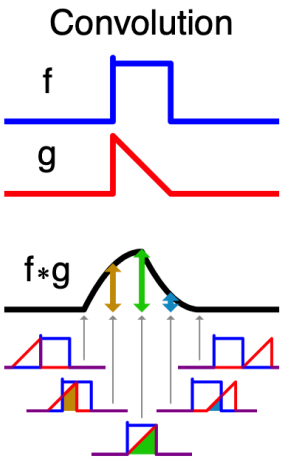
Communication System Lab
Assignment 1

Name- Pushkal Mishra
Roll- EE20BTECH11042

1) **Convolution Operator–**

Convolution is an operator which acts on two signals and produces a 3rd signal that expresses how the shape of one signal modifies the other when one of it is passed through a system with the other as the impulse response of the system. This operation is only applicable for Linear Time Invariant systems since the derivation of the operation uses both linearity and time invariance properties of a system.

Formally, say a given signal x[n] is passed through a LTI system with impulse response h[n], then the output of the system y[n] can be calculated as follows –



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

This is a visual example of convolution between two signals f and g in continuous time domain (same applies for discrete domain also).

Implementation in C language –

```c
double convolution(double *x, double *h, ll size_x, ll size_h, ll n)
{
    double answer = 0;
    if(n < 0 || n > (size_x + size_h - 2))
        answer = 0;
    else
    {
        ll start = 0;
        if(n >= size_h)
            start += (n - size_h + 1);
        ll k = start;
        while((k < size_x) && ((n - k) > -1))
        {
            answer += (x[k] * h[n - k]);
            k += 1;
        }
    }
    return answer;
}
```

In actuality, y[n] is a sequence of numbers so the above function calculates the convolution for a particular time instant n which is passed into the convolution function and we can calculate for all n in the testbench using a loop.

Some observations-

- Both x and h are of finite lengths with indexes starting with 0, so we consider the values at indexes less than 0 or more than their size as 0.
- We need to control the value of k such that it remains in the valid ranges for both x and h since if it does not then the product will be 0 and we can discard that value.
- So both the starting and ending value of k will depend on n and k must be greater than or equal to 0.

Sizes of both x and h are stored in size_x and size_h respectively.

Corner case-
If n is less than 0, the index for h (which is n – k) will always be negative for positive k and so there is no range of k for which the product will be non-zero. Same is the case when n is greater than (size_x + size_h – 2) since k should be greater than (size_x – 1) for the index of h (which is n – k) to be valid but that k will always violate the index of x (which is k) which should be less than (size_x – 1).
So in this case we return 0.

Now we can just determine the valid start and end values of k and run a loop to compute the convolution as given by the formula.

We know that k must be greater than or equal to 0 but if n exceeds (size_h - 1), k cannot start from 0 as the index will not be valid for h. So we can shift the starting point of k so that n – k comes into the range [0, size_h – 1] by setting the starting point as (n – (size_h – 1)) if n exceeds (size_h - 1).

Then we can run a loop which computes the convolution sum and increments k until either k exceeds (size_x – 1) or (n – k) < 0, then store the result in a variable and return it.

Corresponding test bench –

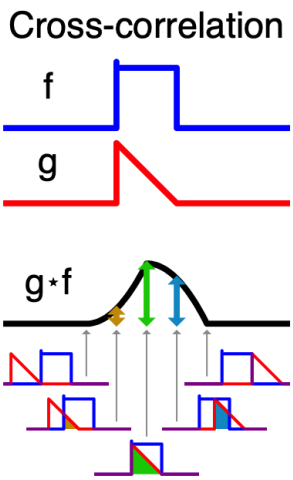
```
// Testing for the convolution operation
double convolution_x[20] = {0.5377, 1.8339, -2.2588, 0.8622, 0.3188, -1.3077, -0.4336, 0.3426, 3.5784, 2.7694, -1.3499, 3.0349, 0.7254, -0.0631, 0.7147, -0.2050,
    -0.1241, 1.4897, 1.4090, 1.4172};
ll size_x = sizeof(convolution_x) / sizeof(double);
double convolution_h[15] = {0.6715, -1.2075, 0.7172, 1.6302, 0.4889, 1.0347, 0.7269, -0.3034, 0.2939, -0.7873, 0.8884, -1.1471, -1.0689, -0.8095, -2.9443};
ll size_h = sizeof(convolution_h) / sizeof(double);
printf("The convolution of the two given sequences x and h is:\n");
// Printing y values for various time instants
for(ll n = -3; n < 37; n++)
{
    printf("y[%lld] = %lf\n", n, convolution(convolution_x, convolution_h, size_x, size_h, n));
}
```

2) Correlation Operator–

Correlation or Cross-correlation is an operator which measures the similarity between two signals when one signal is displaced in time with respect to the other. This is commonly referred to as a sliding dot product and used in pattern recognition in one signal when compared to another.

Specifically, say we have to measure the correlation between two signals x[n] and y[n] as when y[n] is shifted in time with respect to x[n], then the correlation can be calculated as follows –

$$R_{xy}[k] = \sum_{n=-\infty}^{\infty} x[n] \times y[n - k]$$



This is a visual example of correlation between two signals f and g in continuous time domain (same applies for discrete domain also). Here as you can see, g is shifted in time and area under the curve is calculated which represents the similarity between the two graphs for that time shift.

Implementation in C language –

```
double correlation(double *x, double *y, ll size_x, ll size_y, ll k)
{
    double answer = 0;
    if(k <= (-1 * size_y) || k >= size_x)
        answer = 0;
    else
    {
        ll start = 0;
        if(k > 0)
            start = k;
        ll n = start;
        while((n < size_x) && ((n - k) < size_y))
        {
            answer += (x[n] * y[n - k]);
            n += 1;
        }
    }
    return answer;
}
```

Note that just like in the case for convolution, here also we feed the value of k which is the amount of time y has to be shifted to calculate the correlation and then we can calculate for every k using a loop in testbench.

Also the analysis of the implementation will be on the same lines as for convolution since the expressions are similar.

Some observations-

- Both x and y are of finite lengths with indexes starting with 0, so we consider the values at indexes less than 0 or more than their size as 0.
- We need to control the value of n such that it remains in the valid ranges for both x and y since if it does not then the product will be 0 and we can discard that value.
- So both the starting and ending value of n will depend on k and n must be greater than or equal to 0.

Sizes of both x and y are stored in size_x and size_y respectively.

Corner case-
If k is less than or equal to (−size_y), the index for y (which is n − k) will always be more than size_y for positive n and so there is no range of n for which the product will be non-zero. Same is the case when k is greater than or equal to size_x since n should be greater than (size_x − 1) for the index of y (which is n − k) to be valid but that n will always violate the index of x (which is n) which should be less than (size_x − 1).
So in this case we return 0.

Similar to convolution, we can determine the valid start and end values for n.

We know that n must be greater than or equal to 0 but if k exceeds 0, n cannot start from 0 as the index will not be valid for y. So we can shift the starting point of n to k so that $(n - k) \geq 0$.

Then we can run a loop which computes the convolution sum and increments n until either n exceeds (size_x − 1) or (n − k) exceeds size_y, then store the result in a variable and return it.

Corresponding test bench –

```
// Testing for the correlation operation
double correlation_x[20] = {0.5377, 1.8339, -2.2588, 0.8622, 0.3188, -1.3077, -0.4336, 0.3426, 3.5784, 2.7694, -1.3499, 3.0349, 0.7254, -0.0631, 0.7147, -0.2050,
    -0.1241, 1.4897, 1.4090, 1.4172};
size_x = sizeof(correlation_x) / sizeof(double);
double correlation_y[15] = {0.6715, -1.2075, 0.7172, 1.6302, 0.4889, 1.0347, 0.7269, -0.3034, 0.2939, -0.7873, 0.8884, -1.1471, -1.0689, -0.8095, -2.9443};
ll size_y = sizeof(correlation_y) / sizeof(double);
printf("\nThe correlation of the two given sequences x and y is:\n");
// Printing the correlation values for various time instants
for(ll k = -17; k < 23; k++)
{
    printf("Rxy[%lld] = %lf\n", k, correlation(correlation_x, correlation_y, size_x, size_y, k));
}
```

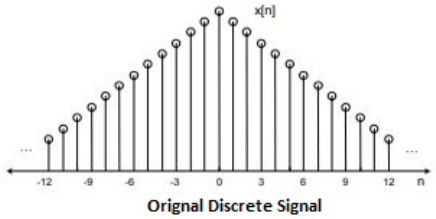
3) Downsampling Operator–

Downsampling is a process of re-sampling a signal with a lower frequency than it was initially sampled with, i.e. sample-rate reduction. So essentially a downsampled signal is an approximation of the sequence that would have been obtained by a lower sampling rate.

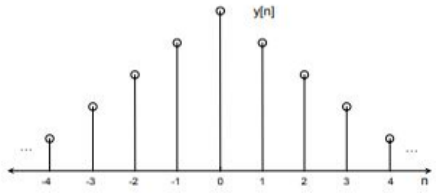
More accurately, the downsampled signal with a factor M from a given signal x[n] can be expressed as follows –

$$y[n] = x[Mn]$$

So basically we take every M^{th} value from x[n] and discard the rest. This is a visual example for downsampling/decimation with a factor of $M = 3$.



Orignal Discrete Signal



Decimation of a discrete-time signal by a factor of 3.

Implementation in C language –

```c
void downsample(double *x, double *y, ll size_x, ll M)
{
    for(ll i = 0; i < size_x; i += M)
        y[(i / M)] = x[i];
}
```

Here we also pass the pointer to a pre-declared array to store the values of the downsampled signal.

Corresponding test bench –

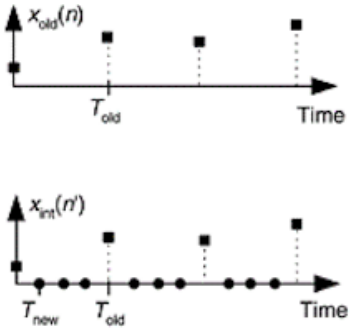
```c
// Testing the downsample operation
double downsample_x[36] = {0.3252, -0.7549, 1.3703, -1.7115, -0.1022, -0.2414, 0.3192, 0.3129, -0.8649,
    -0.0301, -0.1649, 0.6277, 1.0933, 1.1093, -0.8637, 0.0774, -1.2141, -1.1135, -0.0068, 1.5326, -0.7697,
    0.3714, -0.2256, 1.1174, -1.0891, 0.0326, 0.5525, 1.1006, 1.5442, 0.0859, -1.4916, -0.7423, -1.0616,
    2.3505, -0.6156, 0.7481};
size_x = sizeof(downsample_x) / sizeof(double);
ll M = 2;
size_y = (ll)(size_x / M);
double *downsample_y = (double *)calloc(size_y, sizeof(double));
downsample(downsample_x, downsample_y, size_x, M);
printf("\nThe downsampled version of the sequence x with M = %lld:\n", M);
// Printing the y values for the downsampled signal
for(ll i = 0; i < size_y; i++)
{
    printf("y[%lld] = %lf\n", i, downsample_y[i]);
}
free(downsample_y);

// Changing M
M = 3;
size_y = (ll)(size_x / M);
downsample_y = (double *)calloc(size_y, sizeof(double));
downsample(downsample_x, downsample_y, size_x, M);
printf("\nThe downsampled version of the sequence x with M = %lld:\n", M);
// Printing the y values for the downsampled signal
for(ll i = 0; i < size_y; i++)
{
    printf("y[%lld] = %lf\n", i, downsample_y[i]);
}
free(downsample_y);
```

Here I used calloc() to allocate a variable memory space to the array and freed it after testing it.

4) Upsampling Operator–

Upsampling is a process of re-sampling a signal with a higher sampling rate than it was initially sampled with. So essentially an upsampled signal is an approximation of the sequence that would have been obtained by a higher sampling rate. In the end, the upsampled signal will have L – 1 zeros inserted in between the samples of the input signal (L is the upsampling factor).

More precisely, the upsampled signal with a factor L from a given signal x[n] can be expressed as follows –



$$y[n] = \begin{cases} x[n/L], & \textit{if n is a multiple of L} \\ 0, & \textit{otherwise} \end{cases}$$

So basically, we insert (L – 1) zeros in between the samples of x[n] and re-index the signal. This is a visual example for upsampling/interpolation with a factor of L = 4.

Implementation in C language –

```c
void upsample(double *x, double *y, ll size_x, ll L)
{
    ll size_y = size_x * L;
    for(ll i = 0; i < size_y; i++)
    {
        if(i % L == 0)
            y[i] = x[i / L];
        else
            y[i] = 0;
    }
}
```

Similar to upsampling, we also pass the pointer to a pre-declared array using calloc().

Corresponding test bench –

```c
// Testing the upsampling operation
double upsample_x[18] = {0.3252, 1.3703, -0.1022, 0.3192, -0.8649, -0.1649, 1.0933,
    -0.8637, -1.2141, -0.0068, -0.7697, -0.2256, -1.0891, 0.5525, 1.5442, -1.4916,
    -1.0616, -0.6156};
size_x = sizeof(upsample_x) / sizeof(double);
ll L = 2;
size_y = size_x * L;
double *upsample_y = (double *)calloc(size_y, sizeof(double));
upsample(upsample_x, upsample_y, size_x, L);
printf("\nThe upsampled version of the sequence x with L = %lld:\n", L);
// Printing the y values for the downsampled signal
for(ll i = 0; i < size_y; i++)
{
    printf("y[%lld] = %lf\n", i, upsample_y[i]);
}
free(upsample_y);

// Changing L value
L = 3;
size_y = size_x * L;
upsample_y = (double *)calloc(size_y, sizeof(double));
upsample(upsample_x, upsample_y, size_x, L);
printf("\nThe upsampled version of the sequence x with L = %lld:\n", L);
// Printing the y values for the downsampled signal
for(ll i = 0; i < size_y; i++)
{
    printf("y[%lld] = %lf\n", i, upsample_y[i]);
}
free(upsample_y);
```

Output of testbench–

```
The convolution of the two given sequences x and h is:
y[-3] = 0.000000
y[-2] = 0.000000
y[-1] = 0.000000
y[0] = 0.361066
y[1] = 0.582191
y[2] = -3.345580
y[3] = 5.498300
y[4] = 0.805462
y[5] = -2.874046
y[6] = 4.106150
y[7] = -0.410274
y[8] = -1.445902
y[9] = -1.804175
y[10] = -4.350491
y[11] = 13.203416
y[12] = -2.715715
y[13] = 4.821051
y[14] = 10.269640
y[15] = -4.427006
y[16] = 9.755986
y[17] = 2.903170
y[18] = -0.592848
y[19] = 4.635050
y[20] = -7.384201
y[21] = 0.336801
y[22] = -9.399403
y[23] = -8.545026
y[24] = 3.613228
y[25] = -9.319175
y[26] = -3.912468
y[27] = 0.599374
y[28] = -3.378534
y[29] = -1.245523
y[30] = -3.972275
y[31] = -7.041554
y[32] = -5.295742
y[33] = -4.172662
y[34] = 0.000000
y[35] = 0.000000
y[36] = 0.000000
```

```
The correlation of the two given sequences x and y is:
Rxy[-17] = 0.000000
Rxy[-16] = 0.000000
Rxy[-15] = 0.000000
Rxy[-14] = -1.583150
Rxy[-13] = -5.834820
Rxy[-12] = 4.591295
Rxy[-11] = -3.287128
Rxy[-10] = -0.848136
Rxy[-9] = 6.467562
Rxy[-8] = -2.287081
Rxy[-7] = 3.294564
Rxy[-6] = -10.074681
Rxy[-5] = -9.504093
Rxy[-4] = -1.490325
Rxy[-3] = -14.680519
Rxy[-2] = 0.247014
Rxy[-1] = -5.437126
Rxy[0] = -12.322000
Rxy[1] = 7.005693
Rxy[2] = -4.859548
Rxy[3] = 0.220402
Rxy[4] = 0.664786
Rxy[5] = 0.141289
Rxy[6] = 5.301993
Rxy[7] = -4.103834
Rxy[8] = 2.326033
Rxy[9] = 8.033679
Rxy[10] = -5.260245
Rxy[11] = 3.125120
Rxy[12] = 2.816635
Rxy[13] = 1.961664
Rxy[14] = 5.222200
Rxy[15] = 4.070427
Rxy[16] = 1.438708
Rxy[17] = 0.315382
Rxy[18] = -0.765126
Rxy[19] = 0.951650
Rxy[20] = 0.000000
Rxy[21] = 0.000000
Rxy[22] = 0.000000
```

```
The downsampled version of the sequence x with M = 2:
y[0] = 0.325200
y[1] = 1.370300
y[2] = -0.102200
y[3] = 0.319200
y[4] = -0.864900
y[5] = -0.164900
y[6] = 1.093300
y[7] = -0.863700
y[8] = -1.214100
y[9] = -0.006800
y[10] = -0.769700
y[11] = -0.225600
y[12] = -1.089100
y[13] = 0.552500
y[14] = 1.544200
y[15] = -1.491600
y[16] = -1.061600
y[17] = -0.615600

The downsampled version of the sequence x with M = 3:
y[0] = 0.325200
y[1] = -1.711500
y[2] = 0.319200
y[3] = -0.030100
y[4] = 1.093300
y[5] = 0.077400
y[6] = -0.006800
y[7] = 0.371400
y[8] = -1.089100
y[9] = 1.100600
y[10] = -1.491600
y[11] = 2.350500
```

```
The upsampled version of the sequence x with L = 2:
y[0] = 0.325200
y[1] = 0.000000
y[2] = 1.370300
y[3] = 0.000000
y[4] = -0.102200
y[5] = 0.000000
y[6] = 0.319200
y[7] = 0.000000
y[8] = -0.864900
y[9] = 0.000000
y[10] = -0.164900
y[11] = 0.000000
y[12] = 1.093300
y[13] = 0.000000
y[14] = -0.863700
y[15] = 0.000000
y[16] = -1.214100
y[17] = 0.000000
y[18] = -0.006800
y[19] = 0.000000
y[20] = -0.769700
y[21] = 0.000000
y[22] = -0.225600
y[23] = 0.000000
y[24] = -1.089100
y[25] = 0.000000
y[26] = 0.552500
y[27] = 0.000000
y[28] = 1.544200
y[29] = 0.000000
y[30] = -1.491600
y[31] = 0.000000
y[32] = -1.061600
y[33] = 0.000000
y[34] = -0.615600
y[35] = 0.000000
```

```
The upsampled version of the sequence x with L = 3:
y[0] = 0.325200
y[1] = 0.000000
y[2] = 0.000000
y[3] = 1.370300
y[4] = 0.000000
y[5] = 0.000000
y[6] = -0.102200
y[7] = 0.000000
y[8] = 0.000000
y[9] = 0.319200
y[10] = 0.000000
y[11] = 0.000000
y[12] = -0.864900
y[13] = 0.000000
y[14] = 0.000000
y[15] = -0.164900
y[16] = 0.000000
y[17] = 0.000000
y[18] = 1.093300
y[19] = 0.000000
y[20] = 0.000000
y[21] = -0.863700
y[22] = 0.000000
y[23] = 0.000000
y[24] = -1.214100
y[25] = 0.000000
y[26] = 0.000000
y[27] = -0.006800
y[28] = 0.000000
y[29] = 0.000000
y[30] = -0.769700
y[31] = 0.000000
y[32] = 0.000000
y[33] = -0.225600
y[34] = 0.000000
y[35] = 0.000000
y[36] = -1.089100
y[37] = 0.000000
y[38] = 0.000000
y[39] = 0.552500
y[40] = 0.000000
y[41] = 0.000000
y[42] = 1.544200
y[43] = 0.000000
y[44] = 0.000000
y[45] = -1.491600
y[46] = 0.000000
y[47] = 0.000000
y[48] = -1.061600
y[49] = 0.000000
y[50] = 0.000000
y[51] = -0.615600
y[52] = 0.000000
y[53] = 0.000000
Program ended with exit code: 0
```