University of Washington – ECE Department
EE242 Lab 2 – Convolution
Background Material

In this lab, we will implement convolution on the computer for time signals and images. You'll learn different packages for easily implementing convolution in Python for both types of signals, and you will learn practical issues associated with working with digital signals. The examples will also show you examples of how convolution can be useful to remove noise and detect edges in signals.

1.0 Discrete-time Convolution on a Computer

The convolution operation describes how a linear time-invariant (LTI) system transforms an input signal to its resulting output signal. When we are working on the computer, everything is discrete and finite. When the input is a time series, the independent variable is an integer n, and both the input and impulse response are finite-length sequences (vectors), where length is always an integer. In this case convolution involves:

1. Flip the impulse response to get hflip
2. Iterate over each time point n in the input signal, i.e. slide hflip one step at a time
    a. Multiply hflip by the same length window in the input centered at that time point (a dot product)
    b. Sum the values in the resulting vector to get y[n]

To simplify the set-up, we'll assume that the vector that holds the time signal starts at time n=0, in which case a signal *x[n]* of length *N1* will have time samples *n = 0, ..., N1-1.* We'll also assume that all our signals are zero-valued in negative time and for times after the vector ends.

Let's say we want to convolve two finite-length signals, *x[n]* and *h[n],* which have length N1 and N2, respectively. Then we know that the resulting signal *y[n]=x[n]*h[n]* will be no greater than N1+N2-1. When you use a convolution function on the computer, the default is for the result to have this length. Of course, **x** and **h** may also include some zero-valued samples, in which case the resulting **y** vector will be longer than the non-zero part of the signal.

In our lab, we'll implement time convolution using the ***numpy.convolve()*** function. This function takes two signals as input, and produces their convolution as the output:

y = numpy.convolve(**x, h**)

For example, let's convolve two pulses. Let **x** be a length 3 pulse, and **h** be a length 5 pulse, and then find the convolution:

```
x=np.ones(3)
h=np.ones(5)
y=np.convolve(x,h)
print(y)
```

```
[1. 2. 3. 3. 3. 2. 1.]
```

If some the vectors have zero values, then the result will typically have zero values.

```
x = np.concatenate((np.ones(3),np.zeros(2)))
h = np.array([1,2,3,0,0])
y = np.convolve(x,h)
print(y)
```

```
[1. 3. 6. 5. 3. 0. 0. 0. 0.]
```

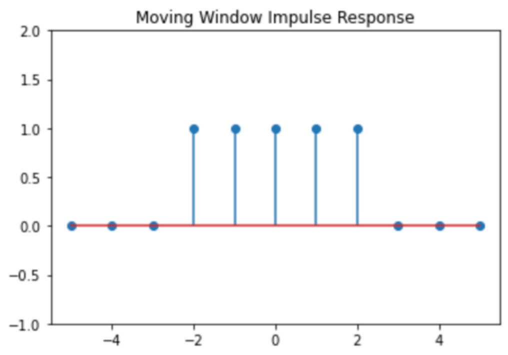
Aside: In applications where you are using computer signals that are truncated versions of longer signals, you may not trust the start/ends of the convolution result, in which case there are options to have the convolution routine return a smaller vector. You'll see this in one of the signals that you work with in the lab. For numpy.convolve(), this is specified in an optional third argument, which we won't worry about in this class.

2.0 Example Uses of Convolution

Convolution (i.e. LTI systems) can be used to solve many problems. In this lab, we'll explore just a few. Different impulse responses correspond to different systems. For example, the system

$$\boldsymbol{y[n] = \frac{1}{5}\sum_{k=-2}^{2} x[n+k]}$$

corresponds to a moving window averaging function that smooths a signal. The impulse response is



Systems like this are used to remove noise from a signal. Such systems are referred to as filters (e.g. they filter out noise), but there are other kinds of filters, which we'll learn about later in the course. You will experiment in lab with this type of smoother and a causal version of it. Note that this is just one possibility for smoothing a signal, and it may not be the best solution for all needs.

Another way convolution can be used is as a way to detect sudden changes. For example, the system

$$\boldsymbol{y[n] = x[n] - x[n-1]}$$

will be large when there is a big change between two samples. The impulse response for this system is $\boldsymbol{h[n] = \delta[n] - \delta[n-1]}$. Systems like this in 2 dimensions are useful for detecting edges in images.

3.0 2-D convolution for Images

Images can be grayscale or color. A grayscale image is a 2D array of pixels (picture elements), where each pixel is associated with a brightness value [0,255]. A true-color color image has three such arrays (R, G, B). We'll describe convolution in terms of operating on a single plane and work only with grayscale images. It is easy to extend this to multiple planes.

When the LTI system operates on 2-dimensional signals, then the independent variable is represented as a pair of integers (n, m), and the impulse response is 2-dimensional (a 2D array or matrix), often referred to as the convolution kernel. In this case, convolution involves:

1. Flip the impulse response (kernel) in both dimensions (up-down and left-right) to get hflip
2. Iterate over each spatial point (n,m) in the input signal, e.g. slide hflip one step at a time through a row and iterate over rows
    a. Multiply hflip by the same size array in the input centered at that time point
    b. Sum the values in the resulting matrix to get y[n,m]

In other words, you have two loops over n and m (vs. one loop over n for a time signal) and a double sum for finding the values.

In our lab, we'll implement convolution for images using the scipy ndimage package, which you will need to include in your import cell. For example:

```
from scipy import ndimage
result = ndimage.filters.convolve(image, kernel)
```

The equivalent to the time moving window for images would be a constant value in all elements of the matrix, e.g. a 3x3 kernel where all values are 1/9. In your lab, you will be implementing both a moving window smoother and edge detection. Edge detection will require multiple filters because we can get edges in different directions. We'll use two convolution kernels are required: one whose response is maximized for **horizontal edges** (h_1) and one whose response is maximized for **vertical edges** (h_2). The following are the convolution kernels of the **Sobel edge detector:**

$$h_1 = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad h_2 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

If y1=x*h1 gives the horizontal edges and y2=x*h2 gives the vertical edges, the overall result is given by computing for all n and m:

$$y(n,m) = (y_1(n,m)^2 + y_2(n,m)^2)^{1/2}$$

which can be implemented in one line with numpy.

$$y_{nm} = np.sqrt\big(np.power(y_1, 2) + np.power(y_2, 2)\big)$$