**ASSIGNMENT NO.B18**

**TITLE:** To generate a Square wave of programmable frequency and generate Pole-ZeroDiagram using multi core programming.

**PROBLEM STATEMENT:**

Write a C++/ Python program to generate a Square wave of programmable frequency. Write a function to generate Pole-Zero Diagram using multi core programming.

**OBJECTIVE:**

1. To learn the basic concept of how to plot pole and zero.

2. To learn the about basic concept of multi core programming.

**PRIREQUISITES:**

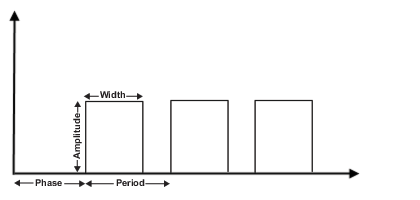
**Software**: Linux, open MP

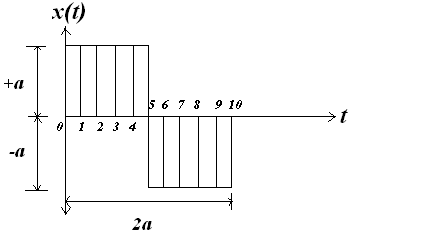
**Hardware:** Quad coreprocessor/ Dual core processor.

**THEORY:**

A square wave is a [non-sinusoidal periodic waveform](http://en.wikipedia.org/wiki/Non-sinusoidal_waveform) (which can be represented as an infinite summation of sinusoidal waves), in which the amplitude alternates at a steady [frequency](http://en.wikipedia.org/wiki/Frequency) between fixed minimum and maximum values, with the same duration at minimum and maximum.

Square waves are often encountered in [electronics](http://en.wikipedia.org/wiki/Electronics) and [signal processing](http://en.wikipedia.org/wiki/Signal_processing). Square waves are universally encountered in [digital](http://en.wikipedia.org/wiki/Digital_data) switching circuits and are naturally generated by binary (two-level) logic devices. They are used as timing references or "[clock signals](http://en.wikipedia.org/wiki/Clock_signal)", because their fast transitions are suitable for triggering [synchronous logic](http://en.wikipedia.org/wiki/Synchronous_logic) circuits at precisely determined intervals.





If the sampling frequency is '', then for a wave of amplitude 'a' the number of samples is given by 2a/

For square wave the only one thing require is the implement counter then switch the sign and reset the counter every time the half period is passed

Where n is the number of half period time that is passed

**Algorithm:**

|  |
| --- |
| Step 1: Enter frequency F and Sampling Frequency Fs.  Step 2:  k=0;  do  {  for(n=(k\*Fs)/(2\*F);n<((k+1)\*Fs)/(2\*F);n++)  { // positive half cycle of square wave  x[n]=A;  if(n>MAX)break;  }  for(n=((k+1)\*Fs)/(2\*F);n<((k+2)\*Fs)/(2\*F);n++)  { // negative half cycle of square wave  x[n] = -A;  if(n>MAX) break;  }  k = k+2; // this count is modified for next cycle  } while(n<MAX);  break;  Step 3: Plotting this wave by using Graphics function. |

**Draw a pole-zero plot for a given system H(Z) expressed as rational function.**

Consider a LTI system with input X(n) and output Y(n).

In terms of z-Transform we can define system transfer function H(z) as the ratio of o/p & i/p

H(Z) = Y(Z)/X(Z)

We can express z-Transform as the ratio of 2 polynomials called as rational z-transform.

X(Z ) = N(Z)/D(Z)

* Poles are the values of Z for which X(Z) is infinity.
* Zeros are the values of Z for which X(Z) is 0.

The equation can be written as,

Rearranging the equation,

Where Z1, Z2 ………………. Zm are zeros.

& b1, b2 …………….... bn are poles.

**EXAMPLE:**

Determine pole-zero plot for the system.

Multiply &divide by Z^2.

Therefore,

Here, X(Z) is 0 for Z=0 & Z=-3.

X(Z) is ∞ for Z=-1,-2.

Therefore,

Poles = -1,-2.

Zeros = 0,-3.

**Algorithm:-**

|  |
| --- |
| Step1:  Accept equation from users.  Step2:  Accept second equation from user.  Step3:  Create thread and run on independent core for to calculate poles.  Step4:  Calculate poles.  Step5:  Create other thread and run on other core for to calculate zero.  Step6:  Calculate zeroes.  Step7:  Display pole and zero. |

**OpenMP**

OpenMP (Open Multi-Processing) is an [API](http://en.wikipedia.org/wiki/Application_programming_interface) that supports multi-platform [shared memory](http://en.wikipedia.org/wiki/Shared_memory_architecture) [multiprocessing](http://en.wikipedia.org/wiki/Multiprocessing) programming. OpenMP is an implementation of [multithreading](http://en.wikipedia.org/wiki/Thread_%28computer_science%29), a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the [runtime environment](http://en.wikipedia.org/wiki/Runtime_environment) allocating threads to different processors. OpenMP uses a [portable](http://en.wikipedia.org/wiki/Software_portability), scalable model that gives [programmers](http://en.wikipedia.org/wiki/Programmer) a simple and flexible interface for developing parallel applications for platforms. An application built with the hybrid model of [parallel programming](http://en.wikipedia.org/wiki/Parallel_programming) can run on a [computer cluster](http://en.wikipedia.org/wiki/Computer_cluster) using both OpenMP and [Message Passing Interface](http://en.wikipedia.org/wiki/Message_Passing_Interface) (MPI), or more transparently through the use of OpenMP extensionsfor non-shared memory systems. OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an id attached to it which can be obtained using a function (called omp\_get\_thread\_num()). The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions.

**Thread Based Parallelism:**

1. OpenMP programs accomplish parallelism exclusively through the use of threads.
2. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
3. Threads exist within the resources of a single process. Without the process, they cease to exist.
4. Typically, the number of threads matches the number of machine processors/cores. However, the actual use of threads is up to the application.

**Explicit Parallelism:**

1. OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
2. Parallelization can be as simple as taking a serial program and inserting compiler directives.
3. Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

**Threads**:-The number of threads in a parallel region is determined by the following factors, in order of precedence:

* 1. Evaluation of the **IF** clause
  2. Setting of the **NUM\_THREADS** clause
  3. Use of the **omp\_set\_num\_threads()** library function
  4. Setting of the **OMP\_NUM\_THREADS** environment variable
  5. Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).
* Threads are numbered from 0 (master thread) to N-1

**Dynamic Threads:**

* Use the **omp\_get\_dynamic()** library function to determine if dynamic threads are enabled.
* If supported, the two methods available for enabling dynamic threads are:
  1. The **omp\_set\_dynamic()** library routine
  2. Setting of the **OMP\_DYNAMIC** environment variable to TRUE

**Conclusion: -** In this way we generate a Square wave of programmable frequency and calculate Pole-Zerousing multi core programming.

#include<iostream>

#include<omp.h>

#include<math.h>

//#include<libgraph.h>

using namespace std;

int neg1=0,neg2=0,a=-1,b=-1;

int main(){

float A1,B1,C1;

float A2,B2,C2;

double pole1,zero1,pole2,zero2;

cout<<"\n Enter First Equation :";

cout<<"\n Enter 'a' :";

cin>>A1;

cout<<"\n Enter 'b' :";

cin>>B1;

cout<<"\n Enter 'c' :";

cin>>C1;

cout<<"\n Enter Second Equation :";

cout<<"\n Enter 'a' :";

cin>>A2;

cout<<"\n Enter 'b' :";

cin>>B2;

cout<<"\n Enter 'c' :";

cin>>C2;

float temp1,temp2,numerator1,numerator2;

#pragma omp parallel

{

int threads= omp\_get\_num\_threads();

// Zero calculation

if(omp\_get\_thread\_num()==0 && a<0)

{

a++;

cout<<"\n Thread A is Caculating Zero. (TID: "<<omp\_get\_thread\_num()<<" )";

temp1=(B1\*B1)-(4\*A1\*C1);

if(temp1<0)

{

neg1=1;

temp1=-temp1;

}

temp1=sqrt(temp1);

numerator1=temp1;

if(!neg1){

zero1=(-B1+ numerator1)/2\*A1;

zero2=(-B1- numerator1)/2\*A1;

}

else

{

temp1=2\*A1;

}

}

// Pole calculation

if(omp\_get\_thread\_num()==1 && b<0)

{

b++;

cout<<"\n Thread B is Caculating Pole. (TID: "<<omp\_get\_thread\_num()<<" )";

temp2=(B2\*B2)-(4\*A2\*C2);

if(temp2<0)

{

neg2=1;

temp2=-temp2;

}

temp2=sqrt(temp2);

numerator2=temp2;

if(!neg2){

pole1=(-B2+ numerator2)/2\*A2;

pole2=(-B2- numerator2)/2\*A2;

}

else

{

temp2=2\*A2;

}

}

}

if(!neg1){

cout<<"\n Zero 1 is :"<<zero1;

cout<<"\n Zero 2 is :"<<zero2;

}

else{

cout<<"\n Zero 1 is : "<<-B1/temp1<<" +i "<<numerator1/temp1;

cout<<"\n Zero 2 is : "<<-B1/temp1<<" -i "<<numerator1/temp1<<endl;

}

if(!neg2){

cout<<"\n Pole 1 is :"<<pole1;

cout<<"\n Pole 2 is :"<<pole2<<endl;

}

else

{

cout<<"\n Pole 1 is : "<<-B2/temp2<<" +i "<<numerator2/temp2;

cout<<"\n Pole 2 is : "<<-B2/temp2<<" -i "<<numerator2/temp2<<endl;

}

return 0;

}

g++ -fopenmp pz.cpp -o pz

./pz

#include<stdio.h>

#include<conio.h>

#include<math.h>

#include<stdlib.h>

#include<graphics.h>

#define MAX 500

#define A 1.0

void main()

{

float x[700],F,Fs,n,Y;

int gd,gm,X,option,k;

char ch,ch1;

clrscr();

l1:

do

{

clrscr();

printf("Generation of discrete time signals \n");

printf("\nEnter the frequency of analog signal F = ");

scanf("%f",&F); //frequency of the signal

printf("Enter the sampling frequency Fs = ");

scanf("%f",&Fs); //sampling frequency

printf("\nEnter your choice");

printf("\nsine wave (Enter 1)"); //sine wave

printf("\nsquare wave (Enter 2)"); //square wave

printf("\n Exit (Enter 3)");

scanf("%d",&option);

// i = 640; //640 samples of the signal will be generated

// A = 1.0; //maximum amplitude of the signals is 1

switch(option)

{

case 1 : for(n = 0; n < MAX; n++) // sine wave

x[n] =A\*sin(2\*M\_PI\*F\*(n/Fs));

break;

case 2 :

k=0; // square wave

do

{

for(n=(k\*Fs)/(2\*F);n<((k+1)\*Fs)/(2\*F);n++)

{ // positive half cycle of aquare wave

x[n]=A;

if(n>MAX)break;

}

for(n=((k+1)\*Fs)/(2\*F);n<((k+2)\*Fs)/(2\*F);n++)

{ // negative half cycle of aquare wave

x[n] = -A;

if(n>MAX) break;

}

k = k+2; // this count is modified for next cycle

} while(n<MAX);

break;

case 3:exit(0);

break;

deault:printf("\n oops .. wrong option");

}

//----- next part of the program displays the generated Signal

Y = X = 0;

gd =gm= DETECT;

// initialize screen in graphics mode

initgraph(&gd,&gm,"c:\\tc\\bgi");

outtextxy(130,20,"Standard signals");

line(1,50,1,350);

line(1,200,550,200);

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

{

// this loop displays 640 samples of discrete time signal

Y = 200 - x[n]\*100; //scaling of x(n) for proper display

putpixel(X,Y,WHITE); //x(n) is displayed as putting pixels

X++;

}

getch();

closegraph();

fflush(stdin);

printf("\n \n do u want to continue...");

printf("\n press y if yes, otherwise press any other key..");

scanf("%c",&ch);

}

while(ch=='y');

}