1 Convolution laboratory - I

1.1 Purpose

When you use Matlab's `conv` function, for example, `y=conv(x, h)`, it assumes that you have available the entire sequences for `x` and `h` and could process them using Matlab's vectorized operations. When you have to write a convolution routine in any 'real' programming language (e.g. C or Java or assembly), perhaps for an embedded processor application, then Matlab's vectorized operations aren't available to you.

Furthermore, in any real-time application, you don't have access to the entire input sequence – in fact the input sequence may be of infinite duration. Rather, you get the input one point at a time, perhaps as the result of an A/D conversion occurring at a constant rate. We will assume that for each point of the input, we require that the application produce one point of the output sequence $y[n]$.

To do a real-time convolution of an input with an impulse response $h[n]$, we need to write an efficient routine that does the minimum number of multiplications and additions and requires the minimum number of storage elements. Consider a convolution of $x[n]$ with $h[n]$ where $h[n]$ has N points, $0 \le n \le N-1$. For the n^{th} point of the output, we have to compute the convolution sum,

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]. \tag{L0.1}$$

So, we have to do the N multiplications and $N-1$ additions. Since in a real-time application we only get *one* new value of $x[n]$ at each time point, we have to keep the past $N-1$ values of $x[n]$ in memory (as well as the current value of $x[n]$). Hence, we need a buffer of N memory elements (plus whatever extra memory is necessary to accumulate the convolution sum).

1.2 Background

To prepare for this exercise, please read Section 9.9 of the book (that's right, Chapter 9).

1.2.1 Convolution

To do a real-time convolution, we need to write an efficient routine that requires the minimum number of multiplications and additions as well as the minimum number of storage elements.

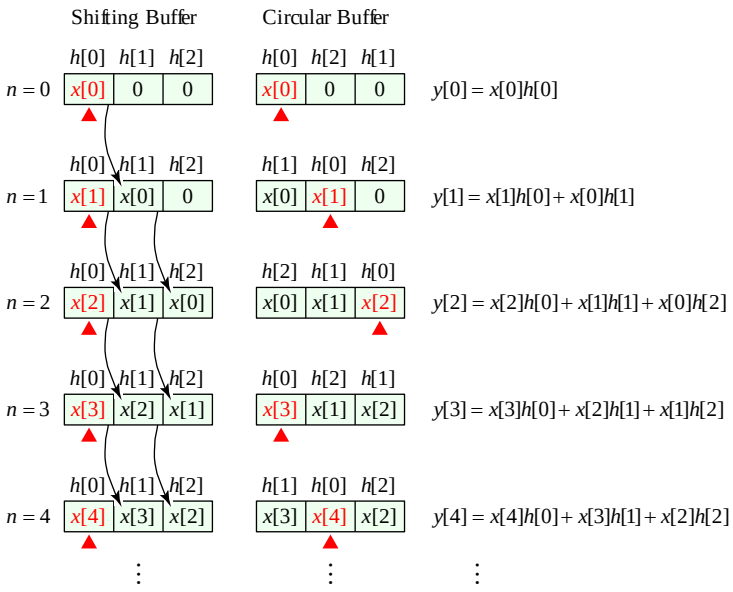

Figure 1: Shifting and circular buffers for real-time convolution

You have essentially two options for implementing this function, a **shifted buffer** approach or a **circular buffer** approach. Please read Section 9.10.1 of the book to understand these two methods. As a brief summary, look at Figure 1.

Assume that the impulse response comprises N elements. To perform the convolution, both the shifting buffer and circular buffer routines require an **input data buffer** of N memory elements (shown shaded in light green) of $x[n]$. In the shifting buffer approach, at each new time point n, we shift the past $N-1$ values, $x[n-k], 1 \le k < N$. of the input data buffer to the right by one element before placing the newest value of $x[n]$ in the leftmost position, as diagrammed in the left column of Figure 1 for a buffer of size, $N = 3$. Then, we have to do the N multiplications and $N-1$ additions of Equation (L0.1) to compute each point of the convolution sum $y[n]$. Notice that the values of $h[n]$ are addressed such that the newest data point is always multiplied by $h[0]$.

The problem with the shifting buffer is that moving elements from one memory location to another in this manner is relatively expensive from a processor perspective. The right panel of Figure 1 shows the less processor-intensive circular buffer approach. For the first N points of the convolution (e.g. $n = 0$, $n = 1$ and $n = 2$), input data points populate the buffer from left to right. For all subsequent points of the convolution, the newest value of $x[n]$ replaces the oldest data point; for example, at $n = 3$, $x[3]$ replaces $x[0]$. The remaining $N-1$ elements of the buffer remain unchanged. With the circular buffer, there are no memory moves of the input buffer; only one memory element gets written at each time point. Note that in this approach, the indices of the impulse response buffer have to be rotated so that the newest data point is always multiplied by $h[0]$. In a high-level language such as C, a circular buffer is easily implemented with pointers.

1.3 Your assignment

Your job is to write a convolution function, `convolv_rt`, in Matlab that basically implements a real-time convolution:

```
function y = convolv_rt(x, h)
% CONVOLV_RT  Convolve two finite-length arrays, x and h
%             returning array, y
```

In our function, we will use only regular Matlab input arrays $x[n]$ and $h[n]$, returning output array $y[n]$. I will guarantee that the length of $h[n]$ is always smaller than that of $x[n]$. We will also not time your convolution, so don't worry about writing the most efficient code possible.

Here is a template for a `conv_rt` program using the shifted or circular buffer:

```
function y = conv_rt(x, h)
    lh = length(h);
    hbuf = h(:)';                % make h a row vector
    x = [x(:); zeros(lh-1, 1)]; % pad x with zeros
    y = zeros(1, length(x));     % preallocate output array
    xbuf =  zeros(lh, 1);        % initialize input array as column vector
    for i = 1:length(x)          % for each new value of x[n]
        ; % put x(i) into the correct place in xbuf
        ; % set up the indexing into the hbuf and/or xbuf arrays here
        y(i) = hbuf * xbuf;      % store output value
    end
end
```

A few points to note:

- You don't always know if $h[n]$ comes into the function as a row vector or a column vector, Regardless of whether `h` is a row or column vector, `h(:)` turns it into a column vector and `h(:)'` makes it a row vector.
- Note that if `hbuf` is a row vector and `xbuf` is a column vector, then the product `hbuf * xbuf` is a value.

The next line of the program makes $x[n]$ into a column vector and pads the end with `lh-1` zeros. Why do we need to pad with these zeros?

`xbuf` is a column vector corresponding to our input data array. In the shifting buffer approach, you have to shift $N-1$ elements of `xbuf` and put `x(i)` into the correct place. You don't have to shift the elements of `hbuf`. For example, with $N=3$:

```
y(1) = [h(1) h(2) h(3)] * [x(1) 0 0]'
y(2) = [h(1) h(2) h(3)] * [x(2) x(1) 0]'
y(3) = [h(1) h(2) h(3)] * [x(3) x(2) x(1)]'
y(4) = [h(1) h(2) h(3)] * [x(4) x(3) x(2)]'
y(5) = [h(1) h(2) h(3)] * [x(5) x(4) x(3)]'
y(6) = [h(1) h(2) h(3)] * [x(6) x(5) x(4)]'
```

In the circular buffer approach, you have to replace the oldest value of the buffer with `x(i)` and index into *both* the `xbuf` and `hbuf` arrays correctly. For example, with $N=3$:

```
y(1) = [h(1) h(3) h(2)] * [x(1) 0 0]'
y(2) = [h(2) h(1) h(3)] * [x(1) x(2) 0]'
y(3) = [h(3) h(2) h(1)] * [x(1) x(2) x(3)]'
y(4) = [h(1) h(3) h(2)] * [x(4) x(2) x(3)]'
y(5) = [h(2) h(1) h(3)] * [x(4) x(5) x(3)]'
```

```
y(6) = [h(3) h(2) h(1)] * [x(4) x(5) x(6)]'
```

The point of this exercise is to emulate a real-time application in which input data are provided one point at a time (for example from an A/D converter), and you must provide one output point for every input point. So, in your function, you must only access each input point $x[n]$ once, one point at a time, and put it in the appropriate place in your `xbuf` array.

You can compare your conv_rt function with Matlab's conv function. They should get roughly the same result within a small margin of error. That is,
`all(abs(conv(x, h) - conv_rt(x, h))<1e-3)` should equal 1.

Download lab2_2024.zip from the website. Unpack and place the programs in your working directory. Remember to `publish('lab2_2024', 'pdf')` and submit your file to Canvas.