**LAB MANUAL**

**Subject: Digital Signal Processing**

**Subject Code: CPC701**

**Class: B.E**

**Semester: VII(SH2017)**

Mrs.Bhagyashri patil Dr.Mahavir Devmane

**Subject In Charge (HOD Computer)**

**EXPERIMENT LIST**

**ACADEMIC YEAR-2017-2018**

**Class : B.E SEM:VII**

**Subject : DIGITAL SIGNAL PROCESSING (CPC701)**

**EXPERIMENT LIST**

|  |  |
| --- | --- |
| **Sr. No.** | **Title** |
| 1. | Study of analog signal waveform generation for the following: (C/C++ & Scilab)  a. Sine waveform  b. Cosine waveform  c. exponential function |
| 2. | To study sampling and reconstruction of signal. (C/C++ & Scilab) |
| 3. | To study mathematical operation Correlation and measure degree of similarity between two signals. (C/C++ & Scilab) |
| 4. | To perform discrete Convolution. (C/C++ & Scilab) To study mathematical operation such as Linear convolution, Circular  convolution, Linear convolution using circular convolution. |
| 5. | To perform Discrete Fourier Transform. (C/C++ & Scilab) To study magnitude spectrum of the DT signal. |
| 6. | To perform Fast Fourier Transform. (C/C++ & Scilab) To implement computationally fast algorithms. |
| 7. | To perform filtering of Long Data Sequence using Overlap Add Method and Overlap Save Method.(C/C++ & Scilab) |
| 8. | To perform real time signal processing using TMS320 Processor. |
| 9. | To implement any signal processing operation on one dimensional signal in scilab.(Mini-Project) |
| 10. | Study of Latex Tool. |

Mrs.Bhagyashri patil

**Subject In Charge (HOD Computer)**

**EXPERIMENT NO. 1**

**Aim:** Study of analog signal waveform generation for the following: (C/C++ & Scilab)

a. Sine waveform

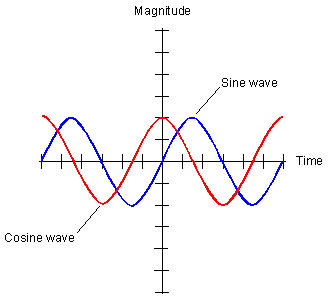
b. Cosine waveform

c. exponential function

**Theory:**

An analog or analogue signal is any continuous signal for which the time varying feature (variable) of the signal is a representation of some other time varying quantity, i.e., analogous to another time varying signal. For example, in an analog audio signal, the instantaneous voltage of the signal varies continuously with the pressure of the sound waves. It differs from a digital signal, in which the continuous quantity is a representation of a sequence of discrete values which can only take on one of a finite number of values. The term analog signal usually refers to electrical signals; however, mechanical, pneumatic, hydraulic, human speech, and other systems may also convey or be considered analog signals. An analog signal uses some property of the medium to convey the signal's information. For example, an aneroid barometer uses rotary position as the signal to convey pressure information. In an electrical signal, the voltage, current, or frequency of the signal may be varied to represent the information. Any information may be conveyed by an analog signal; often such a signal is a measured response to changes in physical phenomena, such as sound, light, temperature, position, or pressure. The physical variable is converted to an analog signal by a transducer. For example, in sound recording, fluctuations in air pressure (that is to say, sound) strike the diaphragm of a microphone which induces corresponding fluctuations in the current produced by a coil in an electromagnetic microphone, or the voltage produced by a condensor microphone. The voltage or the current is said to be an "analog" of the sound. An analog signal has a theoretically infinite resolution. In practice an analog signal is subject to electronic noise and distortion introduced by communication channels and signal processing operations, which can progressively degrade the signal-to-noise ratio (SNR). In contrast, digital signals have a finite resolution. Converting an analog signal to digital form introduces a constant low-level noise called quantization noise into the signal which determines the noise floor, but once in digital form the signal can in general be processed or transmitted without introducing additional noise or distortion. In analog systems, it is difficult to detect when such degradation occurs. However, in digital systems, degradation can not only be detected but corrected as well.

Signal generators, also known variously as function generators, RF and microwave signal generators, pitch generators, arbitrary waveform generators, digital pattern generators or frequency generators, are electronic device that generate repeating or non-repeating electronic signals (in either the analog or digital domains). They are generally used in designing, testing, troubleshooting, and repeating electroacoustic devices; though they often have artistic uses as well.



**Sine wave:**

A sine wave or sinusoid is a mathematical curve that describes a smooth repetitive oscillation. A sine wave is a continuous wave. It is named after the function sine, of which it is the graph. It occurs often in pure and applied mathematics, as well as physics, engineering, signal processing and many other fields. Its most basic form as a function of time (t) is:

where:

• A = the amplitude, the peak deviation of the function from zero.

• f = the ordinary frequency, the number of oscillations (cycles) that occur each second of time.

• ω = 2πf, the angular frequency, the rate of change of the function argument in units of radians per second

• = the phase, specifies (in radians) where in its cycle the oscillation is at t = 0.

• When is non-zero, the entire waveform appears to be shifted in time by the amount /ω seconds. A negative value represents a delay, and a positive value represents an advance.

The sine wave is important in physics because it retains its wave shape when added to another sine wave of the same frequency and arbitrary phase and magnitude. It is the only periodic waveform that has this property. This property leads to its importance in Fourier analysis and makes it acoustically unique.

**Cosine wave:**

A cosine wave is a signal waveform with a shape identical to that of a sine wave , except each point on the cosine wave occurs exactly 1/4 cycle earlier than the corresponding point on the sine wave. A cosine wave and its corresponding sine wave have the same frequency, but the cosine wave leads the sine wave by 90 degrees of phase.

**Exponential waveform:**

Exponential functions are uniquely [characterized](https://en.wikipedia.org/wiki/Characterization_(mathematics)) by the fact that the growth rate of such a function is directly proportional to the value of the function. This proportionality can be expressed by saying

where ln b is a constant, and a constant is a quantity that does not change as the variable *x* changes.

For just one base *b* this constant factor is equal to 1, and that base is [the number *e* ≈ 2.71828...](https://en.wikipedia.org/wiki/E_(mathematical_constant)):{\displaystyle {\frac {d}{dx}}e^{x}=e^{x}\times 1}

The exponential function models a relationship in which a constant change in the independent variable gives the same proportional change (i.e. percentage increase or decrease) in the dependent variable. The function is often written as exp(*x*), especially when it is impractical to write the independent variable as a [superscript](https://en.wikipedia.org/wiki/Superscript). The exponential function is widely used in [physics](https://en.wikipedia.org/wiki/Physics), [chemistry](https://en.wikipedia.org/wiki/Chemistry), [engineering](https://en.wikipedia.org/wiki/Engineering), [mathematical biology](https://en.wikipedia.org/wiki/Mathematical_biology), [economics](https://en.wikipedia.org/wiki/Economics) and mathematics.

|  |  |
| --- | --- |
| **Exponential function** | |
| **Representation** | *ex* |
| **Inverse** | ln *x* |
| **Derivative** | *ex* |
| **Indefinite Integral** | *ex* + *C* |

The [graph](https://en.wikipedia.org/wiki/Graph_of_a_function) of *y* = *ex* is upward-sloping, and increases faster as *x* increases. The graph always lies above the *x*-axis but can get arbitrarily close to it for negative *x*; thus, the *x*-axis is a horizontal [asymptote](https://en.wikipedia.org/wiki/Asymptote). The [slope](https://en.wikipedia.org/wiki/Slope) of the [tangent](https://en.wikipedia.org/wiki/Tangent) to the graph at each point is equal to its *y*-coordinate at that point. The [inverse function](https://en.wikipedia.org/wiki/Inverse_function) is the [natural logarithm](https://en.wikipedia.org/wiki/Natural_logarithm) ln(*x*); because of this, some old texts[[3]](https://en.wikipedia.org/wiki/Exponential_function#cite_note-3) refer to the exponential function as the **antilogarithm**.

In general, the [variable](https://en.wikipedia.org/wiki/Variable_(mathematics)) *x* can be any real or [complex number](https://en.wikipedia.org/wiki/Complex_number) or even an entirely different kind of [mathematical object](https://en.wikipedia.org/wiki/Mathematical_object); see the [formal definition below](https://en.wikipedia.org/wiki/Exponential_function#Formal_definition).

A nonrepetitive waveform that rises or falls exponentially from some initial value at some initial time, according to the law y(t) = eat For a>0 the waveform rises without bound with increasing time t; for a<0 the waveform decays to zero. One way in which logic signals can become distorted as they travel through a system is for their switching edges to become exponentials, usually due to capacitive loading when the output waveform follows the lawv(t) = V(1–exp(–t/CR) where V is the final voltage, R the source resistance, and C the capacitive load.

**Expected Input:**

**Steps:**

1.Take time period 1 to 20 seconds.

2.Take fundamental frequency F.

3. Take value for amplitude A.

4. For time t=0 to 20 seconds

apply formula for sine & cosine wave

a. sine wave: sin2\*pi\*F\*t(0 to 20)

b. cosine wave: sin2\*pi\*F\*t(0 to 20)

5. print value for sine & cosine waves.

6. To display wavefrom use scilab

a. Take output in Excel sheet

b. Read this file in Scilab

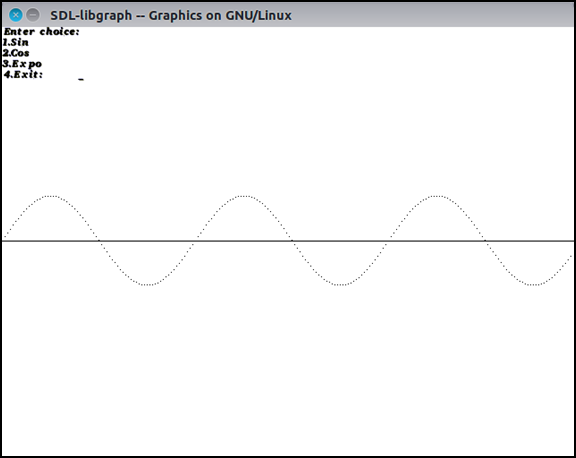
c. use subplot function to plot wavefrom.

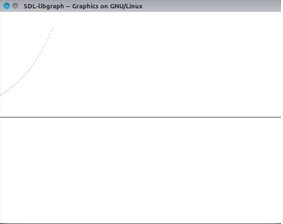
**Expected Output: .**

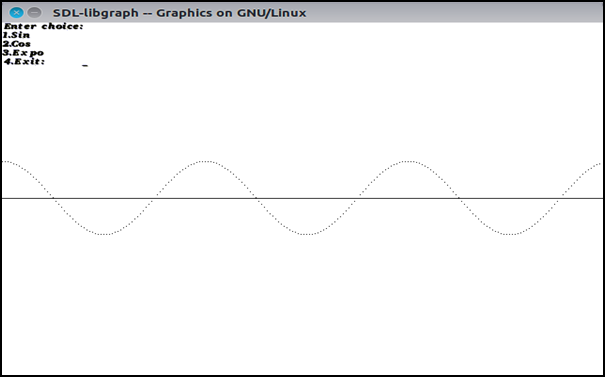
a.sine wave

b.cosine wave

c.exponential wave sholud be displayed







**Conclusion:**

Thus, Analog Signal Processing has performed and the Sine, cosine, exponential wave result is displayed.

**Program: -**

#include<math.h>

#include<graphics.h>

void main() {

int gd = DETECT, gm;

int angle = 0,ch;

double x, y;

printf("Enter choice:\n1.Sin\n2.Cos\n3.Expo\n4.Exit:\t");

scanf("%d",&ch);

initgraph(&gd, &gm, "C:\\TC\\BGI");

setbkcolor(BLACK);

line(0, getmaxy() / 2, getmaxx(), getmaxy() / 2);

switch(ch) {

case 1: /\* generate a sine wave \*/

for(x = 0; x < getmaxx(); x+=3) {

/\* calculate y value given x \*/

y = 50\*sin(angle\*3.141/180);

y = getmaxy()/2 - y;

/\* color a pixel at the given position \*/

putpixel(x, y, 15);

delay(100);

/\* increment angle \*/

angle+=5;

}

main();

break;

case 2: /\* generate a cosine wave \*/

for(x = 0; x < getmaxx(); x+=3) {

/\* calculate y value given x \*/

y = 50\*cos(angle\*3.141/180);

y = getmaxy()/2 - y;

/\* color a pixel at the given position \*/

putpixel(x, y, 15);

delay(100);

/\* increment angle \*/

angle+=5;

}

main();

break;

case 3: /\* generate a exp^n wave \*/

for(x = 0; x < getmaxx(); x+=3) {

/\* calculate y value given x \*/

y = 50\*exp(angle\*3.141/180);

y = getmaxy()/2 - y;

/\* color a pixel at the given position \*/

putpixel(x, y, 15);

delay(100);

/\* increment angle \*/

angle+=2;

}

main();

break;

case 4:exit(0);

break;

default:

printf("Invalid choice!");

}

getch();

/\* deallocate memory allocated for graphics screen \*/

closegraph();

}

**EXPERIMENT NO. 2**

**2. Sampling and Reconstruction**

**Aime:** To study sampling and reconstruction of signal

**Objective:**

Develop a program to sample a continuous time signal and convert it to Discrete Time Signal.

**Problem Definition:**

1. Sample the input signal and display first 50 samples. Calculate data rate and bit rate.

2. Reconstruct the original signal and display the original and reconstructed signals.

3.Vary the sampling frequency and observe the change in the quality of reconstructed

signal.

**Theory:**

In the field of [digital signal processing](https://en.wikipedia.org/wiki/Digital_signal_processing), the **sampling theorem** is a fundamental bridge between [continuous-time signals](https://en.wikipedia.org/wiki/Continuous-time_signal) (often called "analog signals") and [discrete-time signals](https://en.wikipedia.org/wiki/Discrete-time_signal) (often called "digital signals"). It establishes a sufficient condition for a [sample rate](https://en.wikipedia.org/wiki/Sample_rate) that permits a discrete sequence of *samples* to capture all the information from a continuous-time signal of finite bandwidth.

Strictly speaking, the theorem only applies to a class of [mathematical functions](https://en.wikipedia.org/wiki/Mathematical_function) having a [Fourier transform](https://en.wikipedia.org/wiki/Continuous_Fourier_transform) that is zero outside of a finite region of frequencies. Intuitively we expect that when one reduces a continuous function to a discrete sequence and [interpolates](https://en.wikipedia.org/wiki/Interpolates) back to a continuous function, the fidelity of the result depends on the density (or [sample rate](https://en.wikipedia.org/wiki/Sampling_(signal_processing))) of the original samples. The sampling theorem introduces the concept of a sample rate that is sufficient for perfect fidelity for the class of functions that are [bandlimited](https://en.wikipedia.org/wiki/Bandlimited) to a given bandwidth, such that no actual information is lost in the sampling process. It expresses the sufficient sample rate in terms of the bandwidth for the class of functions. The theorem also leads to a formula for perfectly reconstructing the original continuous-time function from the samples.

Perfect reconstruction may still be possible when the sample-rate criterion is not satisfied, provided other constraints on the signal are known. (See [§ Sampling of non-baseband signals](https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem#Sampling_of_non-baseband_signals) below, and [compressed sensing](https://en.wikipedia.org/wiki/Compressed_sensing).) In some cases (when the sample-rate criterion is not satisfied), utilizing additional constraints allows for approximate reconstructions. The fidelity of these reconstructions can be verified and quantified utilizing Bochner's theorem.

General principle

Let *F* be any sampling method, i.e. a linear map from the [Hilbert space](https://en.wikipedia.org/wiki/Hilbert_space) of square-integrable functions  to [complex](https://en.wikipedia.org/wiki/Complex_number) space {\mathbb  C}^{n}.

In our example, the vector space of sampled signals  is *n*-dimensional complex space. Any proposed inverse *R* of *F* (*reconstruction formula*, in the lingo) would have to map {\mathbb  C}^{n} to some subset of . We could choose this subset arbitrarily, but if we're going to want a reconstruction formula *R* that is also a linear map, then we have to choose an *n*-dimensional linear subspace of .

This fact that the dimensions have to agree is related to the [Nyquist–Shannon sampling theorem](https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem).

The elementary linear algebra approach works here. Let  (all entries zero, except for the *k*th entry, which is a one) or some other basis of . To define an inverse for *F*, simply choose, for each *k*, an  so that . This uniquely defines the (pseudo-)inverse of *F*.

Of course, one can choose some reconstruction formula first, then either compute some sampling algorithm from the reconstruction formula, or analyze the behavior of a given sampling algorithm with respect to the given formula.

Ideally, the reconstruction formula is derived by minimizing the expected error variance. This requires that either the signal statistics is known or a prior probability for the signal can be specified. [Information field theory](https://en.wikipedia.org/wiki/Information_field_theory) is then an appropriate mathematical formalism to derive an optimal reconstruction formula.

Popular reconstruction formulae

Perhaps the most widely used reconstruction formula is as follows. Let  be a basis of  in the Hilbert space sense; for instance, one could use the eikonal,although other choices are certainly possible. Note that here the index *k* can be any integer, even negative.Then we can define a linear map *R* by for each , where  is the basis of  given by (This is the usual discrete Fourier basis.) The choice of range  is somewhat arbitrary, although it satisfies the dimensionality equirement and reflects the usual notion that the most important information is contained in the low frequencies. In some cases, this is incorrect, so a different reconstruction formula needs to be chosen.

A similar approach can be obtained by using [wavelets](https://en.wikipedia.org/wiki/Wavelet) instead of Hilbert bases. For many applications, the best approach is still not clear today.

**Expected Input:**

**Steps:**

1. Take two fundamental frquencies F1=0.1 and F2=0.2.

2. Original signal values are calculated by using formulae:

sin2\*pi\*F1\*t(0 to 20.5 sec)+sin2\*pi\*F2\*t(0 to 20.5 sec) for the difference of 0.1 sec.

3. Output i.e total 201 values will be written into orig.txt file.

4. Sampling

a.Take 0 to 201 values.

b. for m=1 to 201 consider sampling period of N=10.

c. write output (sampled value=20) into sample.txt file.

5. Reconstruction

a. Replacing each sample by a sinc function, centered at time of the sample and scaled by the sample value x(nT) times 2fc/fs

b. adding all the sinc fuctions so created.

c. write output of reconstructed signal into recon.txt.

6. To display original, sampled and reconstructed waveform : use scilab

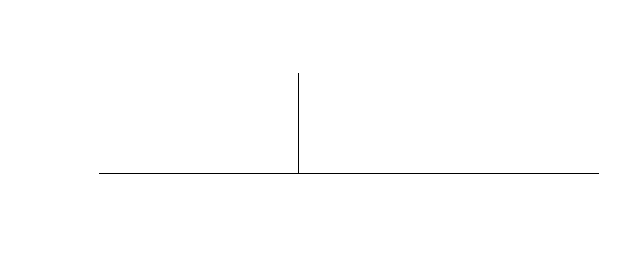
a. Copy contents of all three txt files(orig.txt, sample.txt, recon.txt) in excel sheets.

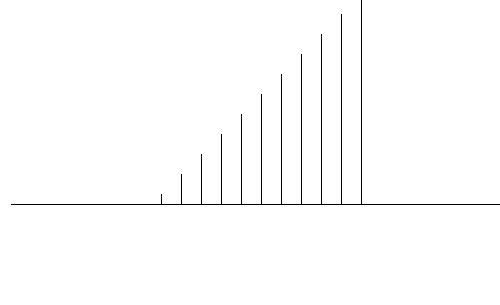
b. Read all excel files.

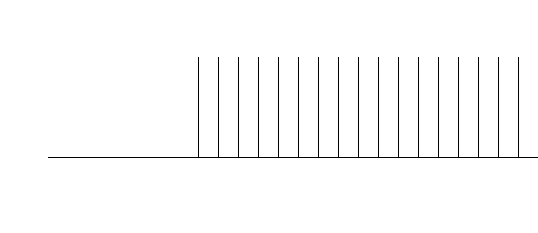
c. While displaying reconstructed signal, use first 200 values then next 200 upto 2001 values which we are getting in recon.txt file.

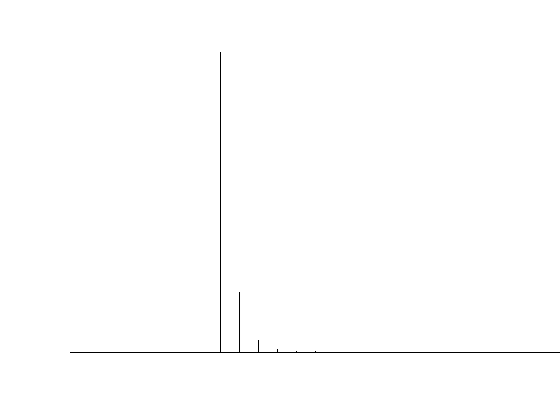
**Expected Output:**

To display original, sampled and reconstructed waveform i.e to sample a continuous time signal and convert it to Discrete Time Signal.









**Conclusion:**

Thus, we developed a program to sample a continuous time signal and convert it to Discrete Time Signal & we displayed the result.

**Program: -**

#include<stdio.h>

#include<math.h>

#include<graphics.h>

void main()

{

int gd=DETECT,gm,option;

int t,A,n,a;

float pi=3.14,X=0,y=0,x[600],f=10,r;

printf("Enter value of amplitude");

scanf("%d",&A);

printf("enter option");

scanf("%d",&option);

initgraph(&gd,&gm,NULL);

switch(option)

{

case 1:

for(t=0;t<=500;t++)

{

if(t==0)

{

x[t]=1;

for(n=0;n<=100;n++)

{y=(200+n)-x[t]\*100;

putpixel((X+300),y,WHITE);}

X++;

}

else

{

x[t]=0;

for(n=0;n<=100;n++)

{y=(200)-x[t]\*100;

putpixel((X+100),y,WHITE);}

X++;

}

}

break;

case 2:

line(150, 200, 800, 200);

for(t=0;t<=500;t++)

{

if(t>=0)

{

x[t]=1;

for(n=0;n<=100;n++)

{y=(200+n)-x[t]\*100;

putpixel((X+300),y,WHITE);}

X=X+20;

} else {

x[t]=0;

for(n=0;n<=100;n++)

{y=(200)-x[t]\*100;

putpixel((X+100),y,WHITE);}

X++;

}

}

break;

case 3:

line(150, 210, 800, 210);

for(t=0;t<=200;t++)

{

if(t>=0)

{

x[t]=t;

a=10+t;

for(n=0;n<=a;n++)

{y=(200+n)-x[t]\*1;

putpixel((t+300),y,WHITE);}

t=t+19;

} else {

x[t]=0;

for(n=0;n<=100;n++)

{y=(200)-x[t]\*100;

putpixel((X+100),y,WHITE);}

X++;

}

}

break;

case 4:

line(150, 400, 800, 400);

r=0.2;

for(t=0;t<=5;t++)

{

if(t>=0)

{

x[t]=pow(r,t);

//printf("x[%d] = %f", t , x[t]);

a=300\*x[t];

for(n=0;n<=a;n++)

{y=(400+n)-x[t]\*300;

putpixel((X+300),y,WHITE);}

X=X+19;

}

else

{

x[t]=0;

for(n=0;n<=100;n++)

{y=(200)-x[t]\*100;

putpixel((X+100),y,WHITE);}

X++;

}

}

break;

default :

printf("Invalid grade\n" );

}

delay(50000);

}

**EXPERIMENT NO. 3**

**3.To perform Discrete Correlation**

**Aim**:

To study mathematical operation Correlation and measure degree of similarity between two signals

**Objective:**

1. Write a function to find correlation operation.

2. Calculate correlation of a DT signals and verify the results using mathematical formulation.

3. Measure the degree of similarity using Carl’s Correlation Coefficient formula in time domain.

**Input Specifications:**

1. Length of first Signal L and signal values.

2. Length of second Signal M and signal values.

**Problem Definition:**

1. Find auto correlation of input signal. What is the significance of value of output signal value at n=0?.

2. Find auto correlation of delayed input signal.

3. Find cross correlation of input signal and delayed input signal,

4. Find cross correlation of input signal and scaled delayed input signal.

5. Compare the resultant signals. Give your conclusion.

6. Take two input finite length DT signals and develop a function to find Carl’s Correlation Coefficient value. Determine the degree of similarity of two signals from the calculated Carl’s Correlation Coefficient value.

**Theory:**

**Correlation**

Correlation is a mathematical operation that closely resembles convolution. Correlation is basically used to compare two signals. Correlation is the measure of the degree to which two signals are similar. The correlation of two signals is divided into two ways: (i) Cross-correlation, (ii) Auto-correlation. Correlation is a bivariate analysis that measures the strengths of association between two variables and the direction of the relationship. In terms of the strength of relationship, the value of the correlation coefficient varies between +1 and -1. When the value of the correlation coefficient lies around ± 1, then it is said to be a perfect degree of association between the two variables. As the correlation coefficient value goes towards 0, the relationship between the two variables will be weaker. the direction of the relationship is simply the + (indicating a positive relationship between the variables) or - (indicating a negative relationship between the variables) sign of the correlation. Usually, in statistics, we measure four types of correlations: Pearson correlation, Kendall rank correlation, Spearman correlation, and the Point-Biserial correlation. The software below allows you to very easily conduct a correlation.

Pearson r correlation: Pearson r correlation is the most widely used correlation statistic to measure the degree of the relationship between linearly related variables. For example, in the stock market, if we want to measure how two stocks are related to each other, Pearson r correlation is used to measure the degree of relationship between the two. The Point-biserial correlation is conducted with the Pearson correlation formula except that one of the variables is dichotomous. The following formula is used to calculate the Pearson r correlation:

pearson r correlation

r = Pearson r correlation coefficient

N = number of value in each data set

∑xy = sum of the products of paired scores

∑x = sum of x scores

∑y = sum of y scores

∑x2= sum of squared x scores

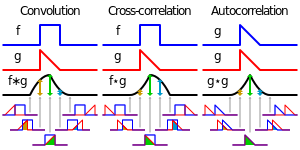
∑y2= sum of squared y scores

In [signal processing](https://en.wikipedia.org/wiki/Signal_processing), **cross-correlation** is a measure of similarity of two series as a function of the lag of one relative to the other. This is also known as a *sliding*[*dot product*](https://en.wikipedia.org/wiki/Dot_product) or *sliding inner-product*. It is commonly used for searching a long signal for a shorter, known feature. It has applications in[pattern recognition](https://en.wikipedia.org/wiki/Pattern_recognition), [single particle analysis](https://en.wikipedia.org/wiki/Single_particle_analysis), [electron tomography](https://en.wikipedia.org/wiki/Electron_tomography), [averaging](https://en.wikipedia.org/wiki/Averaging), [cryptanalysis](https://en.wikipedia.org/wiki/Cryptanalysis), and [neurophysiology](https://en.wikipedia.org/wiki/Neurophysiology).

For continuous functions *f* and *g*, the cross-correlation is defined as**:**

where  denotes the [complex conjugate](https://en.wikipedia.org/wiki/Complex_conjugate) of  and  is the lag.

Similarly, for discrete functions, the cross-correlation is defined as**:**

(f\star g)[n]\ {\stackrel {\mathrm {def} }{=}}\sum _{m=-\infty }^{\infty }f^{*}[m]\ g[m+n].

**Visual comparison of**[**convolution**](https://en.wikipedia.org/wiki/Convolution)**, cross-correlation and**[**autocorrelation**](https://en.wikipedia.org/wiki/Autocorrelation)**.**

The cross-correlation is similar in nature to the [convolution](https://en.wikipedia.org/wiki/Convolution) of two functions.

In an [autocorrelation](https://en.wikipedia.org/wiki/Autocorrelation), which is the cross-correlation of a signal with itself, there will always be a peak at a lag of zero, and its size will be the signal power.

In [probability](https://en.wikipedia.org/wiki/Probability) and [statistics](https://en.wikipedia.org/wiki/Statistics), the term **cross-correlations** is used for referring to the [correlations](https://en.wikipedia.org/wiki/Covariance_and_correlation) between the entries of two [random vectors](https://en.wikipedia.org/wiki/Multivariate_random_variable) *X* and *Y*, while the *autocorrelations* of a random vector *X* are considered to be the [correlations](https://en.wikipedia.org/wiki/Covariance_and_correlation) between the entries of *X* itself, those forming the [correlation matrix](https://en.wikipedia.org/wiki/Correlation_matrix) (matrix of correlations) of *X*. This is analogous to the distinction between [autocovariance](https://en.wikipedia.org/wiki/Autocovariance) of a random vector and [cross-covariance](https://en.wikipedia.org/wiki/Cross-covariance) of two random vectors. One more distinction to point out is that in [probability](https://en.wikipedia.org/wiki/Probability) and [statistics](https://en.wikipedia.org/wiki/Statistics) the definition of *correlation* always includes a standardising factor in such a way that correlations have values between −1 and +1.

If  and  are two [independent](https://en.wikipedia.org/wiki/Independent_(probability)) [random variables](https://en.wikipedia.org/wiki/Random_variable) with [probability density functions](https://en.wikipedia.org/wiki/Probability_density_function) *f* and *g*, respectively, then the probability density of the difference  is formally given by the cross-correlation (in the signal-processing sense) ; however this terminology is not used in probability and statistics. In contrast, the [convolution](https://en.wikipedia.org/wiki/Convolution)  (equivalent to the cross-correlation of *f*(*t*) and *g*(−*t*) ) gives the probability density function of the sum .

Properties

* The cross-correlation of functions *f*(*t*) and *g*(*t*) is equivalent to the [convolution](https://en.wikipedia.org/wiki/Convolution) of *f*\*(−*t*) and *g*(*t*). That is:

If *f* is a [Hermitian function](https://en.wikipedia.org/wiki/Hermitian_function), then

* If both *f* and *g* are Hermitian, then .
* Analogous to the [convolution theorem](https://en.wikipedia.org/wiki/Convolution_theorem), the cross-correlation satisfies

where  denotes the [Fourier transform](https://en.wikipedia.org/wiki/Fourier_transform), and an asterisk again indicates the complex conjugate. Coupled with [fast Fourier transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform) algorithms, this property is often exploited for the efficient numerical computation of cross-correlations [[2]](https://en.wikipedia.org/wiki/Cross-correlation#cite_note-KAP-2) (see [circular cross-correlation](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#Circular_convolution_theorem_and_cross-correlation_theorem)).

The cross-correlation is related to the [spectral density](https://en.wikipedia.org/wiki/Spectral_density) (see [Wiener–Khinchin theorem](https://en.wikipedia.org/wiki/Wiener%E2%80%93Khinchin_theorem)).The cross-correlation of a convolution of *f* and *h* with a function *g* is the convolution of the cross-correlation of *f* and *g* with the kernel *h*

**Autocorrelation**, also known as **serial correlation**, is the [correlation](https://en.wikipedia.org/wiki/Correlation) of a [signal](https://en.wikipedia.org/wiki/Signal_(information_theory)) with itself at different points in time. Informally, it is the similarity between observations as a function of the time lag between them. It is a mathematical tool for finding repeating patterns, such as the presence of a periodic signal obscured by noise, or identifying the [missing fundamental](https://en.wikipedia.org/wiki/Missing_fundamental) frequency in a signal implied by its [harmonic](https://en.wikipedia.org/wiki/Harmonic) frequencies. It is often used in [signal processing](https://en.wikipedia.org/wiki/Signal_processing) for analyzing functions or series of values, such as [time domain](https://en.wikipedia.org/wiki/Time_domain) signals.

[Unit root](https://en.wikipedia.org/wiki/Unit_root) processes, [trend stationary](https://en.wikipedia.org/wiki/Trend_stationary) processes, [autoregressive processes](https://en.wikipedia.org/wiki/Autoregressive_process), and [moving average processes](https://en.wikipedia.org/wiki/Moving_average_process) are specific forms of processes with autocorrelation.

## Definitions

Different fields of study define autocorrelation differently, and not all of these definitions are equivalent. In some fields, the term is used interchangeably with [autocovariance](https://en.wikipedia.org/wiki/Autocovariance).

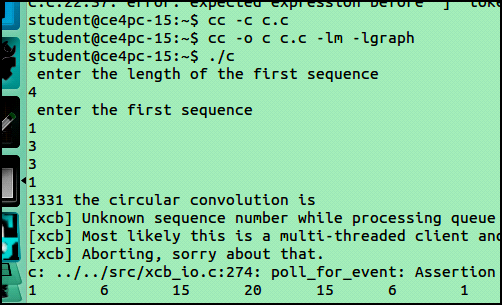
### Statistics

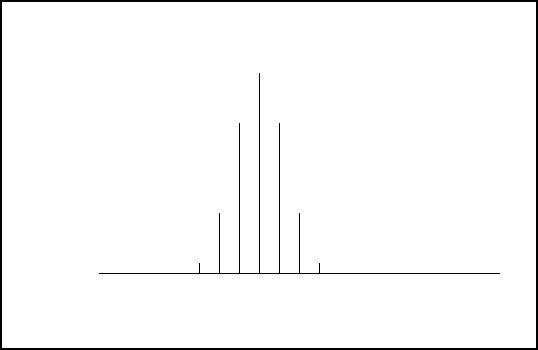
In [statistics](https://en.wikipedia.org/wiki/Statistics), the autocorrelation of a [random process](https://en.wikipedia.org/wiki/Random_process) is the [correlation](https://en.wikipedia.org/wiki/Correlation) between values of the process at different times, as a function of the two times or of the time lag. Let *X* be a stochastic process, and *t* be any point in time. (*t* may be an [integer](https://en.wikipedia.org/wiki/Integer)for a [discrete-time](https://en.wikipedia.org/wiki/Discrete-time) process or a [real number](https://en.wikipedia.org/wiki/Real_number) for a [continuous-time](https://en.wikipedia.org/wiki/Continuous-time) process.) Then *Xt* is the value (or [realization](https://en.wikipedia.org/wiki/Realization_(probability))) produced by a given [run](https://en.wikipedia.org/wiki/Execution_(computing)) of the process at time *t*. Suppose that the process has [mean](https://en.wikipedia.org/wiki/Mean) *μt* and [variance](https://en.wikipedia.org/wiki/Variance) *σt2* at time *t*, for each *t*. Then the definition of the autocorrelation between times *s* and *t* is where "E" is the [expected value](https://en.wikipedia.org/wiki/Expected_value) operator. Note that this expression is not well-defined for all time series or processes, because the mean may not exist, or the variance may be zero (for a constant process) or infinite (for processes with distribution lacking well-behaved moments, such as certain types of power law). If the function *R* is well-defined, its value must lie in the range [−1, 1], with 1 indicating perfect correlation and −1 indicating perfect [anti-correlation](https://en.wikipedia.org/wiki/Anti-correlation).

**Expected Input:**

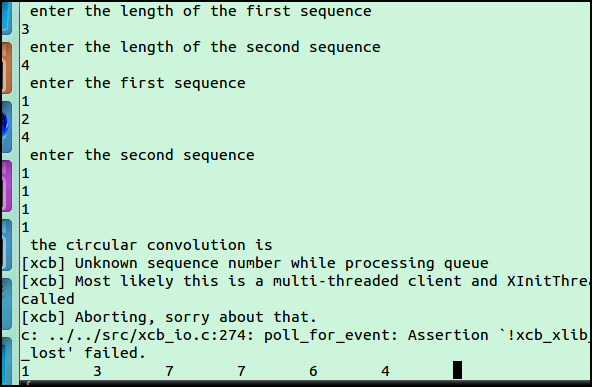
**Expected Output:**

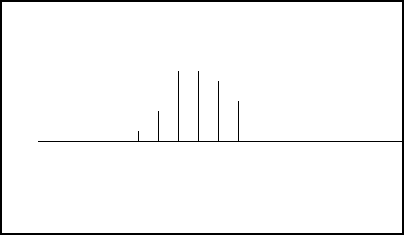
**Auto Correlation Output:-**





**Correlation Output: -**





**Conclusion:**

Thus we studied & implemented mathematical operation Correlation and measure degree of similarity between two signals and achieved desired result.

**Autocorrelation Program: -**

#include<stdio.h>

#include<graphics.h>

int m,n,x[30],h[30],y[30],i,j, k,x2[30],a[30],p,rh2[30];

void main()

{int gd=DETECT,gm,option;

int Y=0,X=200,A=0;

printf(" enter the length of the first sequence\n");

scanf("%d",&m);

printf(" enter the first sequence\n");

for(i=0;i<m;i++)

scanf("%d",&x[i]);

for(j=0;j<m;j++)

{//scanf("%d",&h[j]);

h[j]=x[j];

rh2[j]=h[-j];

printf("%d",h[j]);

}p=m+m-1;

for(i=m;i<p;i++)

{h[i]=0;}

for(i=m;i<p;i++)

{x[i]=0;

}y[0]=0;

a[0]=h[0];

for(j=1;j<p;j++) a[j]=h[p-j];

for(i=0;i<p;i++)

y[0]+=x[i]\*a[i];

for(k=1;k<p;k++)

{y[k]=0;

for(j=1;j<p;j++)

x2[j]=a[j-1];

x2[0]=a[p-1];

for(i=0;i<p;i++)

{a[i]=x2[i];

y[k]+=x[i]\*x2[i];

}}printf(" the auto correlation is\n");

for(i=0;i<p;i++)

{printf("%d \t",y[i]);}

initgraph(&gd,&gm,NULL);

line(100,300,500,300);

for(i=0;i<p;i++)

{A=y[i]\*10;

if(A>=0)

{for(j=0;j<=A;j++)

{Y=(300+j)-y[i]\*10;

putpixel(X,Y,WHITE);

}}else

{for(j=0;j>=A;j--)

{Y=(300+j)-y[i]\*10;

putpixel(X,Y,WHITE);

}}X=X+20;

}delay(50000);

closegraph();

}

**Cross Correlation**

#include<stdio.h>

#include<graphics.h>

int m,n,x[30],h[30],y[30],i,j, k,x2[30],a[30],p,rh2[30];

void main()

{ int gd=DETECT,gm,option;

int Y=0,X=200,A=0;

printf(" enter the length of the first sequence\n");

scanf("%d",&m);

printf(" enter the length of the second sequence\n");

scanf("%d",&n);

printf(" enter the first sequence\n");

for(i=0;i<m;i++)

scanf("%d",&x[i]);

printf(" enter the second sequence\n");

for(j=0;j<n;j++)

{scanf("%d",&h[j]);

rh2[j]=h[-j];

}p=m+n-1;

for(i=n;i<p;i++)

{h[i]=0;

}for(i=m;i<p;i++)

{x[i]=0;

}y[0]=0;

a[0]=h[0];

for(j=1;j<p;j++) /\*folding h(n) to h(-n)\*/

a[j]=h[p-j];

for(i=0;i<p;i++)

y[0]+=x[i]\*a[i];

for(k=1;k<p;k++)

{y[k]=0;

/\*circular shift\*/

for(j=1;j<p;j++)

x2[j]=a[j-1];

x2[0]=a[p-1];

for(i=0;i<p;i++)

{a[i]=x2[i];

y[k]+=x[i]\*x2[i];}}

/\*displaying the result\*/

printf(" the circular convolution is\n");

for(i=0;i<p;i++)

{printf("%d \t",y[i]);}

initgraph(&gd,&gm,NULL);

line(100,300,500,300);

for(i=0;i<p;i++)

{A=y[i]\*10;

if(A>=0)

{ for(j=0;j<=A;j++)

{ Y=(300+j)-y[i]\*10;

putpixel(X,Y,WHITE); } }

else {

for(j=0;j>=A;j--)

{ Y=(300+j)-y[i]\*10;

putpixel(X,Y,WHITE); }}

X=X+20;

}delay(50000); closegraph();

}

**EXPERIMENT NO. 4**

**4. To perform Discrete Convolution**

**Aim:**

The aim of this experiment is to study mathematical operation such as Linear convolution, Circular convolution, Linear convolution using circular convolution.

**Objective:**

1. Develop a function to find Linear Convolution and Circular Convolution

2. Calculate Linear Convolution, Circular Convolution, Linear Convolution using Circular Convolution and verify the results using mathematical formulation.

3. Conclude on aliasing effect in Circular convolution

**Input Specifications:**

1. Length of first Signal L and signal values.

2. Length of second Signal M and signal values.

**Problem Definition:**

1. Find Linear Convolution and Circular Convolution of L point sequence x[n] and M point sequence h[n].

2. Find Linear Convolution of L point sequence x[n] and M point sequence h[n] using Circular convolution.

3. Give your conclusion about No of values in linearly convolved signal, and Aliasing effect in Circular Convolution.

**Theory:**

**Linear Convolution**

Convolution is a formal mathematical operation, just as multiplication, addition, and integration. Addition takes two numbers and produces a third number, while convolution takes two signals and produces a third signal. Convolution is used in the mathematics of many fields, such as probability and statistics. In linear systems, convolution is used to describe the relationship between three signals of interest: the input signal, the impulse response, and the output signal. If the input and impulse response of a system are x[n] and h[n] respectively, the convolution is given by the expression, x[n] \* h[n] = ε x[k] h[n-k] Where k ranges between -∞ and ∞ If, x(n) is a M- point sequence h(n) is a N – point sequence then, y(n) is a (M+N-1) – point sequence. In this equation, x(k), h(n-k) and y(n) represent the input to and output from the system at time n. Here we could see that one of the inputs is shifted in time by a value every time it is multiplied with the other input signal. Linear Convolution is quite often used as a method of implementing filters of various types.

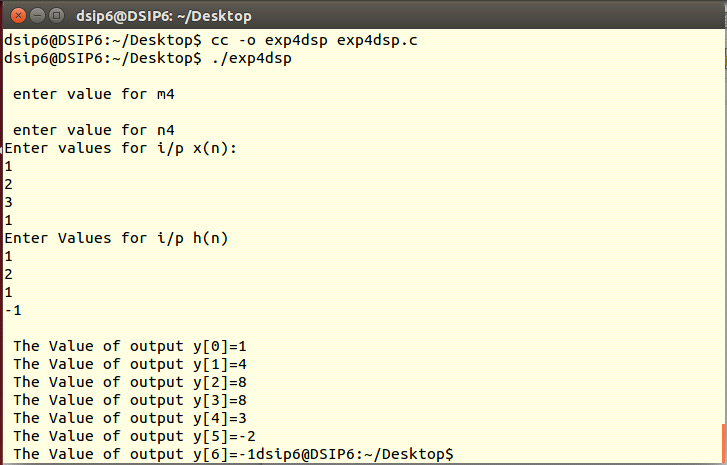
**Circular Convolution**

The circular convolution, also known as cyclic convolution. A convolution operation that contains a circular shift is called circular convolution. Circular convolution of two sequences x1[n] and x2[n] is given by x1[n]\*x2[n] = εk x1[k] x2((n-k))N, 0≤ n ≤N-1 where k ranges between 0 and N-1. In circular convolution the length of the output sequence will be equal to length of the input sequence ie. length(y)=length(x) So first perform linear convolution using any of the methods u find easier. If m is the length of 'x' and n is the length of the 'h' then length of 'yl' from linear conv is m+n-1. Since length of output from circular conv is m, we will bring the last n-1 terms from 'yl' and add them to first n-1 terms. So the obtained output is circularly convoluted output.

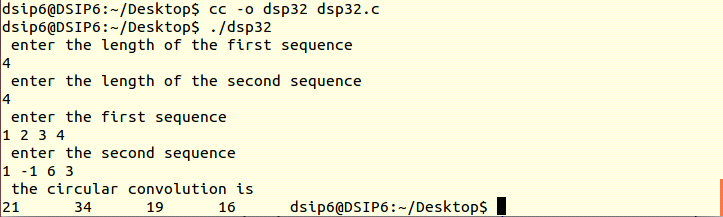
**Expected Input:**

**Expected Output:** .

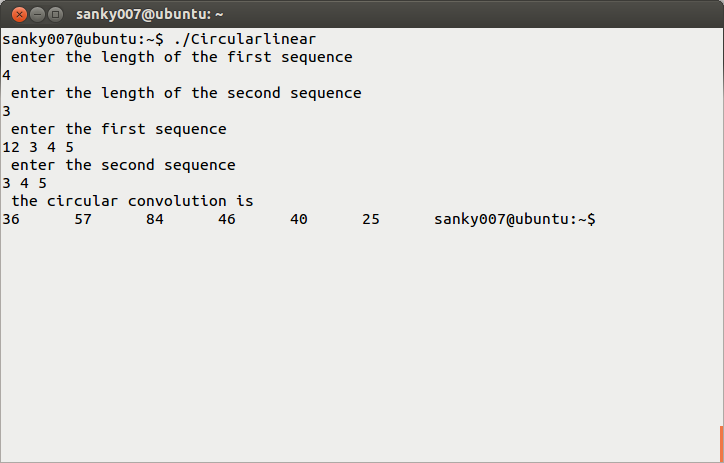
**Linear Convolution Output:**



**Circular Convolution Output:**



**Linear Circular Convolution Output: -**



**Conclusion:** Thus, the Linear convolution, Circular convolution, Linear convolution using circular convolution of two given discrete sequence has performed and the result is displayed.

**Programs: -**

**Linear Convolution:**

#include<stdio.h>

int x[15],h[15],y[15];

main()

{

int i,j,m,n;

printf("\n enter value for m");

scanf("%d",&m);

printf("\n enter value for n");

scanf("%d",&n);

printf("Enter values for i/p x(n):\n");

for(i=0;i<m;i++)

scanf("%d",&x[i]);

printf("Enter Values for i/p h(n) \n");

for(i=0;i<n; i++)

scanf("%d",&h[i]);

// padding of zeors

for(i=m;i<=m+n-1;i++)

x[i]=0;

for(i=n;i<=m+n-1;i++)

h[i]=0;

/\* convolution operation \*/

for(i=0;i<m+n-1;i++)

{

y[i]=0;

for(j=0;j<=i;j++)

{

y[i]=y[i]+(x[j]\*h[i-j]);

}

}

//displaying the o/p

for(i=0;i<m+n-1;i++)

printf("\n The Value of output y[%d]=%d",i,y[i]);

}

**Circular convolution:**

#include<stdio.h>

int m,n,x[30],h[30],y[30],i,j, k,x2[30],a[30];

void main()

{

printf(" enter the length of the first sequence\n");

scanf("%d",&m);

printf(" enter the length of the second sequence\n");

scanf("%d",&n);

printf(" enter the first sequence\n");

for(i=0;i<m;i++)

scanf("%d",&x[i]);

printf(" enter the second sequence\n");

for(j=0;j<n;j++)

scanf("%d",&h[j]);

if(m-n!=0) /\*If length of both sequences are not equal\*/

{

if(m>n) /\* Pad the smaller sequence with zero\*/

{

for(i=n;i<m;i++)

h[i]=0;

n=m;

}

for(i=m;i<n;i++)

x[i]=0;

m=n;

}

y[0]=0;

a[0]=h[0];

for(j=1;j<n;j++) /\*folding h(n) to h(-n)\*/

a[j]=h[n-j];

/\*Circular convolution\*/

for(i=0;i<n;i++)

y[0]+=x[i]\*a[i];

for(k=1;k<n;k++)

{

y[k]=0;

/\*circular shift\*/

for(j=1;j<n;j++)

x2[j]=a[j-1];

x2[0]=a[n-1];

for(i=0;i<n;i++)

{

a[i]=x2[i];

y[k]+=x[i]\*x2[i];

}

}

/\*displaying the result\*/

printf(" the circular convolution is\n");

for(i=0;i<n;i++)

printf("%d \t",y[i]); }

**Circular using Linear Convolution**

#include<stdio.h>

int m,n,x[30],h[30],y[30],i,j, k,l,x2[30],a[30];

void main()

{

printf(" enter the length of the first sequence\n");

scanf("%d",&m);

printf(" enter the length of the second sequence\n");

scanf("%d",&n);

printf(" enter the first sequence\n");

for(i=0;i<m;i++)

scanf("%d",&x[i]);

printf(" enter the second sequence\n");

for(j=0;j<n;j++)

scanf("%d",&h[j]);

if(m-n!=0) /\*If length of both sequences are not equal\*/

{

l=m+n-1;

if(m>n || m<n) /\* Pad the smaller sequence with zero\*/

{

for(i=l;i<m;i++)

for(i=l;i<n;i++)

h[i]=0;

n=l;

m=l;

}

}

y[0]=0;

a[0]=h[0];

for(j=1;j<n;j++) /\*folding h(n) to h(-n)\*/

a[j]=h[n-j];

/\*Circular convolution\*/

for(i=0;i<n;i++)

y[0]+=x[i]\*a[i];

for(k=1;k<n;k++)

{

y[k]=0;

/\*circular shift\*/

for(j=1;j<n;j++)

x2[j]=a[j-1];

x2[0]=a[n-1];

for(i=0;i<n;i++)

{

a[i]=x2[i];

y[k]+=x[i]\*x2[i];

}

}

/\*displaying the result\*/

printf(" the circular convolution is\n");

for(i=0;i<n;i++)

printf("%d \t",y[i]);

}

**EXPERIMENT NO. 5**

**5. To perform Discrete Fourier Transform**

**Aim:**

The aim of this experiment is to study magnitude spectrum of the DT signal.

**Objective:**

1. Develop a function to perform DFT of N point signal

2. Calculate DFT of a DT signal and Plot spectrum of the signal.

3. Conclude the effect of zero padding on magnitude spectrum.

4. Calculate the number of real multiplications and real additions required to find DFT.

**Input Specifications:**

1. Length of Signal N

2. Signal values

**Problem Definition:**

1. Take any four-point sequence x[n].

Find DFT X[k].

Compute number of real multiplications and real additions required to find X[k].

Plot Magnitude Spectrum of the signal.

2. Append the input sequence by four zeros. Find DFT and plot magnitude spectrum. Repeat the same by appending the sequence by eight zeros. Observe and compare the magnitude spectrum. Give your conclusion.

**Theory:**

In [mathematics](https://en.wikipedia.org/wiki/Mathematics), the discrete Fourier transform (DFT) converts a finite sequence of equally-spaced [samples](https://en.wikipedia.org/wiki/Sampling_(signal_processing)) of a [function](https://en.wikipedia.org/wiki/Function_(mathematics)) into an equivalent-length sequence of equally-spaced samples of the [discrete-time Fourier transform](https://en.wikipedia.org/wiki/Discrete-time_Fourier_transform) (DTFT), which is a [complex-valued](https://en.wikipedia.org/wiki/Complex_number) function of frequency. The interval at which the DTFT is sampled is the reciprocal of the duration of the input sequence. An inverse DFT is a [Fourier series](https://en.wikipedia.org/wiki/Fourier_series), using the DTFT samples as coefficients of [complex](https://en.wikipedia.org/wiki/Complex_number) [sinusoids](https://en.wikipedia.org/wiki/Sine_wave) at the corresponding DTFT frequencies. It has the same sample-values as the original input sequence. The DFT is therefore said to be a [frequency domain](https://en.wikipedia.org/wiki/Frequency_domain) representation of the original input sequence. If the original sequence spans all the non-zero values of a function, its DTFT is continuous (and periodic), and the DFT provides discrete samples of one cycle. If the original sequence is one cycle of a periodic function, the DFT provides all the non-zero values of one DTFT cycle. The DFT is the most important [discrete transform](https://en.wikipedia.org/wiki/Discrete_transform), used to perform [Fourier analysis](https://en.wikipedia.org/wiki/Fourier_analysis) in many practical applications.[[1]](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#cite_note-1) In [digital signal processing](https://en.wikipedia.org/wiki/Digital_signal_processing), the function is any quantity or [signal](https://en.wikipedia.org/wiki/Signal_(information_theory)) that varies over time, such as the pressure of a [sound wave](https://en.wikipedia.org/wiki/Sound_wave), a [radio](https://en.wikipedia.org/wiki/Radio) signal, or daily [temperature](https://en.wikipedia.org/wiki/Temperature) readings, sampled over a finite time interval (often defined by a [window function](https://en.wikipedia.org/wiki/Window_function)[[2]](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#cite_note-2)). In [image processing](https://en.wikipedia.org/wiki/Image_processing), the samples can be the values of [pixels](https://en.wikipedia.org/wiki/Pixel) along a row or column of a [raster image](https://en.wikipedia.org/wiki/Raster_image). The DFT is also used to efficiently solve [partial differential equations](https://en.wikipedia.org/wiki/Partial_differential_equations), and to perform other operations such as [convolutions](https://en.wikipedia.org/wiki/Convolution) or multiplying large integers.Since it deals with a finite amount of data, it can be implemented in [computers](https://en.wikipedia.org/wiki/Computer) by [numerical algorithms](https://en.wikipedia.org/wiki/Numerical_algorithm) or even dedicated [hardware](https://en.wikipedia.org/wiki/Digital_circuit). These implementations usually employ efficient [fast Fourier transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform) (FFT) algorithms;[[3]](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#cite_note-colley-3) so much so that the terms "FFT" and "DFT" are often used interchangeably. Prior to its current usage, the "FFT" [initialism](https://en.wikipedia.org/wiki/Initialism) may have also been used for the ambiguous term "[finite Fourier transform](https://en.wikipedia.org/wiki/Finite_Fourier_transform_(disambiguation))".

Definition

The [sequence](https://en.wikipedia.org/wiki/Sequence) of *N* [complex numbers](https://en.wikipedia.org/wiki/Complex_number)  is transformed into an *N*-periodic sequence of complex numbers:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Because of [periodicity](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#Periodicity), the customary domain of **k** actually computed is [*0*, *N* − 1]. That is always the case when the DFT is implemented via the [Fast Fourier transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform) algorithm. But other common domains are  [−*N*/2, *N*/2 − 1]  (*N* even)  and  [−(*N* − 1)/2, (*N* − 1)/2]  (*N* odd), as when the left and right halves of an FFT output sequence are swapped.[[4]](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#cite_note-5)

The transform is sometimes denoted by the symbol , as in  or  or .

[**Eq.1**](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#math_Eq.1) can be interpreted or derived in various ways, for example:

* It completely describes the [discrete-time Fourier transform](https://en.wikipedia.org/wiki/Discrete-time_Fourier_transform) (DTFT) of an *N*-periodic sequence, which comprises only discrete frequency components. ([Using the DTFT with periodic data](https://en.wikipedia.org/wiki/Discrete-time_Fourier_transform#Periodic_data))
* It can also provide uniformly spaced samples of the continuous DTFT of a finite length sequence. ([Sampling the DTFT](https://en.wikipedia.org/wiki/Discrete-time_Fourier_transform#Sampling_the_DTFT))
* It is the [cross correlation](https://en.wikipedia.org/wiki/Cross_correlation) of the *input* sequence, *xn*, and a complex sinusoid at frequency *k*/*N*.  Thus it acts like a [matched filter](https://en.wikipedia.org/wiki/Matched_filter) for that frequency.
* It is the discrete analogy of the formula for the coefficients of a [Fourier series](https://en.wikipedia.org/wiki/Fourier_series):

The normalization factor multiplying the DFT and IDFT (here 1 and 1/*N*) and the signs of the exponents are merely [conventions](https://en.wikipedia.org/wiki/Sign_convention), and differ in some treatments. The only requirements of these conventions are that the DFT and IDFT have opposite-sign exponents and that the product of their normalization factors be 1/*N*.  A normalization of  for both the DFT and IDFT, for instance, makes the transforms unitary.\scriptstyle {\sqrt {1/N}}

In the following discussion the terms "sequence" and "vector" will be considered interchangeable.

Using [Euler's formula](https://en.wikipedia.org/wiki/Euler%27s_formula), the DFT formulae can be converted to the trigonometric forms sometimes used in engineering and computer science:

Given a sequence of *N* samples *f*(*n*), indexed by *n*= 0..*N*-1, the Discrete Fourier Transform (DFT) is defined as *F*(*k*), where *k*=0..*N*-1:

equation

*F*(*k*) are often called the 'Fourier Coefficients' or 'Harmonics'.

The sequence *f*(*n*) can be calculated from *F*(*k*) using the Inverse Discrete Fourier Transform (IDFT):

equation

In general, both *f*(*n*) and *F*(*k*) are complex.

Annex A shows that the IDFT defined above really is an *inverse* DFT.

Conventionally, the sequences *f*(*n*) and *F*(*k*) is referred to as 'time domain' data and 'frequency domain' data respectively. Of course there is no reason why the samples in *f*(*n*) need be samples of a time dependant signal. For example, they could be spatial image samples (though in such cases a 2 dimensional set would be more common).

Although we have stated that both *n* and *k* range over 0..*N*-1, the definitions above have a periodicity of *N*:

equation

So both *f*(*n*) and *F*(*k*) are defined for all (integral) *n* and *k* respectively, but we only need to calculate values in the range 0..*N*-1. Any other points can be obtained using the above periodicity property.

For the sake of simplicity, when considering various Fast Fourier Transform (FFT) algorithms, we shall ignore the scaling factors and simply define the FFT and Inverse FFT (IFFT) like this:

equation

equation

In fact, we shall only consider the FFT algorithms in detail. The inverse FFT (IFFT) is easily obtained from the FFT.

Here are some simple DFT's expressed as matrix multiplications.

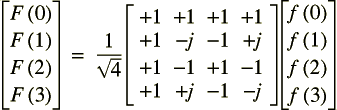
1 point DFT:

equation

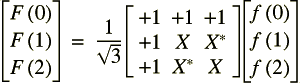
2 point DFT:

equation

4 point DFT:



3 point DFT:



equation

Note that each of the matrix multipliers can be inverted by conjugating the elements. This what we would expect, given that the only difference between the DFT and IDFT is the sign of the complex exponential argument.

Here's another couple of useful transforms:

If..

equation

equation

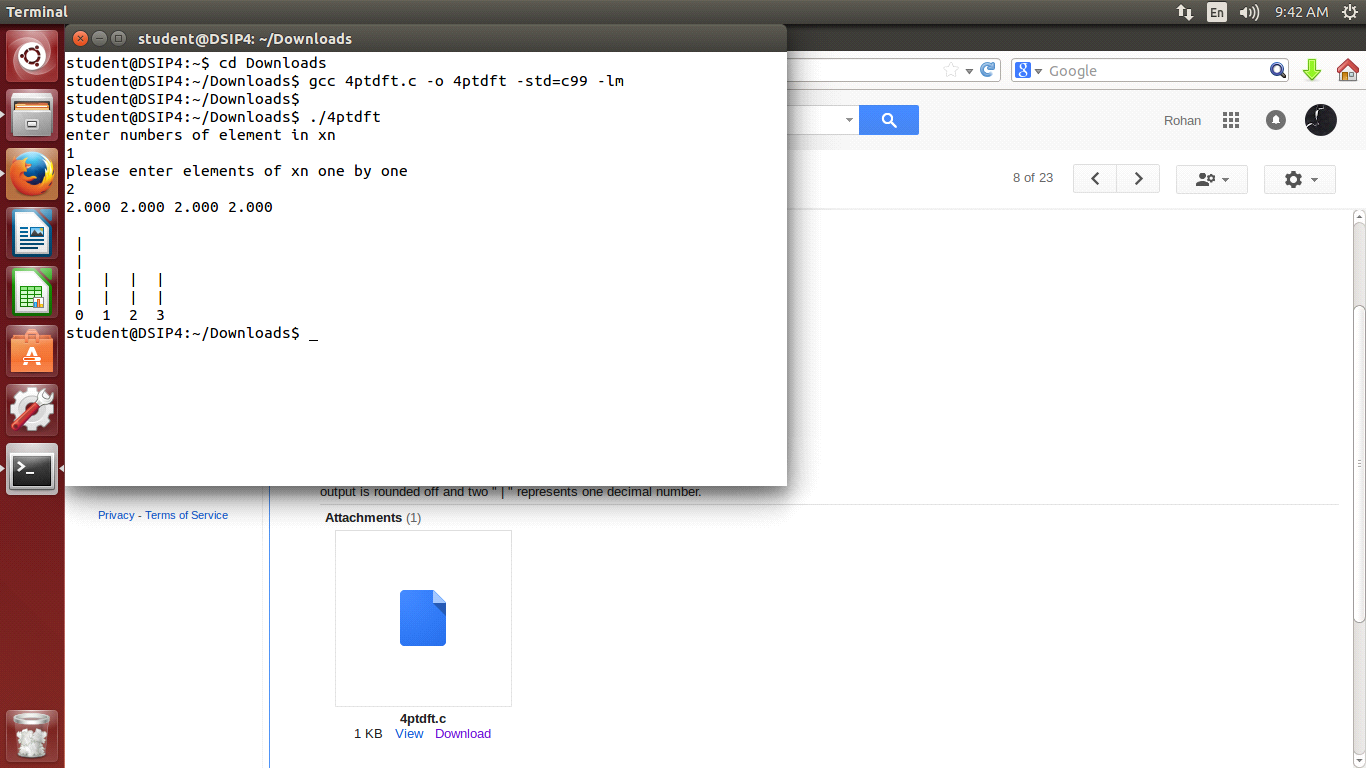
equation

This is the 'Delta Function'. The usual implied periodicity has been made explicit by using *MOD N*. The DFT is therefore:

equation

**Expected Input:**

**Expected Output:**



**Conclusion:**

Hence we implemented magnitude spectrum of the DT signal.

**Program: -**

#include <stdio.h>

#include <math.h>

void star(float[], int);

int max\_find(float [],int );

int main()

{

int n;

float w4[4][4] = {{1, 1, 1, 1}, {1, 0, -1, 0}, {1, -1, 1, -1}, {1, 0, -1, 0}};

float xn[4], j[4] = {0}, ans[4] = {0};

float k[4];

printf("enter numbers of element in xn\n");

scanf("%d", &n);

printf("please enter elements of xn one by one\n");

for (int i = 0; i < n; i++)

scanf("%f", &xn[i]);

for (int i = n; i < 4; i++)

xn[i] = 0;

for(int i = 0; i < 4;i++)

for(int k = 0;k < 4 ; k++)

ans[i] = ans[i] + w4[i][k] \* xn[k];

j[1] = -xn[1] + xn[3];

j[3] = xn[1] - xn[3];

for ( int i = 0; i < 4; i++)

{

k[i] = sqrt(ans[i]\*ans[i] + j[i]\*j[i]);

}

for ( int i = 0; i < 4; i++)

{

printf("%.3f ",k[i]);

}

printf("\n");

for (int i = 0; i < n; i++)

{

k[i] = 2\*k[i]; // representing two || for one decimal

if ( (k[i] - ((int) k[i])) > 0.5 )

{

k[i] =( (int) k[i] ) + 1;

}

else

k[i] = (int)k[i];

}

star(k, 4);

}

void star(float h3[], int n)

{ char array[50][50] ;

for (int i = 0; i < 50; i++)

for (int j = 0; j < 50; j++)

array[i][j] = ' ';

int max = max\_find(h3, n);

for (int i = 0; i < n; i++)

{

for (int j = 0; j < h3[i]; j++)

{

array[j][i] = '|' ;

}

}

for (int i = max; i >= 0; i--)

{

for (int j = 0; j < n; j++)

{

printf(" ");

printf("%c",array[i][j]);

printf(" ");

}

printf("\n");

}

for (int i = 0; i < n; i++) {

printf(" ");

printf("%d",i);

printf(" ");

}

printf("\n");

}

int max\_find(float h3[], int n)

{

int max = h3[0];

for (int i = 1; i < n; i++) {

max = (max<h3[i])?h3[i]:max;

}

return max;

}

**EXPERIMENT NO. 6**

**6. To perform Fast Fourier Transform**

**Aim:**

To implement computationally fast algorithms.

**Objective:**

1. Develop a program to perform FFT of N point signal.

2. Calculate FFT of a given DT signal and verify the results using mathematical formulation.

3. Illustrate the computational efficiency of FFT.

**Input Specifications:**

· Length of Signal N

· Signal values

**Problem Definition:**

Take any eight-point sequence x[n].

· Find FFT X[k].

· Write number of real multiplications and real additions involved in finding X[k].

**Theory:**

A fast Fourier transform (FFT) algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors.[1] As a result, it manages to reduce the complexity of computing the DFT from {\displaystyle O(n^{2})} O(n^{2}), which arises if one simply applies the definition of DFT, to {\displaystyle O(n\log n)} O(n\log n), where {\displaystyle n} n is the data size.

Fast Fourier transforms are widely used for many applications in engineering, science, and mathematics. The basic ideas were popularized in 1965, but some algorithms had been derived as early as 1805.[2] In 1994, Gilbert Strang described the FFT as "the most important numerical algorithm of our lifetime" and it was included in Top 10 Algorithms of 20th Century by the IEEE journal Computing in Science & Engineering.

**Overview: -**

A fast Fourier transform (FFT) algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors.[1] As a result, it manages to reduce the complexity of computing the DFT from Fast Fourier transforms are widely used for many applications in engineering, science, and mathematics. The basic ideas were popularized in 1965, but some algorithms had been derived as early as 1805.[2] In 1994, Gilbert Strang described the FFT as "the most important numerical algorithm of our lifetime"[3] and it was included in Top 10 Algorithms of 20th Century by the IEEE journal Computing in Science & Engineering.

Definition and speed

An FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly; the most important difference is that an FFT is much faster. (In the presence of round-off error, many FFT algorithms are also much more accurate than evaluating the DFT definition directly, as discussed below.)

Let x0, ...., xN−1 be complex numbers. The DFT is defined by the formula

{\displaystyle X\_{k}=\sum \_{n=0}^{N-1}x\_{n}e^{-i2\pi kn/N}\qquad k=0,\dots ,N-1.} {\displaystyle X\_{k}=\sum \_{n=0}^{N-1}x\_{n}e^{-i2\pi kn/N}\qquad k=0,\dots ,N-1.}

Evaluating this definition directly requires O(N2) operations: there are N outputs Xk, and each output requires a sum of N terms. An FFT is any method to compute the same results in O(N log N) operations. All known FFT algorithms require Θ(N log N) operations, although there is no known proof that a lower complexity score is impossible.(Johnson and Frigo, 2007)

To illustrate the savings of an FFT, consider the count of complex multiplications and additions. Evaluating the DFT's sums directly involves N2 complex multiplications and N(N−1) complex additions, of which O(N) operations can be saved by eliminating trivial operations such as multiplications by 1. The radix-2 Cooley–Tukey algorithm, for N a power of 2, can compute the same result with only (N/2)log2(N) complex multiplications (again, ignoring simplifications of multiplications by 1 and similar) and N log2(N) complex additions. In practice, actual performance on modern computers is usually dominated by factors other than the speed of arithmetic operations and the analysis is a complicated subject (see, e.g., Frigo & Johnson, 2005), but the overall improvement from O(N2) to O(N log N) remains.

**Definition and speed**

An FFT computes the [DFT](https://en.wikipedia.org/wiki/Discrete_Fourier_transform) and produces exactly the same result as evaluating the DFT definition directly; the most important difference is that an FFT is much faster. (In the presence of [round-off error](https://en.wikipedia.org/wiki/Round-off_error), many FFT algorithms are also much more accurate than evaluating the DFT definition directly, as discussed below.)

Let x0, ...., xN−1 be [complex numbers](https://en.wikipedia.org/wiki/Complex_number). The DFT is defined by the formula

Evaluating this definition directly requires O(N2) operations: there are N outputs Xk, and each output requires a sum of N terms. An FFT is any method to compute the same results in O(N log N) operations. All known FFT algorithms require [Θ](https://en.wikipedia.org/wiki/Big_O_notation#Use_in_computer_science)(N log N) operations, although there is no known proof that a lower complexity score is impossible.(Johnson and Frigo, 2007)

To illustrate the savings of an FFT, consider the count of complex multiplications and additions. Evaluating the DFT's sums directly involves N2 complex multiplications and N(N−1) complex additions, of which O(N) operations can be saved by eliminating trivial operations such as multiplications by 1. The radix-2 [Cooley–Tukey algorithm](https://en.wikipedia.org/wiki/Fast_Fourier_transform#Cooley.E2.80.93Tukey_algorithm), for N a power of 2, can compute the same result with only (N/2)log2(N) complex multiplications (again, ignoring simplifications of multiplications by 1 and similar) and N log2(N) complex additions. In practice, actual performance on modern computers is usually dominated by factors other than the speed of arithmetic operations and the analysis is a complicated subject (see, e.g., Frigo & Johnson, 2005), but the overall improvement from O(N2) to O(N log N) remains.

## Computational issues

### Bounds on complexity and operation counts

A fundamental question of longstanding theoretical interest is to prove lower bounds on the [complexity](https://en.wikipedia.org/wiki/Computational_complexity_theory) and exact operation counts of fast Fourier transforms, and many open problems remain. It is not even rigorously proved whether DFTs truly require Ω(*N* log *N*) (i.e., order *N* log *N* or greater) operations, even for the simple case of [power of two](https://en.wikipedia.org/wiki/Power_of_two) sizes, although no algorithms with lower complexity are known. In particular, the count of arithmetic operations is usually the focus of such questions, although actual performance on modern-day computers is determined by many other factors such as [cache](https://en.wikipedia.org/wiki/Cache_(computing)) or [CPU pipeline](https://en.wikipedia.org/wiki/Pipeline_(computing)) optimization.

Following pioneering work by [Winograd](https://en.wikipedia.org/wiki/Shmuel_Winograd) (1978), a tight Θ(*N*) lower bound *is* known for the [number of real multiplications required by an FFT](https://en.wikipedia.org/wiki/Arithmetic_complexity_of_the_discrete_Fourier_transform). It can be shown that only {\displaystyle 4N-2\log _{2}^{2}N-2\log _{2}N-4}irrational real multiplications are required to compute a DFT of power-of-two length N=2^{m}. Moreover, explicit algorithms that achieve this count are known (Heideman & Burrus, 1986; Duhamel, 1990). Unfortunately, these algorithms require too many additions to be practical, at least on modern computers with hardware multipliers (Duhamel, 1990; Frigo & Johnson, 2005).

A tight lower bound is *not* known on the number of required additions, although lower bounds have been proved under some restrictive assumptions on the algorithms. In 1973, Morgenstern proved an Ω(*N* log *N*) lower bound on the addition count for algorithms where the multiplicative constants have bounded magnitudes (which is true for most but not all FFT algorithms). This result, however, applies only to the unnormalized Fourier transform (which is a scaling of a unitary matrix by a factor of {\sqrt {N}}), and does not explain why the Fourier matrix is harder to compute than any other unitary matrix (including the identity matrix) under the same scaling. Pan (1986) proved an Ω(*N* log *N*) lower bound assuming a bound on a measure of the FFT algorithm's "asynchronicity", but the generality of this assumption is unclear. For the case of power-of-two *N*, Papadimitriou (1979) argued that the number N\log _{2}N of complex-number additions achieved by Cooley–Tukey algorithms is *optimal* under certain assumptions on the [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) of the algorithm (his assumptions imply, among other things, that no additive identities in the roots of unity are exploited). (This argument would imply that at least 2N\log _{2}N real additions are required, although this is not a tight bound because extra additions are required as part of complex-number multiplications.) Thus far, no published FFT algorithm has achieved fewer than N\log _{2}N complex-number additions (or their equivalent) for power-of-two *N*.

A third problem is to minimize the *total* number of real multiplications and additions, sometimes called the "arithmetic complexity" (although in this context it is the exact count and not the asymptotic complexity that is being considered). Again, no tight lower bound has been proven. Since 1968, however, the lowest published count for power-of-two *N* was long achieved by the [split-radix FFT algorithm](https://en.wikipedia.org/wiki/Split-radix_FFT_algorithm), which requires 4N\log _{2}N-6N+8 real multiplications and additions for *N* > 1. This was recently reduced to \sim {\frac {34}{9}}N\log _{2}N (Johnson and Frigo, 2007; Lundy and Van Buskirk, 2007). A slightly larger count (but still better than split radix for *N*≥256) was shown to be provably optimal for *N*≤512 under additional restrictions on the possible algorithms (split-radix-like flowgraphs with unit-modulus multiplicative factors), by reduction to a [satisfiability modulo theories](https://en.wikipedia.org/wiki/Satisfiability_modulo_theories) problem solvable by[brute force](https://en.wikipedia.org/wiki/Proof_by_exhaustion) (Haynal & Haynal, 2011).

Most of the attempts to lower or prove the complexity of FFT algorithms have focused on the ordinary complex-data case, because it is the simplest. However, complex-data FFTs are so closely related to algorithms for related problems such as real-data FFTs, [discrete cosine transforms](https://en.wikipedia.org/wiki/Discrete_cosine_transform), [discrete Hartley transforms](https://en.wikipedia.org/wiki/Discrete_Hartley_transform), and so on, that any improvement in one of these would immediately lead to improvements in the others (Duhamel & Vetterli, 1990).

### Approximations

All of the FFT algorithms discussed above compute the DFT exactly (in exact arithmetic, i.e. neglecting [floating-point](https://en.wikipedia.org/wiki/Floating-point) errors). A few "FFT" algorithms have been proposed, however, that compute the DFT *approximately*, with an error that can be made arbitrarily small at the expense of increased computations. Such algorithms trade the approximation error for increased speed or other properties. For example, an approximate FFT algorithm by Edelman et al. (1999) achieves lower communication requirements for [parallel computing](https://en.wikipedia.org/wiki/Parallel_computing) with the help of a [fast multipole method](https://en.wikipedia.org/wiki/Fast_multipole_method). A [wavelet](https://en.wikipedia.org/wiki/Wavelet)-based approximate FFT by Guo and Burrus (1996) takes sparse inputs/outputs (time/frequency localization) into account more efficiently than is possible with an exact FFT. Another algorithm for approximate computation of a subset of the DFT outputs is due to Shentov et al. (1995). The Edelman algorithm works equally well for sparse and non-sparse data, since it is based on the compressibility (rank deficiency) of the Fourier matrix itself rather than the compressibility (sparsity) of the data. Conversely, if the data are sparse—that is, if only *K* out of *N* Fourier coefficients are nonzero—then the complexity can be reduced to O(*K* log(*N*)log(*N*/*K*)), and this has been demonstrated to lead to practical speedups compared to an ordinary FFT for *N*/*K* > 32 in a large-*N* example (*N* = 222) using a probabilistic approximate algorithm (which estimates the largest *K* coefficients to several decimal places).[[15]](https://en.wikipedia.org/wiki/Fast_Fourier_transform#cite_note-Hassanieh12-15)

### Accuracy

Even the "exact" FFT algorithms have errors when finite-precision floating-point arithmetic is used, but these errors are typically quite small; most FFT algorithms, e.g. Cooley–Tukey, have excellent numerical properties as a consequence of the [pairwise summation](https://en.wikipedia.org/wiki/Pairwise_summation) structure of the algorithms. The upper bound on the [relative error](https://en.wikipedia.org/wiki/Approximation_error) for the Cooley–Tukey algorithm is O(ε log *N*), compared to O(ε*N*3/2) for the naïve DFT formula,[[13]](https://en.wikipedia.org/wiki/Fast_Fourier_transform#cite_note-GentlemanSande-13) where ε is the machine floating-point relative precision. In fact, the [root mean square](https://en.wikipedia.org/wiki/Root_mean_square) (rms) errors are much better than these upper bounds, being only O(ε √log *N*) for Cooley–Tukey and O(ε √*N*) for the naïve DFT (Schatzman, 1996). These results, however, are very sensitive to the accuracy of the twiddle factors used in the FFT (i.e. the [trigonometric function](https://en.wikipedia.org/wiki/Trigonometric_function) values), and it is not unusual for incautious FFT implementations to have much worse accuracy, e.g. if they use inaccurate [trigonometric recurrence](https://en.wikipedia.org/wiki/Generating_trigonometric_tables) formulas. Some FFTs other than Cooley–Tukey, such as the Rader–Brenner algorithm, are intrinsically less stable.

In [fixed-point arithmetic](https://en.wikipedia.org/wiki/Fixed-point_arithmetic), the finite-precision errors accumulated by FFT algorithms are worse, with rms errors growing as O(√*N*) for the Cooley–Tukey algorithm (Welch, 1969). Moreover, even achieving this accuracy requires careful attention to scaling to minimize loss of precision, and [fixed-point FFT algorithms](https://en.wikipedia.org/w/index.php?title=Fixed-point_FFT_algorithm&action=edit&redlink=1) involve rescaling at each intermediate stage of decompositions like Cooley–Tukey.

To verify the correctness of an FFT implementation, rigorous guarantees can be obtained in O(*N* log *N*) time by a simple procedure checking the linearity, impulse-response, and time-shift properties of the transform on random inputs (Ergün, 1995).

## Multidimensional FFTs

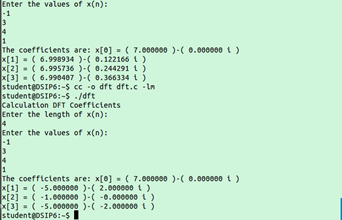
Transforms an array *x***n** with a *d*-dimensional [vector](https://en.wikipedia.org/wiki/Coordinate_vector) of indices  by a set of *d* nested summations (over  for each *j*), where the division **n**/**N**, defined as , is performed element-wise. Equivalently, it is the composition of a sequence of*d* sets of one-dimensional DFTs, performed along one dimension at a time (in any order).This compositional viewpoint immediately provides the simplest and most common multidimensional DFT algorithm, known as the **row-column** algorithm (after the two-dimensional case, below). That is, one simply performs a sequence of *d* one-dimensional FFTs (by any of the above algorithms): first you transform along the *n*1 dimension, then along the *n*2 dimension, and so on (or actually, any ordering works). This method is easily shown to have the usual O(*N* log , *N*) complexity where  is the total number of data points transformed. In particular, there are *N*/*N*1transforms of size *N*1, etcetera, so the complexity of the sequence of FFTs is:

In two dimensions, the *x***k** can be viewed as an  [matrix](https://en.wikipedia.org/wiki/Matrix_(mathematics)), and this algorithm corresponds to first performing the FFT of all the rows (resp. columns), grouping the resulting transformed rows (resp. columns) together as another  matrix, and then performing the FFT on each of the columns (resp. rows) of this second matrix, and similarly grouping the results into the final result matrix.

In more than two dimensions, it is often advantageous for [cache](https://en.wikipedia.org/wiki/Cache_(computing)) locality to group the dimensions recursively. For example, a three-dimensional FFT might first perform two-dimensional FFTs of each planar "slice" for each fixed *n*1, and then perform the one-dimensional FFTs along the *n*1 direction. More generally, an [asymptotically optimal](https://en.wikipedia.org/wiki/Asymptotically_optimal) [cache-oblivious](https://en.wikipedia.org/wiki/Cache-oblivious) algorithm consists of recursively dividing the dimensions into two groups (n_{1},\ldots ,n_{d/2}) and (n_{d/2+1},\ldots ,n_{d}) that are transformed recursively (rounding if *d* is not even) (see Frigo and Johnson, 2005). Still, this remains a straightforward variation of the row-column algorithm that ultimately requires only a one-dimensional FFT algorithm as the base case, and still has O(*N* log *N*) complexity. Yet another variation is to perform matrix [transpositions](https://en.wikipedia.org/wiki/Transpose) in between transforming subsequent dimensions, so that the transforms operate on contiguous data; this is especially important for [out-of-core](https://en.wikipedia.org/wiki/Out-of-core) and [distributed memory](https://en.wikipedia.org/wiki/Distributed_memory) situations where accessing non-contiguous data is extremely time-consuming. There are other multidimensional FFT algorithms that are distinct from the row-column algorithm, although all of them have O(*N* log *N*) complexity. Perhaps the simplest non-row-column FFT is the [vector-radix FFT algorithm](https://en.wikipedia.org/w/index.php?title=Vector-radix_FFT_algorithm&action=edit&redlink=1), which is a generalization of the ordinary Cooley–Tukey algorithm where one divides the transform dimensions by a vector  of radices at each step. (This may also have cache benefits.) The simplest case of vector-radix is where all of the radices are equal (e.g. vector-radix-2 divides *all* of the dimensions by two), but this is not necessary. Vector radix with only a single non-unit radix at a time, i.e. , is essentially a row-column algorithm. Other, more complicated, methods include polynomial transform algorithms due to Nussbaumer (1977), which view the transform in terms of convolutions and polynomial products. See Duhamel and Vetterli (1990) for more information and references.

**Expected Input:**

**Expected Output:**



**Conclusion:**

Hence we performed Fast Fourier Transform & implement computationally fast algorithms.

**Program: -**

#include<stdio.h>

#include<math.h>

#define PI 3.14159265

int main() {

int N = 0;

int n;

float x[10];

int k;

float x\_real[10], x\_img[10];

printf("Calculation DFT Coefficients\n");

printf("Enter the length of x(n):\n");

scanf("%d", &N);

printf("Enter the values of x(n):\n");

for (n = 0; n < N; n++) {

scanf("%f", &x[n]);

}

for(k=0;k<N;k++)

{x\_real[k]=0;

x\_img[k]=0;

for (n = 0; n < N; n++) {

x\_real[k]=(x[n]\*cos(2\*PI\*k\*n/(N)) ) + x\_real[k];

x\_img[k]=(x[n]\*sin(2\*PI\*k\*n/(N)) ) + x\_img[k];

}}

printf("The coefficients are: ");

for (k = 0; k < N; k++) {

printf("x[%d] = ( %f )-( %f i ) \n", k, x\_real[k], x\_img[k]);

}

return 0;

}

**EXPERIMENT NO. 7**

**7. Filtering of long Data Sequence**

**Aim:**

To perform filtering of Long Data Sequence using Overlap Add Method and Overlap Save Method.

**Objective:**

Develop a function to implement Fast Overlap Add and Fast Overlap Save Algorithm using FFT.

**Input Specifications:**

1. Length of long data sequence and signal values.

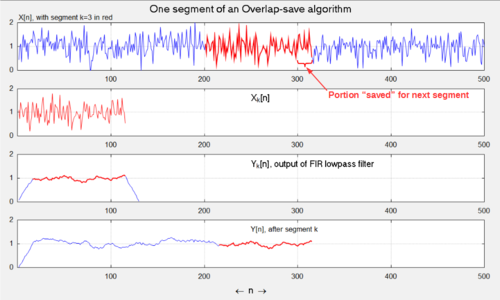
2. Length of impulse response M and coefficient values of h[n].

**Problem Definition:**

Find the output of a Discrete Time system using Fast Overlap Add Method OR Fast Overlap Save Method.

|  |  |  |
| --- | --- | --- |
| **Theory: -**  **Overlap–save** is the traditional name for an efficient way to evaluate the [discrete convolution](https://en.wikipedia.org/wiki/Convolution#Discrete_convolution) between a very long signal  and a [finite impulse response](https://en.wikipedia.org/wiki/Finite_impulse_response) (FIR) filter **:** |  |  |

where h[m]=0 for m outside the region [1, *M*].



**A sequence of 4 plots depicts one cycle of the Overlap-save convolution algorithm. The 1st plot is a long sequence of data to be processed with a lowpass FIR filter. The 2nd plot is one segment of the data to be processed in piecewise fashion. The 3rd plot is the filtered segment, with the usable portion colored red. The 4th plot shows the filtered segment appended to the output stream. The FIR filter is a boxcar lowpass with M=16 samples, the length of the segments is L=100 samples and the overlap is 15 samples.**

The concept is to compute short segments of *y*[*n*] of an arbitrary length *L*, and *k*, concatenate the segments together. Consider a segment that begins at *n* = *kL* + *M*, for y integerand define**:**

Then,for *kL* + *M*≤*n*≤*kL* + *L* + *M* − 1,and equivalently *M*≤*n* − *kL*≤*L* + *M* − 1, we can write**:**

The task is thereby reduced to computing *yk*[*n*], for *M*≤*n*≤*L* +*M* − 1. The process described above is illustrated in the accompanying figure.

Now note that if we periodically extend *xk*[*n*] with period *N*≥*L* + *M* − 1, according to**:**

the convolutionsandare equivalent in the region *M*≤*n*≤*L* + *M* − 1. So it is sufficient to compute the **N**-point [circular (or cyclic) convolution](https://en.wikipedia.org/wiki/Circular_convolution) of  within the region [1, *N*].The subregion [*M*, *L* + *M* − 1] is appended to the output stream, and the other values are discarded.The advantage is that the circular convolution can be computed very efficiently as follows, according to the [circular convolution theorem](https://en.wikipedia.org/wiki/Discrete_Fourier_transform#Circular_convolution_theorem_and_cross-correlation_theorem)**:**

* DFT and DFT−1 refer to the Discrete Fourier transform and inverse Discrete Fourier transform, respectively, evaluated over *N* discrete points, and
* *N* is customarily chosen to be an integer power-of-2, which optimizes the efficiency of the [FFT](https://en.wikipedia.org/wiki/Fast_Fourier_transform) algorithm.
* Optimal N is in the range [4M, 8M].
* Unlike the third graph in the figure above, depicting separate leading and trailing edge-effects, this method causes them to be overlapped and added. So they are discarded together.
* *This article uses common abstract notations, such asorin which it is understood that the functions should be thought of in their totality, rather than at specific instants  (see*[*Convolution#Notation*](https://en.wikipedia.org/wiki/Convolution#Notation)*)*
* In [signal processing](https://en.wikipedia.org/wiki/Signal_processing), the **overlap–add method (OA, OLA)** is an efficient way to evaluate the discrete [convolution](https://en.wikipedia.org/wiki/Convolution) of a very long signal  with a [finite impulse response](https://en.wikipedia.org/wiki/Finite_impulse_response) (FIR) filter **:**
* where *h*[*m*] = 0 for *m* outside the region [1, *M*].
* The concept is to divide the problem into multiple convolutions of *h*[*n*] with short segments of **:**
* where *L* is an arbitrary segment length. Then**:**
* and *y*[*n*] can be written as a sum of short convolutions**:**
* whereis zero outside the region [1, *L* + *M* − 1].And for any parameterit is equivalent to the -point [circular convolution](https://en.wikipedia.org/wiki/Circular_convolution) of  with in the region [1, *N*].

|  |  |  |
| --- | --- | --- |
|  |  |  |

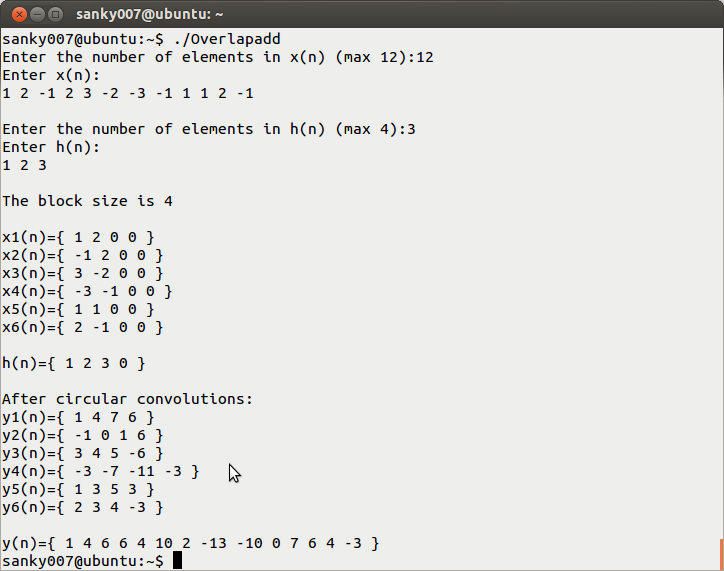
* where FFT and IFFT refer to the [fast Fourier transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform) and inverse fast Fourier transform, respectively, evaluated over  discrete points.
* When sequence *x*[*n*] is periodic, and *Nx* is the period, then *y*[*n*] is also periodic, with the same period.To compute one period of y[n], Algorithm 1 can first be used to convolve *h*[*n*] with just one period of *x*[*n*].In the region *M* ≤ *n* ≤ *Nx*,the resultant *y*[*n*] sequence is correct.And if the next *M* − 1 values are added to the first *M* − 1 values, then the region 1 ≤ *n* ≤ *Nx* will represent the desired convolution. The modified pseudocode is**:**

**Expected Input:**

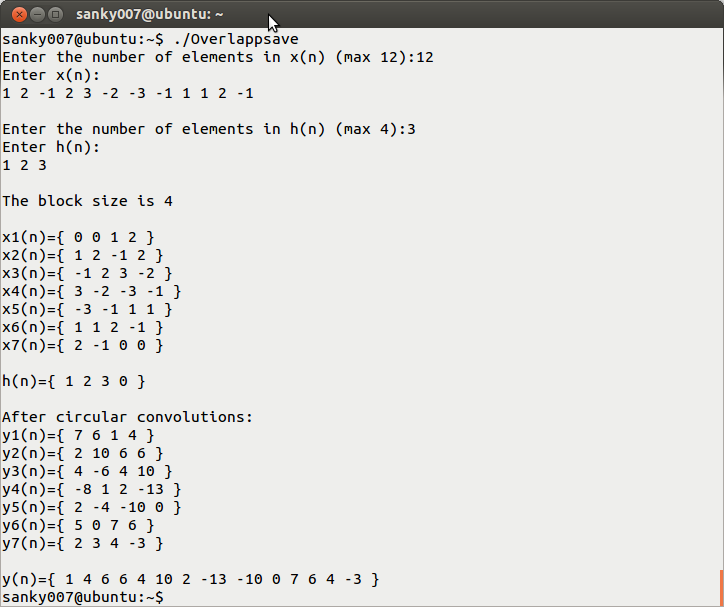
* **Algorithm 2** (*OA for circular convolution*)
* Evaluate Algorithm 1
* y(1:M-1) = y(1:M-1) + y(Nx+1:Nx+M-1)
* y = y(1:Nx)
* **end**

**Expected Output:**

OVERLAP ADD METHOD OUTPUT

****

**OVERLAP SAVE METHOD OUTPUT**

****

**Conclusion:**

Thus, performed filtering of Long Data Sequence using Overlap Add Method and Overlap Save Method.

**Program: -OverlapAdd**

#include<stdio.h>

int main(){

int x[12],h[4];

int c[4][4];

int i,j,k,n,m;

int xs[6][4],xa[6][4];

int y[14];

for(i=0;i<12;i++)

x[i]=0;

for(i=0;i<4;i++)

h[i]=0;

for(i=0;i<6;i++){

for(j=0;j<4;j++){

xs[i][j]=0;

}

}

printf("Enter the number of elements in x(n) (max 12):");

scanf("%d",&n);

printf("Enter x(n):\n");

for(i=0;i<n;i++)

scanf("%d",&x[i]);

/\*for(i=0;i<16;i++)

printf("%d ",x1[i]);\*/

printf("\nEnter the number of elements in h(n) (max 4):");

scanf("%d",&m);

printf("Enter h(n):\n");

for(i=0;i<m;i++)

scanf("%d",&h[i]);

printf("\nThe block size is 4\n\n");

k=0;

for(i=0;i<6;i++){

for(j=0;j<2;j++){

xs[i][j]=x[k++];

}

}

for(i=0;i<6;i++){

printf("x%d(n)={ ",i+1);

for(j=0;j<4;j++){

printf("%d ",xs[i][j]);

}

printf("}\n");

}

printf("\nh(n)={ ");

for(i=0;i<4;i++){

printf("%d ",h[i]);

}

printf("}\n");

for(i=0;i<4;i++){

for(j=0;j<4;j++){

c[j][i]=h[(j+4-i)%4];}

}

for(i=0;i<6;i++){

for(j=0;j<4;j++){

xa[i][j]=0;

for(k=0;k<4;k++){

xa[i][j]+=c[j][k]\*xs[i][k];

}}}

printf("\nAfter circular convolutions:\n");

for(i=0;i<6;i++){

printf("y%d(n)={ ",i+1);

for(j=0;j<4;j++){

printf("%d ",xa[i][j]);

}printf("}\n");

}i=0;

y[0]=xa[0][0];

y[1]=xa[0][1];

for(k=2;k<12;k++){

y[k++]=xa[i][2]+xa[i+1][0];

y[k]=xa[i][3]+xa[i+1][1];

i++;

}y[12]=xa[5][2];

y[13]=xa[5][3];printf("\ny(n)={ ");

for(i=0;i<14;i++){

printf("%d ",y[i]);

}

printf("}\n");

return 0;

**}**

**Program: -OverlapAdd**

#include<stdio.h>

int main(){

int x[12],x1[16],h[4];

int c[4][4];

int i,j,k,n,m;

int xs[7][4],xa[7][4];

int y[14];

for(i=0;i<12;i++)

x[i]=0;

for(i=0;i<4;i++)

h[i]=0;

printf("Enter the number of elements in x(n) (max 12):");

scanf("%d",&n);

printf("Enter x(n):\n");

for(i=0;i<n;i++)

scanf("%d",&x[i]);

x1[0]=0;

x1[1]=0;

x1[14]=0;

x1[15]=0;

for(i=2;i<2+n;i++)

x1[i]=x[i-2];

printf("\nEnter the number of elements in h(n) (max 4):");

scanf("%d",&m);

printf("Enter h(n):\n");

for(i=0;i<m;i++)

scanf("%d",&h[i]);

printf("\nThe block size is 4\n\n");

k=0;

for(i=0;i<7;i++){

for(j=0;j<4;j++){

xs[i][j]=x1[k++]; }

k=k-2;

}

for(i=0;i<7;i++){

printf("x%d(n)={ ",i+1);

for(j=0;j<4;j++){

printf("%d ",xs[i][j]);

}

printf("}\n");

}

printf("\nh(n)={ ");

for(i=0;i<4;i++){

printf("%d ",h[i]);

}

printf("}\n");

for(i=0;i<4;i++){

for(j=0;j<4;j++){

c[j][i]=h[(j+4-i)%4];

}

}

for(i=0;i<7;i++){

for(j=0;j<4;j++){

xa[i][j]=0;

for(k=0;k<4;k++){

xa[i][j]+=c[j][k]\*xs[i][k];

}

}

}printf("\nAfter circular convolutions:\n");

for(i=0;i<7;i++){

printf("y%d(n)={ ",i+1);

for(j=0;j<4;j++){

printf("%d ",xa[i][j]);

}printf("}\n");}

k=0;for(i=0;i<7;i++){

for(j=2;j<4;j++){

y[k++]=xa[i][j];}

}printf("\ny(n)={ ");

for(i=0;i<14;i++){

printf("%d ",y[i]);}

printf("}\n"); return 0;}

**EXPERIMENT NO. 8**

**8. Real Time Signal Processing**

**Aim:**

To perform real time signal processing using TMS320 Processor.

**Objective:**

Study real time signal processing.

**Input Specifications:**

1. Real Time Speech Signal

**Problem Definition:**

1) Capture the real time audio signal.

2) Filter it by convolving input signal with the impulse response of FIR filter using Fast Overlap Add filtering Algorithm OR Fast Overlao Save Filtering Algorithm.

3) Observe the quality of output signal.

**Theory:**

**INTRODUCTION TO DSP PROCESSORS**

A signal can be defined as a function that conveys information, generally about the state or behavior of a physical system. There are two basic types of signals viz. Analog (continuous time signals which are defined along a continuum of times) and Digital (discrete-time).Remarkably, under reasonable constraints; a continuous time signal can be adequately represented by samples, obtaining discrete time signals. Thus digital signal processing is an ideal choice for anyone who needs the performance advantage of digital manipulation along with today’s analog reality. Hence a processor which is designed to perform the special operations (digital manipulations) on the digital signal within very less time can be called as a Digital signal processor. The difference between a DSP processor, conventional microprocessor and a microcontroller are listed below.

**Microprocessor** or General Purpose Processor such as Intel xx86 or Motorola 680xx familyContains - only CPU

-No RAM

-No ROM

-No I/O ports

-No Timer

**Microcontroller** such as 8051 family

Contains - CPU

* RAM
* ROM

-I/O ports

* Timer &
* Interrupt circuitry

Some Micro Controllers also contain A/D, D/A and Flash Memory

**DSP Processors** such as Texas instruments and Analog Devices

Contains - CPU

-RAM ---ROM

* I/O ports
  + Timer

Optimized for – fast arithmetic

|  |  |  |
| --- | --- | --- |
| - | Extended precision | |
| - | Dual operand fetch | |
| - | Zero overhead loop | |
| - | Circular buffering | |
| The basic features of a DSP Processor are | | |
| **Feature** |  | **Use** |
| Fast-Multiply accumulate | | Most DSP algorithms, including filtering, transforms, etc. are multiplication- intensive |
|  | |  |
| Multiple – access memory | | Many data-intensive DSP operations require reading a program instruction and multiple data items |
|  | |  |
| Specialized addressing modes | | Efficient handling of data arrays and first-in, first-out buffers in memory |
|  | |  |
| Specialized program control | | Efficient control of loops for many iterative DSP algorithms. Fast interrupt handling for frequent I/O |
|  | |  |
| On-chip peripherals and I/O | | On-chip peripherals like A/D converters allow for small low cost system designs. Similarly I/O |
|  |  |  |



**ARCHITECTURE OF 6713 DSP PROCESSOR**

This chapter provides an overview of the architectural structure of the TMS320C67xx DSP, which comprises the central processing unit (CPU), memory, and on-chip peripherals. The C67xE DSPs use an advanced modified Harvard architecture that maximizes processing power with eight buses. Separate program and data spaces allow simultaneous access to program instructions and data, providing a high degree of parallelism. For example, three reads and one write can be performed in a single cycle. Instructions with parallel store and application-specific instructions fully utilize this architecture. In addition, data can be transferred between data and program spaces. Such Parallelism supports a powerful set of arithmetic, logic, and bit-manipulation operations that can all be performed in a single machine cycle. Also, the C67xx DSP includes the control mechanisms to manage interrupts, repeated operations, and function calling.

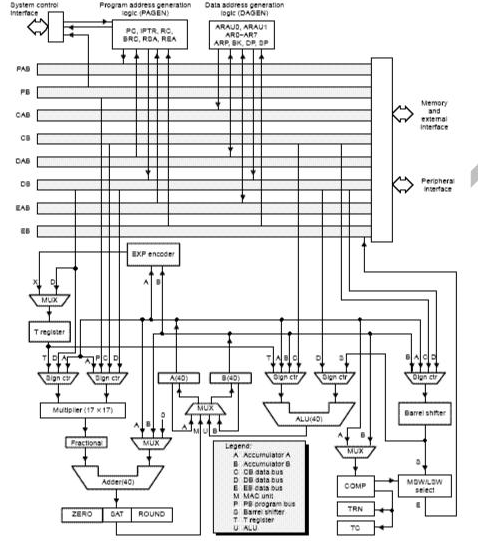


Fig 2 – 1 BLOCK DIAGRAM OF TMS 320VC 6713

**Bus Structure**

The C67xx DSP architecture is built around eight major 16-bit buses (four program/data buses and four address buses):

\_ The program bus (PB) carries the instruction code and immediate operands from program memory. \_ Three data buses (CB, DB, and EB) interconnect to various elements, such as the CPU, data address generation logic, program address generation logic, on-chip peripherals, and data memory. \_ The CB and DB carry the operands that are read from data memory.

\_ The EB carries the data to be written to memory.

\_ Four address buses (PAB, CAB, DAB, and EAB) carry the addresses needed for instruction execution.

The C67xx DSP can generate up to two data-memory addresses per cycle using the two auxiliary register arithmetic units (ARAU0 and ARAU1). The PB can carry data operands stored in program space (for instance, a coefficient table) to the multiplier and adder for multiply/accumulate operations or to a destination in data space for data move instructions (MVPD and READA). This capability, in conjunction with the feature of dual-operand read, supports the execution of single-cycle, 3-operand instructions such as the FIRS instruction. The C67xx DSP also has an on-chip bidirectional bus for accessing on-chip peripherals. This bus is connected to DB and EB through the bus exchanger in the CPU interface. Accesses that use this bus can require two or more cycles for reads and writes, depending on the peripheral’s structure.

**Central Processing Unit (CPU)**

The CPU is common to all C67xE devices. The C67x CPU contains:

\_ 40-bit arithmetic logic unit (ALU)

\_ Two 40-bit accumulators

\_ Barrel shifter

\_ 17 × 17-bit multiplier

\_ 40-bit adder

\_ Compare, select, and store unit (CSSU)

\_ Data address generation unit

\_ Program address generation unit

**Arithmetic Logic Unit (ALU)**

The C67x DSP performs 2s-complement arithmetic with a 40-bit arithmetic logic unit (ALU) and two 40-bit accumulators (accumulators A and B). The ALU can also perform Boolean operations. The ALU uses these inputs:

\_ 16-bit immediate value

\_ 16-bit word from data memory

\_ 16-bit value in the temporary register, T

\_ Two 16-bit words from data memory

\_ 32-bit word from data memory

\_ 40-bit word from either accumulator

The ALU can also function as two 16-bit ALUs and perform two 16-bit operations simultaneously.

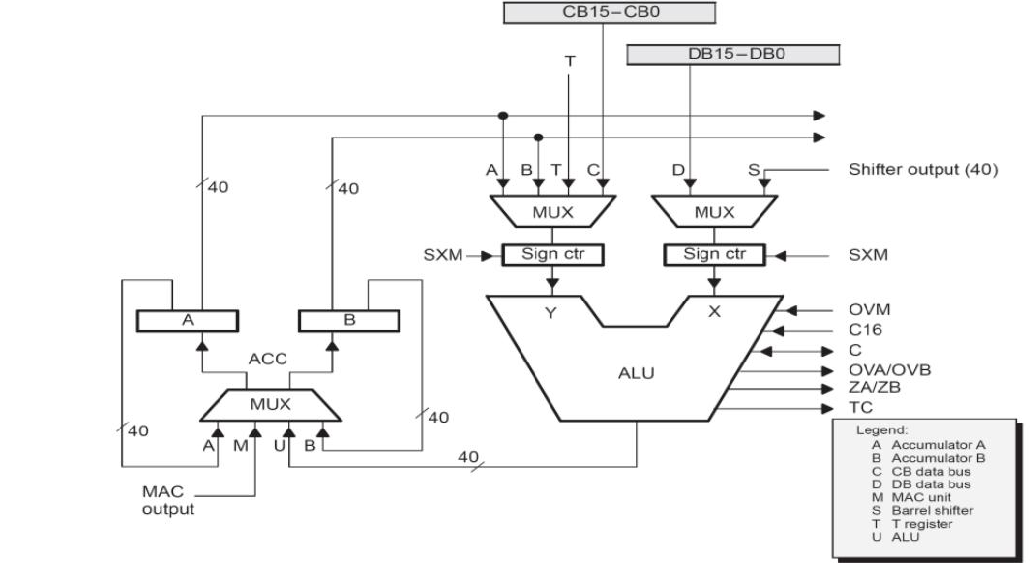


Fig 2 – 2 ALU UNIT

**Accumulators**

Accumulators A and B store the output from the ALU or the multiplier/adder block. They can also provide a second input to the ALU; accumulator A can be an input to the multiplier/adder. Each accumulator is divided into three parts:

\_ Guard bits (bits 39–32)

\_ High-order word (bits 31–16)

\_ Low-order word (bits 15–0)

Instructions are provided for storing the guard bits, for storing the high- and the low-order accumulator words in data memory, and for transferring 32-bit accumulator words in or out of data memory. Also, either of the accumulators can be used as temporary storage for the other.

**Barrel Shifter**

The C67x DSP barrel shifter has a 40-bit input connected to the accumulators or to data memory (using CB or DB), and a 40-bit output connected to the ALU or to data memory (using EB). The barrel shifter can produce a left shift of 0 to 31 bits and a right shift of 0 to 16 bits on the input data. The shift requirements are defined in the shift count field of the instruction, the shift count field (ASM) of status register ST1, or in temporary register T (when it is designated as a shift count register).The barrel shifter and the exponent encoder normalize the values in an accumulator in a single cycle. The LSBs of the output are filled with 0s, and the MSBs can be either zero filled or sign extended, depending on the state of the sign-extension mode bit (SXM) in ST1. Additional shift capabilities enable the processor to perform numerical scaling, bit extraction, extended arithmetic,and overflow prevention operations.

**Multiplier/Adder Unit**

The multiplier/adder unit performs 17 \_ 17-bit 2s-complement multiplication with a 40-bit addition in a single instruction cycle. The multiplier/adder block consists of several elements: a multiplier, an adder, signed/unsigned input control logic, fractional control logic, a zero detector, a rounder (2s complement), overflow/saturation logic, and a 16-bit temporary storage register (T). The multiplier has two inputs: one input is selected from T, a data-memory operand, or accumulator A; the other is selected from program memory, data memory, accumulator A, or an immediate value. The fast, on-chip multiplier allows the C54x DSP to perform operations efficiently such as convolution, correlation, and filtering. In addition, the multiplier and ALU together execute multiply/accumulate (MAC) computations and ALU operations in parallel in a single instruction cycle. This function is used in determining the Euclidian distance and in implementing symmetrical and LMS filters, which are required for complex DSP algorithms. See section 4.5, Multiplier/Adder Unit, on page 4-19, for more details about the multiplier/adder unit. These are the some of the important parts of the processor and you are instructed to go through the detailed architecture once which helps you in developing the optimized code for the required application.

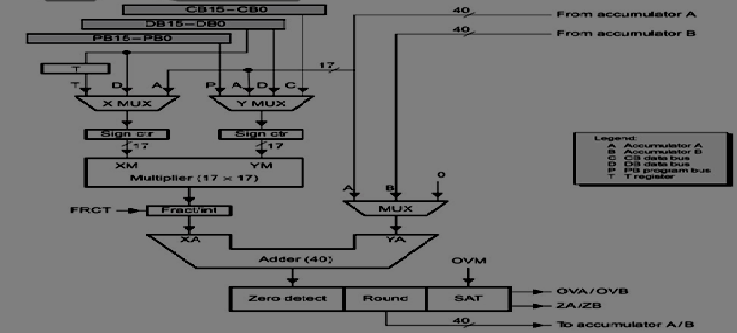


Fig 2 – 3 MULTIPLIER/ADDER UNIT

**Conclusion:** The architecture of DSP chips-TMS 320c 5x/6x is studied successfully.

**EXPERIMENT NO. 9**

**9. Application of Digital Signal Processing**

**Aim:**

To implement any Signal Processing operation on one dimensional signal.

**Objective:**

To develop application of signal processing.

**Input Specifications:**

One dimensional signal.

**Theory:**

**Rules**:

1. Number of students in one Group : min - 2 max -3

2. Decide one DSP application of your choice. Collect the information related to the application from the published granted patents. Download the related published papers from the standard refereed journals and conferences.

3. Develop a block diagram of the proposed system and flowchart of proposed system algorithm, implement it using Scilab/C, C++ language and obtain the appropriate results.

4. Prepare the three to four pages report on the mini project in IEEE paper format.

Report should include Abstract, Introduction, Related Theory, Proposed System Design/Algorithm, Experimentation & Result Analysis, Conclusion, and References.

**Conclusion:**

Thus we implemented any Signal Processing operation on one dimensional signal

**EXPERIMENT NO. 10**

**Aim:** Study of Latex Tool

**Theory:**

LaTeX styled as LaTeX, and a shortening of Lamport TeX is a word processor and a document markup language. It is distinguished from typical word processors such as Microsoft Word, LibreOffice Writer and Apple Pages in that the writer uses plain text as opposed to formatted text, relying on markup tagging conventions to define the general structure of a document (such as article, book, and letter), to stylise text throughout a document (such as bold and italic), and to add citations and cross-referencing. A TeX distribution such as TeXlive or MikTeX is used to produce an output file (such as PDF or DVI) suitable for printing or digital distribution.

LaTeX is used for the communication and publication of scientific documents in many fields, including mathematics, physics, computer science, statistics, economics, and political science not in citation given. It also has a prominent role in the preparation and publication of books and articles that contain complex multilingual materials, such as Sanskrit and Arabic[citation needed]. LaTeX uses the TeX typesetting program for formatting its output, and is itself written in the TeX macro language.

LaTeX is widely used in academia.[not in citation given] LaTeX can be used as a standalone document preparation system, or as an intermediate format. In the latter role, for example, it is often used as part of a pipeline for translating DocBook and other XML-based formats to PDF. The typesetting system offers programmable desktop publishing features and extensive facilities for automating most aspects of typesetting and desktop publishing, including numbering and cross-referencing of tables and figures, chapter and section headings, the inclusion of graphics, page layout, indexing and bibliographies.

Like TeX, LaTeX started as a writing tool for mathematicians and computer scientists, but from early in its development it has also been taken up by scholars who needed to write documents that include complex math expressions or non-Latin scripts, such as Arabic, Sanskrit and Chinese[citation needed].

LaTeX is intended to provide a high-level language that accesses the power of TeX in an easier way for writers. In short, TeX handles the layout side, while LaTeX handles the content side for document processing. LaTeX comprises a collection of TeX macros and a program to process LaTeX documents. Because the plain TeX formatting commands are elementary, it provides authors with ready-made commands for formatting and layout requirements such as chapter headings, footnotes, cross-references and bibliographies.

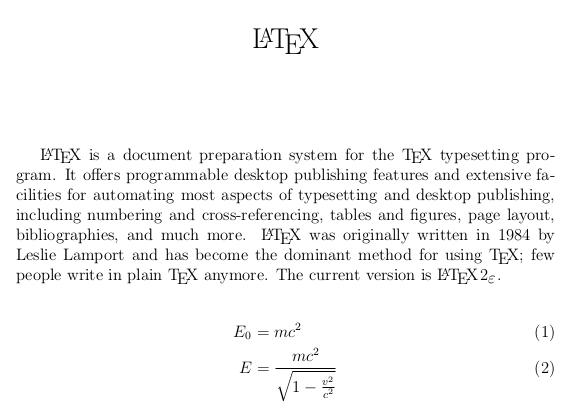
LaTeX was originally written in the early 1980s by Leslie Lamport at SRI International. The current version is LaTeX2e (styled as LaTeX2ε). LaTeX is free software and is distributed under the LaTeX Project Public License (LPPL).

**Typesetting system**

LaTeX follows the design philosophy of separating presentation from content, so that authors can focus on the content of what they are writing without attending simultaneously to its visual appearance. In preparing a LaTeX document, the author specifies the logical structure using simple, familiar concepts such as chapter, section, table, figure, etc., and lets the LaTeX system worry about the formatting and layout of these structures. It therefore encourages the separation of layout from content while still allowing manual typesetting adjustments where needed. This concept is similar to the mechanism by which many word processors allow styles to be defined globally for an entire document or the use of Cascading Style Sheets to style HTML. The LaTeX system is a markup language that also handles typesetting and rendering.

LaTeX can be arbitrarily extended by using the underlying macro language to develop custom formats. Such macros are often collected into packages, which are available to address special formatting issues such as complicated mathematical content or graphics. Indeed, in the example below, the align environment is provided by the amsmath package.

In order to create a document in LaTeX, you first write a file, say foobar.tex, using your preferred text editor. Then you give your foobar.tex file as input to the TeX program (with the LaTeX macros loaded), and TeX writes out a file suitable for viewing onscreen or printing.This write-format-preview cycle is one of the chief ways in which working with LaTeX differs from what-you-see-is-what-you-get word-processing. It is similar to the code-compile-execute cycle familiar to computer programmers. Today, many LaTeX-aware editing programs make this cycle a simple matter of pressing a single key, while showing the output preview on the screen beside the input window. Some online Latex editors automatically refresh the preview.



**Pronouncing and writing "LaTeX"**

LaTeX is usually pronounced /ˈleɪtɛk/ or /ˈlɑːtɛk/ in English (that is, not with the /ks/ pronunciation English speakers normally associate with X, but with a /k/).

The characters T, E, X in the name come from capital Greek letters tau, epsilon, and chi, as the name of TeX derives from the Greek: τέχνη (skill, art, technique); for this reason, TeX's creator Donald Knuth promotes a pronunciation of /tɛx/ (tekh) (that is, with a voiceless velar fricative as in Modern Greek, similar to the ch in loch). Lamport, on the other hand, has said he does not favor or discourage any pronunciation for LaTeX.[citation needed]

The name is traditionally printed in running text with a special typographical logo: LaTeX. In media where the logo cannot be precisely reproduced in running text, the word is typically given the unique capitalization LaTeX. The TeX, LaTeX and XeTeX logos can be rendered via pure CSS and XHTML for use in graphical web browsers following the specifications of the internal \LaTeX macro.

**Licensing**

LaTeX is typically distributed along with plain TeX. It is distributed under a free software license, the LaTeX Project Public License (LPPL). The LPPL is not compatible with the GNU General Public License, as it requires that modified files must be clearly differentiable from their originals (usually by changing the filename); this was done to ensure that files that depend on other files will produce the expected behavior and avoid dependency hell. The LPPL is DFSG compliant as of version 1.3. As free software, LaTeX is available on most operating systems including UNIX (Solaris, HP-UX, AIX), BSD (FreeBSD, Mac OS X, NetBSD, OpenBSD), Linux (Red Hat, Debian GNU/Linux, Arch, Gentoo), Microsoft Windows (9x, XP, Vista, 7, 8), DOS, RISC OS, AmigaOS and Plan9.

**Conclusion:**

Thus we studied the Latex tool.

**PVPP-DM-COMP-FF-7 Revision 00 9/3/17**