing pre-recorded signals and real-time signals
- applications: filtering pre-recorded 1D signal (zero padding)
- applications: filtering real-time 1D signal (linear shift and circular shift buffering)
- applications: filtering 2D signals (spatial filtering)
- applications: filtering 3D signals (temporal filtering)

processing pre-recorded and real-time signals

- a **pre-recorded/finite-length signal** can be loaded into memory (if there is enough memory) and processed all at once

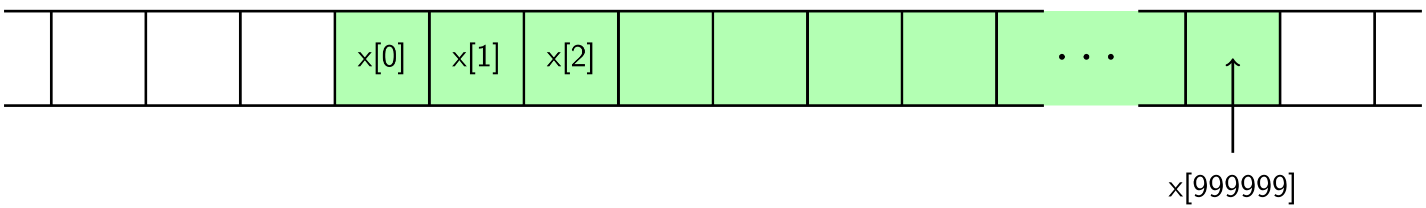
- a **real-time/infinite-length signal** must be processed as the samples become available (only a few samples in memory at any time)

processing pre-recorded signals

processing pre-recorded signals

- a **pre-recorded/finite-length signal** can be loaded into memory (if there is enough memory) and processed all at once

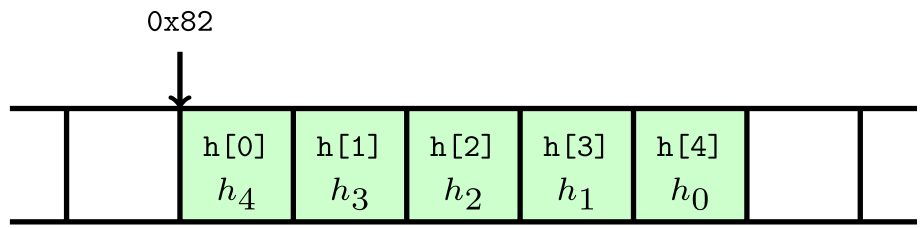
```
short x[1000000];
FILE *fid = fopen("datafile.bin","rb");
fread(x,sizeof(short),1000000,fid);
fclose(fid);
// do something to this signal
// and write out the result
```



application: discrete-time filtering

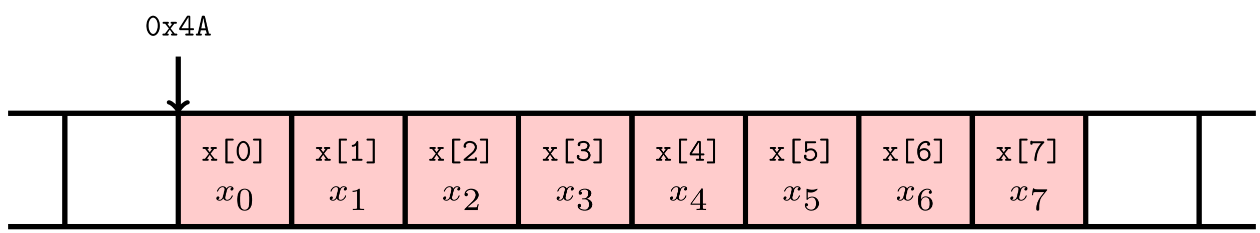
- assume finite impulse response h_n with length $L = 5 : h_0, h_1, h_2, h_3, h_4$
- suppose the impulse response is stored in reverse order in memory in array `h`

`float h[5] = { ...  };` ⟶



- let the finite length input signal $x_n, n = 0, 1, \cdots, 7$ be stored in natural order in memory in array `x`

`float x[8] = { ...  };` ⟶



review of convolution

$$y_n = \sum_{k=0}^{4} x_{n-k} h_k = x_{n-4}h_4 + x_{n-3}h_3 + x_{n-2}h_2 + x_{n-1}h_1 + x_n h_0$$

$$x_n = 0 \text{ for } n < 0 \text{ and } n > 7$$

$$\begin{aligned}
y_0 &= 0\,h_4 + 0\,h_3 + 0\,h_2 + 0\,h_1 + x_0 h_0 \\
y_1 &= 0\,h_4 + 0\,h_3 + 0\,h_2 + x_0 h_1 + x_1 h_0 \\
y_2 &= 0\,h_4 + 0\,h_3 + x_0 h_2 + x_1 h_1 + x_2 h_0 \\
y_3 &= 0\,h_4 + x_0 h_3 + x_1 h_2 + x_2 h_1 + x_3 h_0 \\
y_4 &= x_0 h_4 + x_1 h_3 + x_2 h_2 + x_3 h_1 + x_4 h_0 \\
y_5 &= x_1 h_4 + x_2 h_3 + x_3 h_2 + x_4 h_1 + x_5 h_0 \\
y_6 &= x_2 h_4 + x_3 h_3 + x_4 h_2 + x_5 h_1 + x_6 h_0 \\
y_7 &= x_3 h_4 + x_4 h_3 + x_5 h_2 + x_6 h_1 + x_7 h_0 \\
y_8 &= x_4 h_4 + x_5 h_3 + x_6 h_2 + x_7 h_1 + 0\,h_0 \\
y_9 &= x_5 h_4 + x_6 h_3 + x_7 h_2 + 0\,h_1 + 0\,h_0 \\
y_{10} &= x_6 h_4 + x_7 h_3 + 0\,h_2 + 0\,h_1 + 0\,h_0 \\
y_{11} &= x_7 h_4 + 0\,h_3 + 0\,h_2 + 0\,h_1 + 0\,h_0
\end{aligned}
\quad \Leftrightarrow \quad
\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \end{bmatrix}
=
\begin{bmatrix}
0 & 0 & 0 & 0 & x_0 \\
0 & 0 & 0 & x_0 & x_1 \\
0 & 0 & x_0 & x_1 & x_2 \\
0 & x_0 & x_1 & x_2 & x_3 \\
x_0 & x_1 & x_2 & x_3 & x_4 \\
x_1 & x_2 & x_3 & x_4 & x_5 \\
x_2 & x_3 & x_4 & x_5 & x_6 \\
x_3 & x_4 & x_5 & x_6 & x_7 \\
x_4 & x_5 & x_6 & x_7 & 0 \\
x_5 & x_6 & x_7 & 0 & 0 \\
x_6 & x_7 & 0 & 0 & 0 \\
x_7 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix} h_4 \\ h_3 \\ h_2 \\ h_2 \\ h_1 \\ h_0 \end{bmatrix}$$

review of convolution

				y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}
0	0	0	0	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	0	0	0	0
h_4	h_3	h_2	h_1	h_0											
	h_4	h_3	h_2	h_1	h_0										
		h_4	h_3	h_2	h_1	h_0									
			h_4	h_3	h_2	h_1	h_0								
				h_4	h_3	h_2	h_1	h_0							
					h_4	h_3	h_2	h_1	h_0						
						h_4	h_3	h_2	h_1	h_0					
							h_4	h_3	h_2	h_1	h_0				
								h_4	h_3	h_2	h_1	h_0			
									h_4	h_3	h_2	h_1	h_0		
										h_4	h_3	h_2	h_1	h_0	
											h_4	h_3	h_2	h_1	h_0
$\leftarrow \; L-1 \; \rightarrow$				$\leftarrow \; N \; \rightarrow$								$\leftarrow \; L-1 \; \rightarrow$			

1. zero pad both ends of input sequence with $L-1$ zeros, where L is the length of the impulse response

2. compute inner product for every shift of time-reversed impulse response

convolution code fragment

```
#define Lh  5     /* length of impulse response */
#define Lx 10     /* length of input signal     */
int Ly = Lx+  (Lh-1); /* length of convolution result */
int Lz = Lx+2*(Lh-1); /* length of zero padded input */
float *x = calloc(sizeof(float),Lz);
float *y = calloc(sizeof(float),Ly);
FILE *fx = fopen( "inputfile","rb");
FILE *fy = fopen("outputfile","wb");
// read data into x array with offset
// x is zero padded on both ends
// [ 0 ... 0 | x[0] ... x[Lx-1] | 0 ... 0 ]
// [  Lh - 1 |         Lx       |  Lh - 1 ]
fread(x+Lh-1,sizeof(float),Lx,fx);
int i, j;
for(i=0; i<Ly; i++) {
  for(j=0; j<Lh; j++) {
    y[i] += h[j]*x[i+j]; // multiply and accumulate (MAC)
  }
}
fwrite(y,sizeof(float),Ly,fy);
fclose(fx);
fclose(fy);
```

processing real-time signals
linear shift buffer

processing real-time signals

- a **real-time/infinite-length signal** must be processed as the samples become available

- only a few samples in memory at any time

real-time signal example

- samples of signal **shift left** across array in memory in **natural order**
- impulse response in **time-reverse** order

	h[5]	h[4]	h[3]	h[2]	h[1]	h[0]
t=-1	0	0	0	0	0	0
t=0	0	0	0	0	0	x[0]
t=1	0	0	0	0	x[0]	x[1]
t=2	0	0	0	x[0]	x[1]	x[2]
t=3	0	0	x[0]	x[1]	x[2]	x[3]
t=4	0	x[0]	x[1]	x[2]	x[3]	x[4]
t=5	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
t=6	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]
			⋮			
t=n	x[n-5]	x[n-4]	x[n-3]	x[n-2]	x[n-1]	x[n]

real-time signal example

- samples of signal **shift right** across array in memory in **time-reversed order**
- impulse response in **natural order**

	h[0]	h[1]	h[2]	h[3]	h[4]	h[5]
t=-1	0	0	0	0	0	0
t=0	x[0]	0	0	0	0	0
t=1	x[1]	x[0]	0	0	0	0
t=2	x[2]	x[1]	x[0]	0	0	0
t=3	x[3]	x[2]	x[1]	x[0]	0	0
t=4	x[4]	x[3]	x[2]	x[1]	x[0]	0
t=5	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]
t=6	x[6]	x[5]	x[4]	x[3]	x[2]	x[1]
				⋮		
t=n	x[n]	x[n-1]	x[n-2]	x[n-3]	x[n-4]	x[n-5]

shift buffer code example

- this code snippit illustrates time-reversed buffering

```
#define M 6
short x[M], i;
FILE *fid=fopen("datafile","rb");
fread(x,sizeof(short),1,fid); // read in a sample
while(!feof(fid))
{
   // do something to this signal
   for(i=M-1 i>0; i--) { x[i]=x[i-1]; } // shift right
   fread(x,sizeof(short),1,fid); // read in next sample
}
fclose(fid);
```

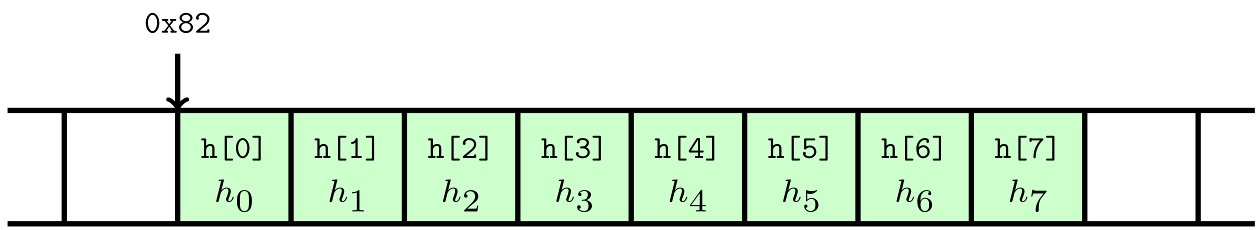
- shift buffers require a loop for moving data

- each sample in the buffer is visited

application: discrete-time filtering using linear buffer

- assume finite impulse response h_n with length $L = 8 : h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7$

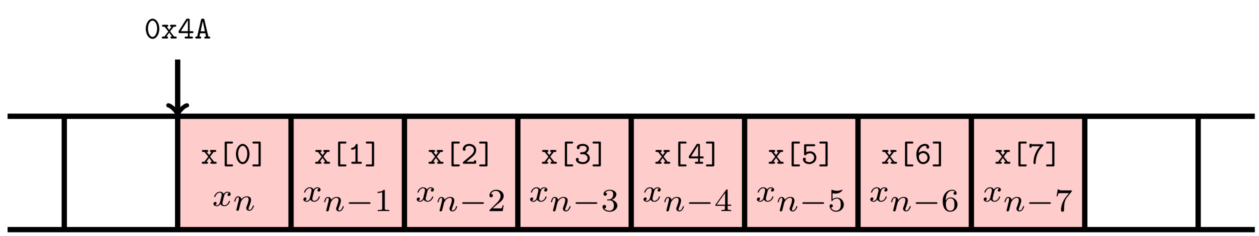
- suppose impulse response is stored in memory in array `h` in natural order



- let the input signal be $x_n, \quad n = 0, 1, 2, 3, \cdots$

- suppose x_n is processed using linear shift buffer in time reverse order

- at time n, data array `x` in memory holds $x_n, x_{n-1}, \cdots, x_{n-L+1}$



application: discrete-time filtering using linear buffer

- linear time-reverse buffering leads to alignment of impulse response and data
- only need to do an inner product between arrays to compute convolution result, i.e. filter output

	h[0] h_0	h[1] h_1	h[2] h_2	h[3] h_3	h[4] h_4	h[5] h_5	h[6] h_6	h[7] h_7	
	x[0] x_n	x[1] x_{n-1}	x[2] x_{n-2}	x[3] x_{n-3}	x[4] x_{n-4}	x[5] x_{n-5}	x[6] x_{n-6}	x[7] x_{n-7}	

- discrete-time convolution formula

$$
\begin{aligned}
y_n &= \sum_{k=0}^{7} h_k x_{n-k} \\
&= h_0 x_n + h_1 x_{n-1} + h_2 x_{n-2} + \cdots + h_7 x_{n-7} \\
&= \texttt{h[0]*x[0] + h[1]*x[1] + h[2]*x[2] + ... + h[7]*x[7]}
\end{aligned}
$$

filtering code fragment using linear time-reversed buffering

```
#define L 7
float h[L] = {0.08, 0.25, 0.64, 0.95, 0.95, 0.64, 0.25, 0.08};
float x[L] = {0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00};
float y;
int k;
FILE *fx=fopen( "inputfile","rb");
FILE *fy=fopen("outputfile","wb");
fread(x,sizeof(float),1,fx); // read in first sample
while(!feof(fx)) {
  for(y=0.0, k=0; k<L; k++) {
    y += h[k]*x[k]; // MAC
  }
  for(k=L-1; k>0; k--} {
    x[k] = x[k-1]; // shift
  }
  fwrite(&y,sizeof(float),1,fy); // save output
  fread(x,sizeof(float),1,fx); // read in next sample
}
fclose(fx);
fclose(fy);
```

- MAC and shift in separate loops

filtering code fragment using linear time-reversed buffering

```
#define L 7
float h[L] = {0.08, 0.25, 0.64, 0.95, 0.95, 0.64, 0.25, 0.08};
float x[L] = {0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00};
float y;
int k;
FILE *fx=fopen( "inputfile","rb");
FILE *fy=fopen("outputfile","wb");
fread(x,sizeof(float),1,fx); // read in first sample
while(!feof(fx)) {
  for(y=0.0, k=L-1; k>0; k--) {
    y += h[k]*x[k]; // MAC
    x[k] = x[k-1]; // shift
  }
  y += h[0]*x[0]; // last MAC
  fwrite(&y,sizeof(float),1,fy); // save output
  fread(x,sizeof(float),1,fx); // read in next sample
}
fclose(fx);
fclose(fy);
```

- MAC and shift combined into one loop

processing real-time signals
circular shift buffer

real-time signal example

- samples of signal shift **circularly** through array in **time-reversed order**

t=-1	0	0	0	0	0	0
t=0	0	0	0	0	0	x[0]
t=1	0	0	0	0	x[1]	x[0]
t=2	0	0	0	x[2]	x[1]	x[0]
t=3	0	0	x[3]	x[2]	x[1]	x[0]
t=4	0	x[4]	x[3]	x[2]	x[1]	x[0]
t=5	x[5]	x[4]	x[3]	x[2]	x[1]	x[0]
t=6	x[5]	x[4]	x[3]	x[2]	x[1]	x[6]
t=7	x[5]	x[4]	x[3]	x[2]	x[7]	x[6]
t=8	x[5]	x[4]	x[3]	x[8]	x[7]	x[6]

- each sample stays in place in the array and is then overwritten by a new sample
- circular buffering reduces overhead associated with shifting samples but requires extra care when indexing

circular buffering code example

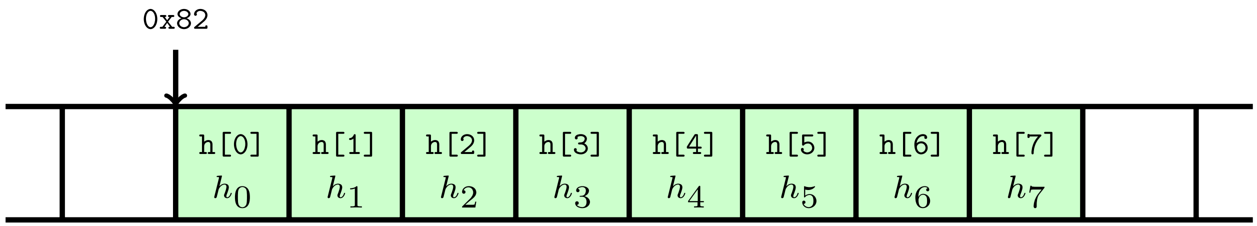
- this code snippit illustrates time-reversed circular buffering

```
#define M 6
short x[M], i=M-1;
FILE *fid=fopen(``datafile'',''rb'');
fread(&(x[i]),sizeof(short),1,fid); // read in a sample
while(!feof(fid)) {
    // do something to this signal
    i += M-1; i %= M; // circular index
    fread(&(x[i]),sizeof(short),1,fid); // read in next sample
}
fclose(fid);
```

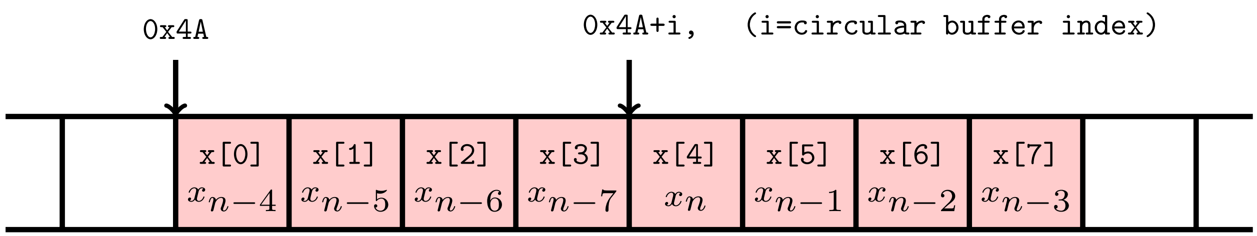
- circular indexing avoids moving the data
- only expense is circular index computation
- same information available linear and circular buffering, but in a different order

application: discrete-time filtering using circular buffer

- assume finite impulse response h_n with length $L = 8 : h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7$
- suppose impulse response is stored in memory in array `h` in natural order



- let the input signal be $x_n, \;\; n = 0, 1, \cdots$
- suppose x_n is processed using circular buffer in time reverse order
- at time n, data array `x` in memory holds $x_n, x_{n-1}, \cdots, x_{n-L+1}$



application: discrete-time filtering using circular buffer

- circular time-reverse buffering leads to data in scrambled order
- need complex indexing in computing inner product between arrays to compute convolution result, i.e. filter output

	h[0] h_0	h[1] h_1	h[2] h_2	h[3] h_3	h[4] h_4	h[5] h_5	h[6] h_6	h[7] h_7	
	x[0] x_{n-4}	x[1] x_{n-5}	x[2] x_{n-6}	x[3] x_{n-7}	x[4] x_n	x[5] x_{n-1}	x[6] x_{n-2}	x[7] x_{n-3}	

- discrete-time convolution formula

$$y_n = \sum_{k=0}^{7} h_k x_{n-k} = h_0 x_n + h_1 x_{n-1} + h_2 x_{n-2} + h_3 x_{n-3} + h_4 x_{n-4} + h_5 x_{n-5} + h_6 x_{n-6} + h_7 x_{n-7}$$

```
= h[0]*x[(0+4)%8] + h[1]*x[(1+4)%8] + h[2]*x[(2+4)%8] + h[3]*x[(3+4)%8] +
  h[4]*x[(4+4)%8] + h[5]*x[(5+4)%8] + h[6]*x[(6+4)%8] + h[7]*x[(7+4)%8]
= h[0]*x[4] + h[1]*x[5] + h[2]*x[6] + h[3]*x[7] +
  h[4]*x[0] + h[5]*x[1] + h[6]*x[2] + h[7]*x[3]
```

- start indexing `x` at `i=4` instead of `0` and wrap when `i>7` (i.e. modulo arithmetic)

filtering code fragment using circular time-reversed buffering

```
#define L 7
float h[L] = {0.08, 0.25, 0.64, 0.95, 0.95, 0.64, 0.25, 0.08};
float x[L] = {0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00};
float y;
int k, i=L-1;
FILE *fx=fopen( "inputfile","rb");
FILE *fy=fopen("outputfile","wb");
fread(x+i,sizeof(float),1,fx); // read in first sample
while(!feof(fx)) {
  for(y=0.0, k=0; k<L; k++) {
    y += h[k]*x[(k+i) % L]; // MAC with circular indexing
  }
  i += L-1; i %= L; // update circular index
  fwrite(&y,sizeof(float),1,fy); // save output
  fread(x+i,sizeof(float),1,fx); // read in next sample
}
fclose(fx);
fclose(fy);
```

- circular buffering avoids shifting the data

image processing & 2D spatial filtering

application: 2D spatial filtering

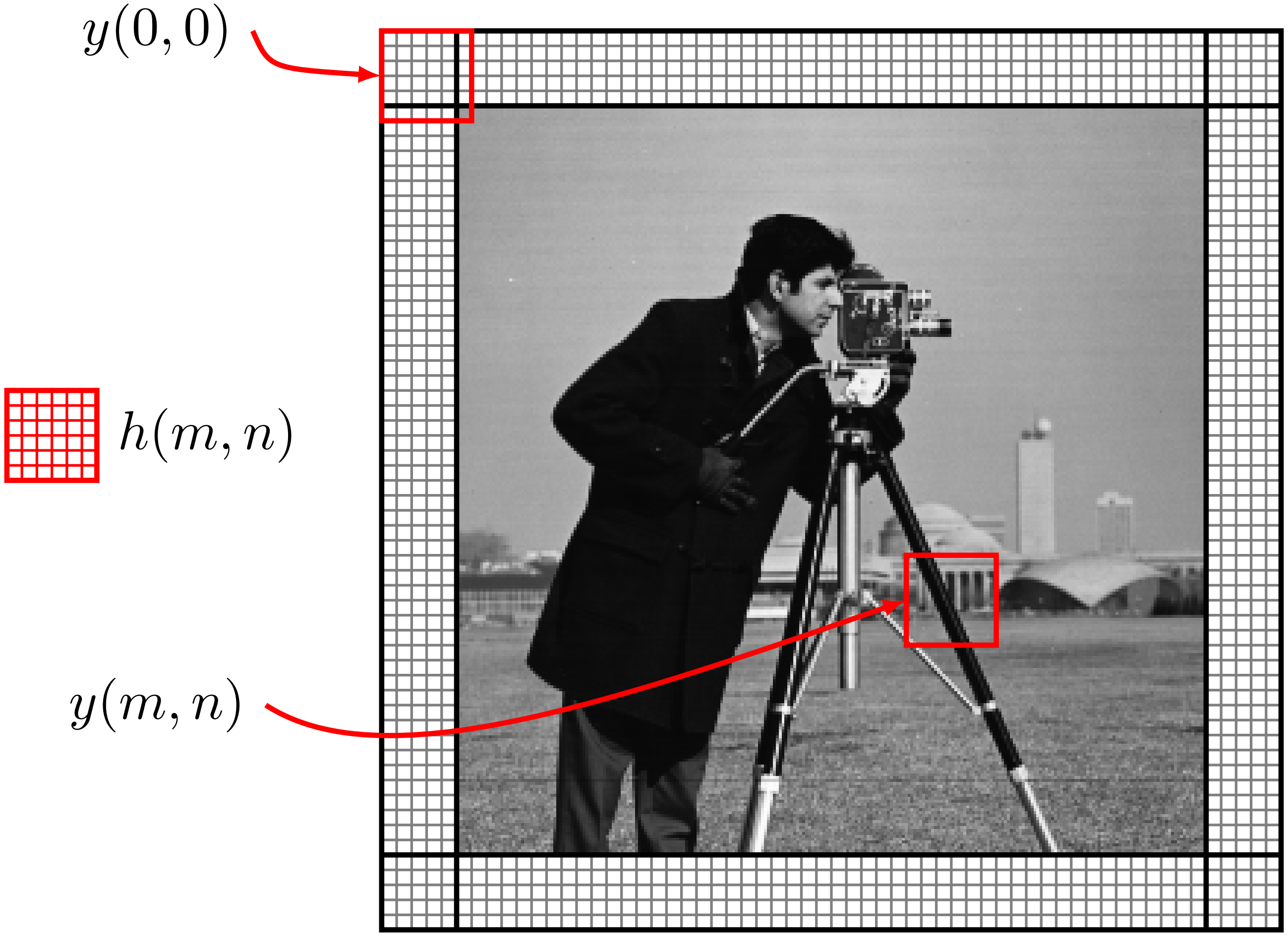
- all the 1D filtering and convolution ideas can be extended to 2D
- an image is a finite length (i.e. pre-recorded) 2D signal
- assume the input image is $R_x \times C_x$ and the impulse response is $R_h \times C_h$
- the convolution result is $R_y \times C_y$, where

$$R_y = R_x + R_h - 1, \qquad\qquad C_y = C_x + C_h - 1$$

- assume the entire image can be loaded into memory
- as in convolution of pre-recorded 1D signals, zero pad by $R_h - 1$ rows of pixels on top and bottom and $C_h - 1$ columns of pixels on left and right sides of the image, then do 2D convolution
- the padded image is $R_z \times C_z$, where

$$R_z = R_x + 2(R_h - 1), \qquad\qquad C_z = C_x + 2(C_h - 1)$$

application: 2D spatial filtering



$$y(m,n) = \sum_{k=0}^{M_h-1} \sum_{l=0}^{N_h-1} h(k,l)x(m-k,n-l)$$

- shift the impulse response around the image and MAC at each position

application: 2D spatial filtering

- the impulse response should be stored in spatially reversed order

$$\texttt{h[m][n]} = h(5-m, 5-n)$$

	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$m=0$	$h(5,5)$	$h(5,4)$	$h(5,3)$	$h(5,2)$	$h(5,1)$	$h(5,0)$
$m=1$	$h(4,5)$	$h(4,4)$	$h(4,3)$	$h(4,2)$	$h(4,1)$	$h(4,0)$
$m=2$	$h(3,5)$	$h(3,4)$	$h(3,3)$	$h(3,2)$	$h(3,1)$	$h(3,0)$
$m=3$	$h(2,5)$	$h(2,4)$	$h(2,3)$	$h(2,2)$	$h(2,1)$	$h(2,0)$
$m=4$	$h(1,5)$	$h(1,4)$	$h(1,3)$	$h(1,2)$	$h(1,1)$	$h(1,0)$
$m=5$	$h(0,5)$	$h(0,4)$	$h(0,3)$	$h(0,2)$	$h(0,1)$	$h(0,0)$

code fragment for 2D convolution

```
#define X(u,v) x[(u)*Cz+(v)] /* 2D to 1D index conversion */
#define H(u,v) h[(u)*Ch+(v)] /* 2D to 1D index conversion */
#define Y(u,v) y[(u)*Cy+(v)] /* 2D to 1D index conversion */

int Ry = Rx +   (Rh-1); // length of convolution result
int Cy = Cx +   (Ch-1); // length of convolution result
int Rz = Rx + 2*(Rh-1); // length of doubly padded input array
int Cz = Cx + 2*(Ch-1); // length of doubly padded input array

float *h = (float*)calloc(sizeof(float),Rh*Ch);
float *x = (float*)calloc(sizeof(float),Rz*Cz);
float *y = (float*)calloc(sizeof(float),Ry*Cy);

for(k=0; k<Ry; k++) {   // loop over rows in output image
  for(l=0; l<Cy; l++) { // loop over cols in output image
    for(tmp=0.0, i=0; i<Rh; i++) {
      for(j=0; j<Ch; j++) {
    tmp += H(i,j)*X(k+i,l+j); // MAC
      }
    }
    Y(k,l) = tmp;
  }
}
```

application: temporal filtering of video (3D)

assignment

assignment

1. process pre-recorded audio

2. process real-time audio using circular buffer

3. 2D spatial filtering for edge detection

4. 1D filtering in each pixel of a video using circular frame buffer

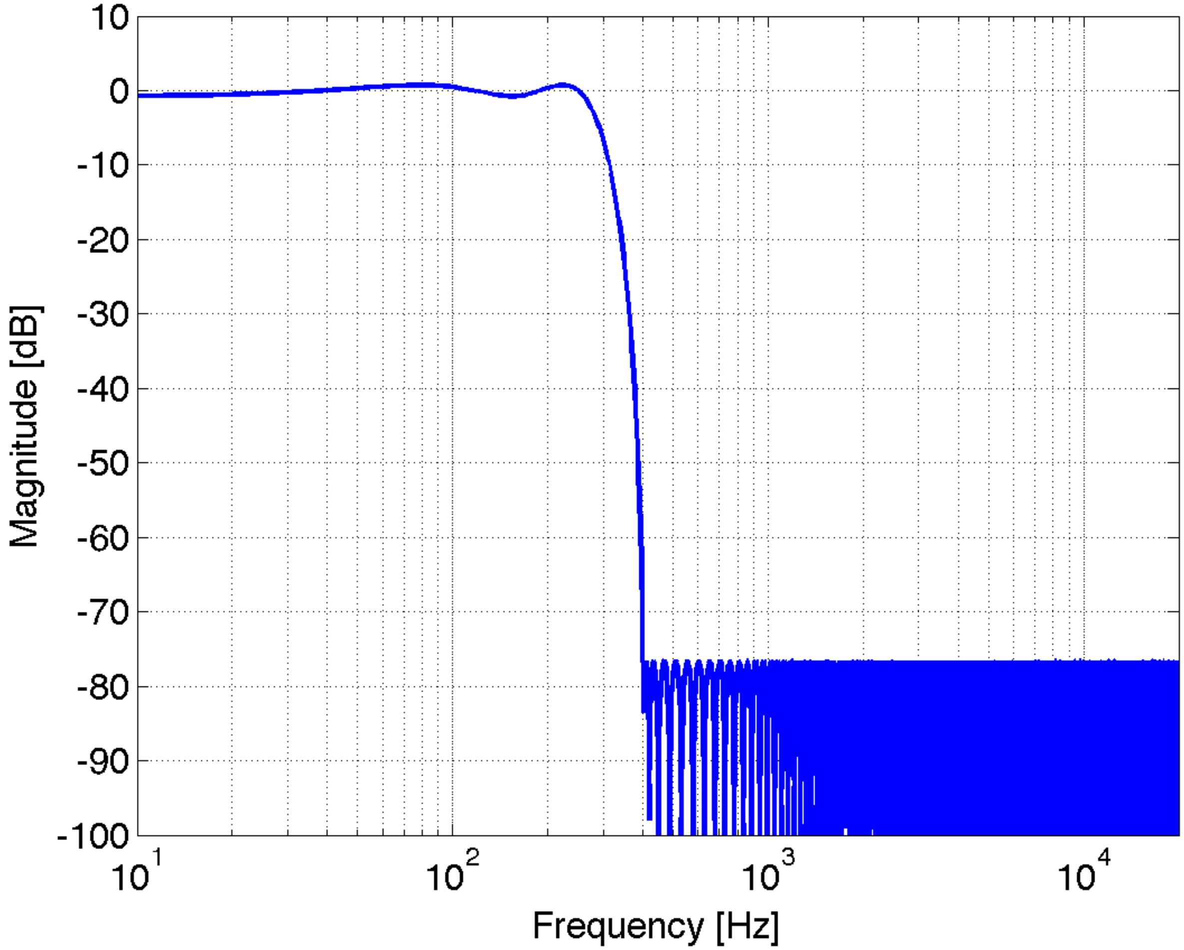
filter audio: pre-recorded mode

- use file `fireflyintro.wav`
- use the impulse response in `lpf_260_400_44100_80dB.bin`
- plot the magnitude frequency response of the filter in Matlab
- plot a spsectrogram of the signal before and after filtering
- listen to the before and after audio and describe the difference
- when doing convolution on pre-recorded signals, explain the advantages of zero padding the input signal
- how many zeros are padded at the start and end of the signal?

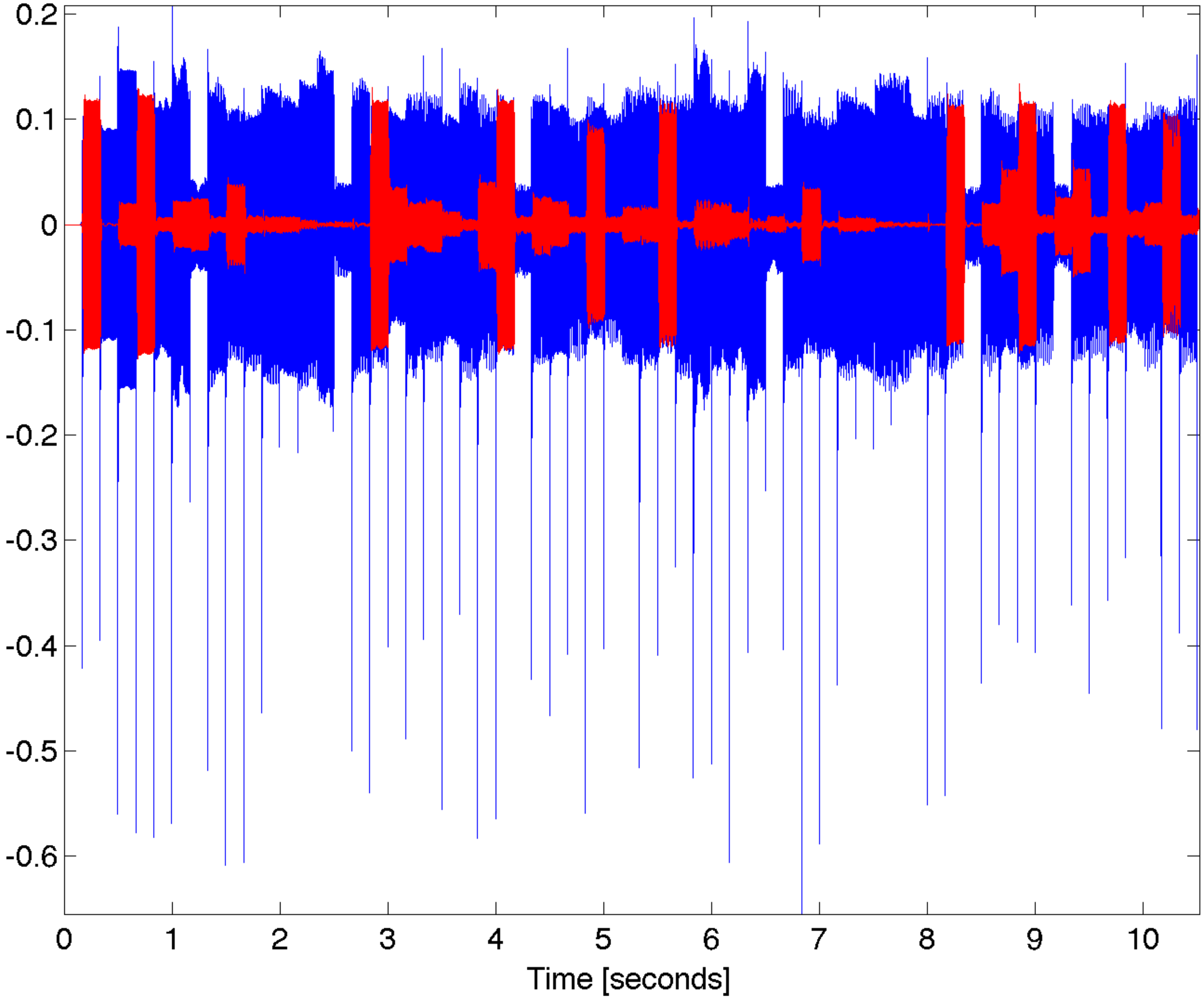
filter magnitude response in Matlab

```
fid = fopen('lpf_260_400_44100_80db.bin','rb');
head = fread(fid,5,'int'); % Read in header
h = fread(fid,inf,'float'); % Read in impulse response
fclose(fid);
stem(h); % Take a look to make sure we pulled in the right stuff

% Now make magnitude response plot
N = 2^14; % FFT size
f = [0:N-1]*44100/N; % Make a frequency vector for plotting
H = abs(fft(h,N)).^2; % Compute the magnitude response

figure(1);
plot(f,10*log10(H));
grid on;
xlim([0 44100/2]);
ylim([-100 10]);
xlabel('frequency [Hz]');

figure(2);
semilogx(f,10*log10(H));
grid on;
xlim([10 44100/2]);
ylim([-100 10]);
xlabel('log(frequency) [Hz]');
```

LPF Magnitude Response



audio before (blue) and after (red) filtering



filter audio: real-time mode

- use file `fireflyintro.wav`
- use the impulse response in `lpf_260_400_44100_80dB.bin`
- plot a spsectrogram of the signal before and after filtering
- listen to the before and after audio and describe the difference
- (this should give the same result as in the first part)
- when filtering real-time signals, explain the advantages of circular indexing

2D spatial filtering for edge detection

- write and test a C program to perform 2D convolution
- use the pre-recorded convolution method in which the entire image is loaded into memory and zero-padded all the way around
- modify the C program to convolve an input image x with the two point spread functions

$$h_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad h_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

- use `cameraman.png` as the input image
- combine the two resulting images into a single image using a root sum of squares combination

$$y(m,n) = \sqrt{[y_x(m,n)]^2 + [y_y(m,n)]^2},$$

where $y_x = h_x * x$ and $y_y = h_y * x$

- save the resulting image and view it in Matlab
- explain what you see
- find another digital picture (one of your own, or one found on the Internet) and apply the edge detection processing to it
- (note: either convert the picture to grayscale before applying edge detection, or apply edge detection to each of the color channels independently)
- prepare a document that includes the following
  - original and processed images
  - code
  - read Wikipedia's article on the "Sobel Operator" (`http://en.wikipedia.org/wiki/Sobel_operator`) and explain how edge detection works (please mention "convolution" in your discussion)
  - explain why zero padding the input image simplifies 2D convolution
  - how much zero padding is needed in the row and column dimension?

1D filtering in each pixel of a video using circular frame buffer