ELECTRONIC DESIGN PROJECT (EE318), 2005

DIRECTION FINDING SYSTEM

GROUP: B06
ADNAN RAJA (02007013)
SHRIRAM SHIVARAMAN (02007014)
AASHWIT MAHAJAN (02007010)

INTRODUCTION

The objective of the Electronic Design Project is to determine the Direction of Arrival (DoA) of unknown signals through an array of antennas, as a practical implementation of the MUltiple SIgnal Classification (MUSIC) algorithm.

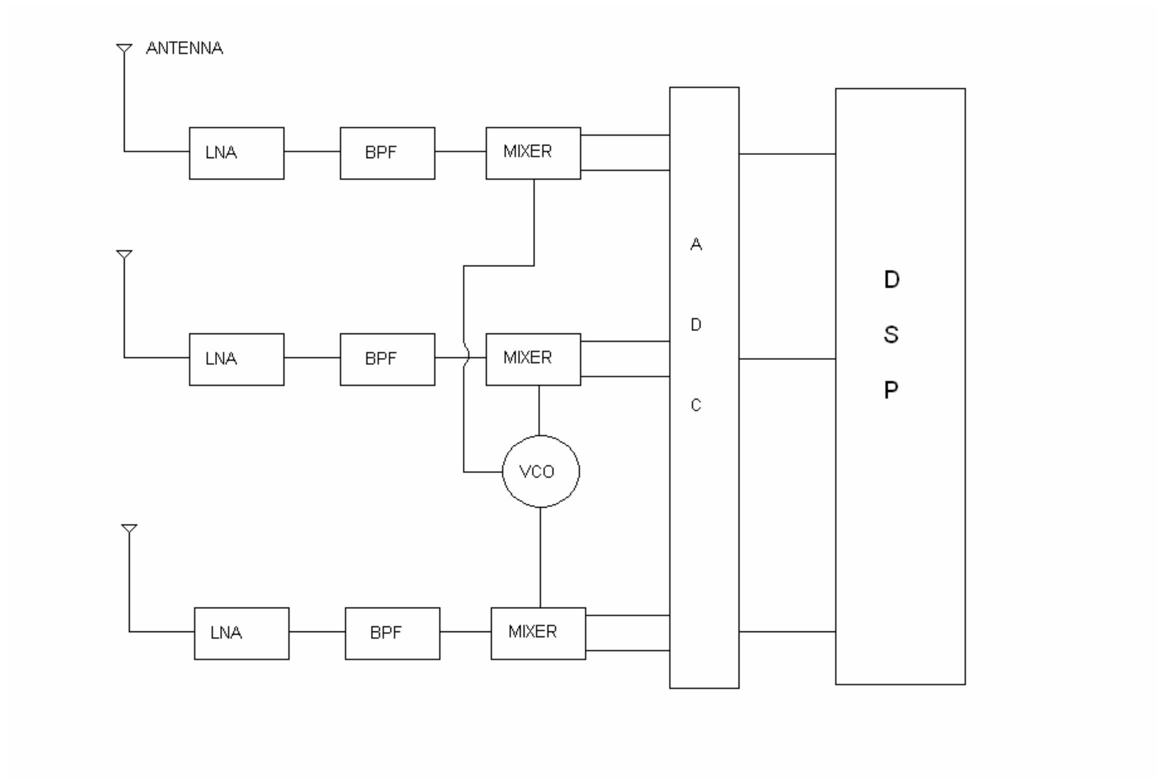
Direction Finding (DF) Systems have wide applications in various military and civilian operations, such as surveillance, reconnaissance, rescue work etc. They can be used to track the direction of any incoming RF signal. With the spurt in wireless communication, they can also be used to optimize the performance of any wireless network, in addition to the common applications like vehicle tracking.

Traditionally, the DF systems have been implemented using beam-forming methods for an array of antenna elements. However, the implementation being considered in this project is based on the MUSIC algorithm. This does not involve changing the phase or the amplitude response of any antenna element to direct the beam in a certain direction. It is based on the method of Triangulation, for estimating the direction of the incoming signal. An antenna array with three elements is designed, such that the angle of the incoming signal is different for the three antenna elements. Therefore, the signal received by each antenna element has a phase difference with respect to the signal received by the other antenna elements. Using this phase difference information, the direction can be estimated using the MUSIC algorithm.

Although the larger aim of the DF System is to estimate the Direction of Arrival of signals emitted by multiple sources in a broad range, at present, for the purpose of principle demonstration, a single source emitting a single known frequency signal is being considered. However, the system design can be easily modified to make it compatible for a broad range of signals.

BLOCK DIAGRAM

The complete system can be described by the following block diagram:



The RF signal is received by three different antenna elements. The antennas are arranged in the uniform circular array configuration. In our case we have three antennas placed at the three vertices of an equilateral triangle. Due to differences in their location with respect to the transmitter, the signals received by each of the antennas will be phase shifted from each other. Each of these signals is passed through a Low Noise Filter (LNA) and a Band Pass Filter (BPF) to filter out unwanted signals. They are then down converted to low frequency (near DC) using a Voltage Control Oscillator (VCO). Notice that the VCO is common for all the three modules. This is to prevent any loss of phase information due to lack of synchronism between the three VCO's if more than one VCO is used. The mixer gives both the in phase and the quadrature phase components of the input RF signal as output. These are then sampled by an ADC, and given to the DSP to perform further computations to estimate the DoA. The DSP implements the MUSIC algorithm to estimate the direction of arrival.

For the present system, since the input signal frequency is assumed to be known, hence the BPF is designed for the particular frequency.

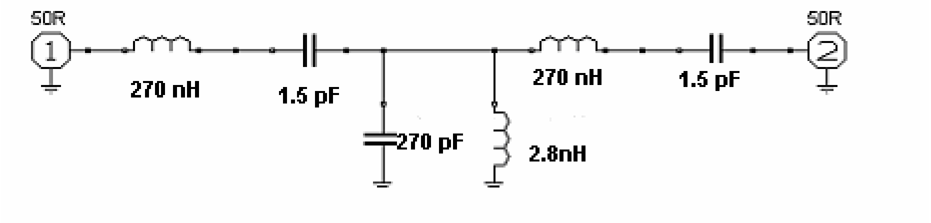
For the sake of further discussion, the system is broadly divided into the RF part for obtaining the phase information, and the software part for processing the phase information and thus estimating the direction of arrival.

THE RF PART

The DF system is being implemented for a fixed given frequency of 180 MHz. The frequency has been so chosen so that the FM band which ranges from 88MHz to 108 MHz, does not interfere with the DF System, and the principle can be demonstrated.

The antenna being used for the system is a monopole antenna with the center frequency of 180 MHz. BGA430 has been used as the Low Noise Amplifier for the system. BGA430 is a broad band low noise amplifier, which is internally matched to 50Ω. It provides a gain of 32dB at the operating frequency of 180MHz. Also the Noise Figure at this frequency is 2.3dB. The Noise Figure is an indicator of the noise being added by a component. It is computed by subtracting the gain of the device from the ratio of noise at the output to that at the input. BGA430 requires a Vcc of 5V for normal operation.

The BPF is a Chebyshev Filter with the center frequency of 180MHz and a bandwidth of 15MHz. The input and the output of the filter is matched to 50Ω. The circuit diagram of the filter is as follows:

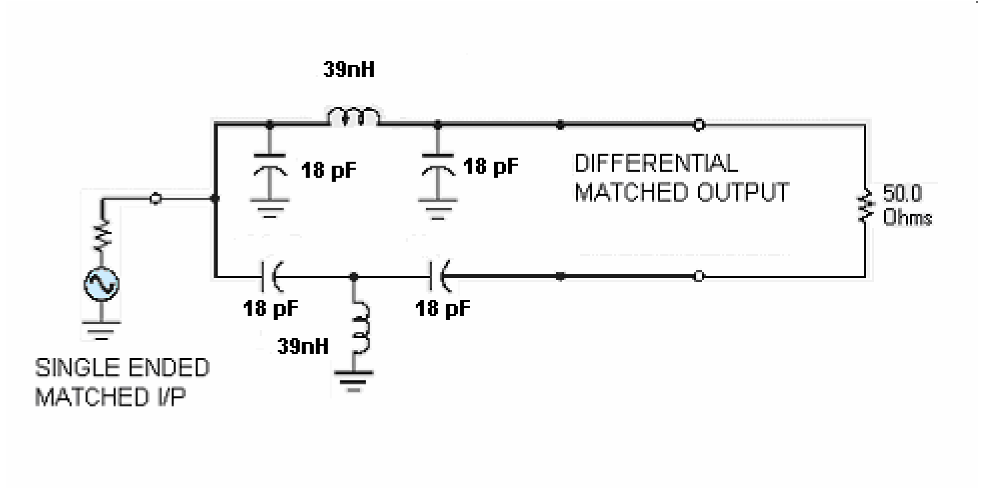


The MAX2309 IC is used as a quadrature mixer as well as a VCO. It has an inbuilt VCO circuit consisting of a frequency multiplier, and a PLL. The input required is a reference frequency signal with the minimum amplitude of 0.2 V peak to peak. The frequency of this reference signal is divided by a number stored in the R register of the MAX2309, and this resultant signal locks on to the input signal divided by 2M, where M is another register in MAX2309. Hence, for appropriate values of M and R, a factor of the reference signal is matched with the input signal, to produce a low frequency (near DC) output. The registers M and R can be programmed by a microcontroller or a DSP.

The reference frequency signal to MAX2309 is provided through a Temperature Compensated Crystal Oscillator (TCXO), GTXO-71T. The output is a sinusoidal signal of frequency 16.0MHz, 0.8V peak to peak. Hence, it is compatible with the input requirements for the MAX2309.

In addition, the MAX2309 has a gain control feature. For a Vcc of 3.3V the gain control voltage can be varied in the range -0.3 V to 3.6 V, and thus, the gain can be varied from -60dB to 60dB.

Also, the input to the MAX2309 required a design of a Balun for impedance matching purposes, as well as for converting the single ended incoming signal to a differential input, as required by MAX2309. The Balun design for the center frequency of 180MHz is as follows:



Clearly, the frequency being matched by the MAX2309 can be varied over a broad band by programming the M and R registers accordingly, with the help of the DSP.

Hence, for the RF part, the MAX2309 gain control is controlled with the help of a potentiometer. The register values, for matching the input signals with the reference signal, are controlled by the DSP through a 3-channel serial bus

For the present case, since the frequency of the incoming signal is already known, there is no feedback from the RF part to the DSP indicating whether the frequency being matched is the correct frequency. However, for the general case, this can also be realized by calibrating the gain control in such a manner that until a particular threshold level of voltage is received at the outputs of the MAX2309, the DSP keeps scanning in the whole band.

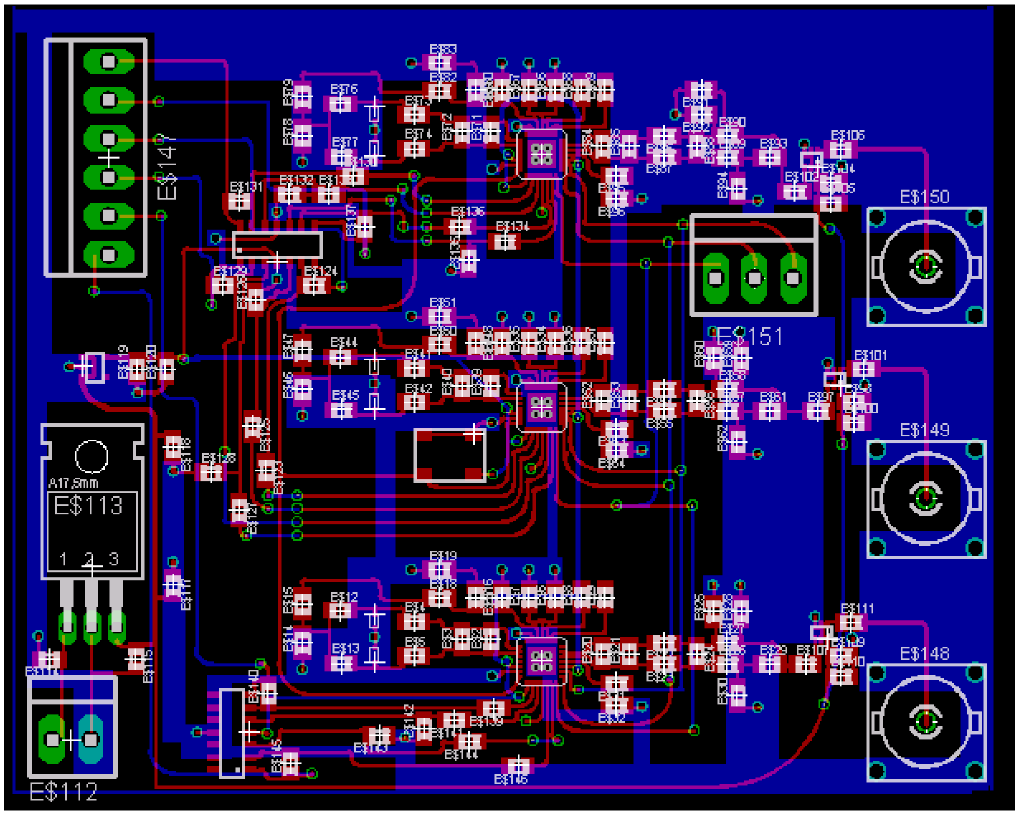
The outputs of the MAX2309 are differential. In order to convert them to single ended outputs, LM324 operational amplifier is used in differential amplifier configuration. LM324 operates with a single ended supply of +5V.

BGA430, LM324, and the TCXO operate with Vcc of 5V. Hence, 7805 Voltage regulator is used to supply regulated power to these components. The MAX2309 requires a Vcc of 2.8V. So, the TPS76928 voltage regulator is used for a regulated power supply to the mixer.

The outputs of the mixer are the in phase and the quadrature phase components of the incoming signal with respect to the reference frequency signal. These signals are then given to the ADC which samples them, and then further to the DSP, for estimating the Direction of Arrival. This is explained in detail in the section on software part.

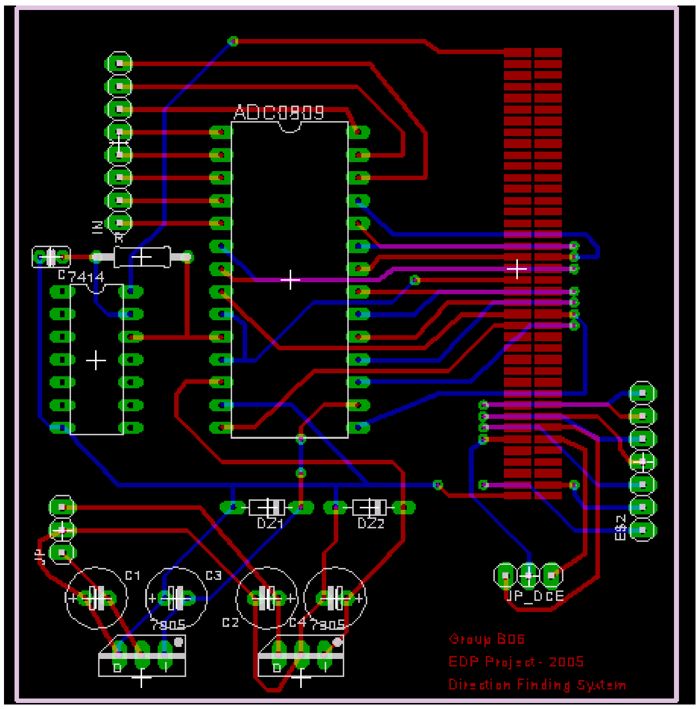
THE RF PCB

A 10.0cm*8.0cm PCB for the RF part has been designed with the above mentioned components and modules. The minimum distance between two neighboring conducting strips has been kept at 0.2 mm. The board file for the RF PCB is as below:



ADC+DSP HARDWARE

The outputs from the RF part are given to the ADC 0809, which in turn is interfaced with the TMS320C6711 DSP Kit. The board of the ADC 0809 PCB is as under:



THE SOFTWARE PART

The code for the sampling done by the ADC is as follows:

```
#include <stdio.h>

#include <c6x.h>

#include "c6211dsk.h"

#define port 0xA0000000


/*******************************************************************************

* Function prototypes

*******************************************************************************/

void delay_usec(int);

unsigned int sample(short channel);
```

```
void covariance(int re[3][100],int im[3][100]);

int rS[3][3],iS[3][3];

int main()
{

  short count;
  short channel;
  short t;
  int re[3][100];
  int im[3][100];

  /* DSP initialization                                    */
  CSR=0x100;                  /* Disable all interrupts          */
  IER=1;                      /* Disable all interrupts except NMI   */
  ICR=0xffff;                 /* Clear all pending interrupts       */

/*********************************************************************
* Standard 6211/6711 DSK includes 2 MT48LC1M16A1-7 devices = 4MB SDRAM   *
* For these devices use the following parameter:                *
*   EMIF_SDCTRL=0x07126000                                        *
* If MT48LC1M16A1-10 devices are installed use the following parameter:  *
*   EMIF_SDCTRL=0x07227000                                        *
* /|\ 16Mb parts = 4MB SDRAM /|\   *-----------------------------------*
*---------------------------------*  \|/ 64Mb parts = 16MB SDRAM \|/   *
```

```
* If MT48LC4M16A2-10 devices are installed use the following parameter:  *
*   EMIF_SDCTRL=0x57227000                                              *
*********************************************************************/


 *(unsigned volatile int *)EMIF_GCR = 0x3300;    /* EMIF global control       */
 *(unsigned volatile int *)EMIF_CE1 = CE1_32;    /* EMIF CE1 control, 32bit  */
 *(unsigned volatile int *)EMIF_SDCTRL = 0x07126000; /* EMIF SDRAM control   */
 *(unsigned volatile int *)EMIF_CE0 = 0x30;      /* EMIF CE0 control          */
 *(unsigned volatile int *)EMIF_SDRP = 0x61a;    /* EMIF SDRM refresh period */
 *(unsigned volatile int *)EMIF_SDEXT= 0x54529;  /* EMIF SDRM extension       */

 //while(1){



 channel = 0;
 for(count=0;count<100;count++){
     for(t = 0; t < 3; t++)
      {
     re[t][count] = sample(channel) - 127;
       channel = channel + 1;
       im[t][count] = sample(channel) - 127;
       channel = (channel + 7)%12;
      }
           //printf("%d",samples);
 }
```

```
  covariance(re,im);
  music(rS,iS);


  while(1){
      sample(0);
      //delay_usec(1000);
  }


}
unsigned int sample(short channel){
  int temp;
  unsigned int  data;
  temp = *(unsigned volatile int * ) port;
  temp = temp & 0xC3FFFFFF;
  temp = temp | channel << 26;
  *(unsigned volatile int * )port = temp;
  temp = temp | 0x08000000;    //0000 1000 000000000000000000000000;
  *(unsigned volatile int * )port = temp;
  temp = temp & 0xF7FFFFFF;
  temp = temp | 0x01000000;            //0000 0001 000000000000000000000000;
  *(unsigned volatile int * )port = temp;
  delay_usec(5);
  temp = temp & 0xFEFFFFFF;
  *(unsigned volatile int * )port = temp;
  delay_usec(113);
  temp = *(unsigned volatile int * )port;

  data = temp >> 16;

  return data ;

}


void delay_usec(int usec){

  int i;

  for(;usec>0;usec--){

       for(i=0;i<8;i++);

  }

}

void covariance(int re[3][100], int im[3][100] ){

  int i, j, k;

  for(i = 0; i < 3; i++){

    for(j = 0; j < 3; j++){

      rS[i][j] = 0;

      iS[i][j] = 0;

    }

  }


  for(i = 0; i < 3; i++){

    for(j = 0; j < 3; j++){

      for(k = 0; k < 100; k++){

        rS[i][j] = rS[i][j] + re[i][k]*re[j][k] + im[i][k]*im[j][k];

      iS[i][j] = iS[i][j] + re[i][k]*im[j][k] - im[i][k]*re[j][k];

      }

      }
  }

}
```

The code implementing the MUSIC Algorithm in the DSP for estimating the DoA from the samples obtained by the ADC, is as follows:

```
#include<stdio.h>

#include<c6x.h>

#include "c6211dsk.h"

#include <math.h>

void music(int[3][3],int[3][3]);

double pi = 3.141592;

double Rev0, Rev1, Rev2;

double Imv0, Imv1, Imv2 = 0.0;

double eps = 1.0e-4;


//int rS[3][3],iS[3][3];

/*int main(){

  double x_arr[3];

  double z_arr[3];

  double* x = &x_arr[0];

  double* z = &z_arr[0];

  *x = 0.684988;

  *++x = 0.684988; *++x = 1.0;

  x--;x--;

  *z = 0.728555;

  *++z = -0.728555;
  *++z = 0.0;
  z--;z--;
  music(x, z);
}*/

void music(int rS[3][3], int iS[3][3]){
  double ReS[3][3], ImS[3][3];
  int i,j,k;
  double alpha,beta,gamma;
  double c,d,h;
  double delta_sq, delta, theta;
  double x1, x2, x3;
  double e1, e2, e3, e;
  double Repivot_0, Impivot_0, Repivot_1, Impivot_1;
  double Repivot_20, Repivot_21, Impivot_21;
  double mag;
  double wav = 0.3*pi;
  double angle;
  double thetamax, valmin;
  double P, ReP, ImP;

  for(i=0;i<3;i++){
      for(j=0;j<3;j++){
            ReS[i][j]=(double)rS[i][j];
            ImS[i][j]=(double)iS[i][j];
      }
  }
```

```
//printf("%f, %f, %f", ImS[0][2], ImS[1][2], ImS[2][2]);

alpha = (ReS[0][0]+ReS[1][1]+ReS[2][2])/3;

beta = (ReS[0][0]*ReS[1][1] + ReS[1][1]*ReS[2][2] + ReS[2][2] * ReS[0][0]);

beta = beta - ( ReS[0][1] * ReS[0][1] + ImS[0][1] * ImS[0][1] ) ;

beta = beta - ( ReS[1][2] * ReS[1][2] + ImS[1][2] * ImS[1][2] ) ;

beta = beta - ( ReS[2][0] * ReS[2][0] + ImS[2][0] * ImS[2][0] ) ;


gamma = ReS[0][0] * ( ReS[1][2] * ReS[1][2] + ImS[1][2] * ImS[1][2] );

gamma = gamma + ReS[1][1] * ( ReS[2][0] * ReS[2][0] + ImS[2][0] * ImS[2][0] );

gamma = gamma + ReS[2][2] * ( ReS[0][1] * ReS[0][1] + ImS[0][1] * ImS[0][1] );

gamma = gamma - ReS[0][0] * ReS[1][1] * ReS[2][2] ;

gamma = gamma - 2 * (ReS[0][1] * ReS[1][2] * ReS[2][0]);

gamma = gamma + 2 * ReS[0][1] * ( ImS[1][2] * ImS[2][0] );

gamma = gamma + 2 * ReS[1][2] * ( ImS[2][0] * ImS[0][1] );

gamma = gamma + 2 * ReS[2][0] * ( ImS[0][1] * ImS[1][2] );


//printf("\nalpha = %f, beta = %f, gamma = %f\n",alpha, beta, gamma) ;


c= -3.0*alpha*alpha + beta;

d= alpha*beta-2.0*alpha*alpha*alpha+gamma;


delta_sq = -c/3.0;

delta = sqrt(delta_sq);

h = 2.0*delta*delta*delta;


theta = 1.0/3.0*acos(-d/h);
```

```
x1 = 2.0*delta*cos(theta);

x2 = 2.0*delta*cos(2.0*pi/3.0 - theta);

x3 = 2.0*delta*cos(2.0*pi/3.0 + theta);

printf("\nThe roots of the equation are %f  %f  %f\n", x1+alpha , x2+alpha, x3+alpha);


e1 = x1 + alpha;

e2 = x2 + alpha;

e3 = x3 + alpha;


e = e1;

if(e1 > e2) e = e2;

if(e2 > e3) e = e3;


//printf("e = %lf",e);


ReS[0][0] -= e;

ReS[1][1] -= e;

ReS[2][2] -= e;


Repivot_1 = ReS[1][0] * ReS[1][0] + ImS[1][0] * ImS[1][0] - ReS[0][0] * ReS[1][1];

Repivot_0 = ReS[2][0] * ReS[1][1] - ReS[1][0] * ReS[2][1] + ImS[1][0] * ImS[2][1];

Impivot_0 = ReS[1][1] * ImS[2][0] - ReS[1][0] * ImS[2][1] - ReS[2][1] * ImS[1][0];


Repivot_21 = ReS[0][2] * ReS[1][0] - ReS[0][0] * ReS[1][2] - ImS[0][2] * ImS[1][0];

Impivot_21 = ReS[0][2] * ImS[1][0] + ReS[1][0] * ImS[0][2] - ReS[0][0] * ImS[1][2];

Repivot_20 = ReS[1][1] * ReS[2][2] - ReS[1][2] * ReS[1][2] - ImS[1][2] * ImS[1][2];
```

```
//printf("%lf %lf %lf\n", Repivot_1, Repivot_0, Impivot_0);


if (fabs(Repivot_1) < eps && (fabs(Repivot_0) < eps && fabs(Impivot_0) < eps)
    && fabs(Repivot_20) < eps && (fabs(Repivot_21) < eps && fabs(Impivot_21) < eps))


{
  Rev2 = 1.0; Rev1 = 1.0; Imv1 = 0.0;
  if(fabs(ReS[0][0]) > eps)
  {
    Rev0 = -(ReS[0][1] + ReS[0][2])/ReS[0][0];
    Imv0 = -(ImS[0][1] + ImS[0][2])/ReS[0][0];
  }
  else
  {
    Rev0 = 0.0;
    Imv0 = 0.0;
  }
}
else if ((fabs(Repivot_1) < eps) || (fabs(Repivot_0) < eps && fabs(Impivot_0) < eps))
{
  Rev2 = 0.0;


  if(fabs(ReS[0][0]) < eps || (fabs(ReS[1][0]) < eps && fabs(ImS[1][0]) < eps) || (fabs(ReS[2][0]) <
eps && fabs(ImS[2][0]) < eps))
  {
    Rev1 = 0.0; Imv1 = 0.0; Rev0 = 1.0; Imv0 = 0.0;
```

```
      }
      else
      {
        Repivot_0 = ReS[1][0] * ReS[2][1] - ImS[1][0] * ImS[2][1] - ReS[2][0] * ReS[1][1];
        Impivot_0 = ReS[1][0] * ImS[2][1] + ReS[2][1] * ImS[1][0] - ReS[1][1] * ImS[2][0];


        Repivot_1 = ReS[0][0] * ReS[2][1] - ReS[2][0] * ReS[0][1] + ImS[2][0] * ImS[0][1];
        Impivot_1 = ReS[0][0] * ImS[2][1] - ReS[2][0] * ImS[0][1] - ReS[0][1] * ImS[2][0];


        if((fabs(Repivot_0) < eps && fabs(Impivot_0) < eps) && (fabs(Repivot_1) < eps &&
fabs(Impivot_1) < eps))
        {
          Rev1 = 1.0; Imv1 = 0.0;
          mag = ReS[1][0] * ReS[1][0] + ImS[1][0] * ImS[1][0];
          Rev0 = -ReS[1][1] * ReS[1][0]/mag;
          Imv0 = ReS[1][1] * ImS[1][0]/mag;
        }
        else
        {
          Rev1 = 0;
          Imv1 = 0;
          Rev0 = 0;
          Imv0 = 0;
        }
      }
    }
    else
```

```
{
 Rev2 = 1.0;
 Rev1 = -Repivot_21/Repivot_1;
 Imv1 = -Impivot_21/Repivot_1;
 mag = Repivot_0 * Repivot_0 + Impivot_0 * Impivot_0;
 Rev0 = -Repivot_20 * Repivot_0/mag;
 Imv0 = Repivot_20 * Impivot_0/mag;

 }

printf("\nThe eigenvector corresponding to min eigenvalue is %lf + i%lf \n %lf + i%lf \n %lf + i%lf \n",
Rev0, Imv0, Rev1, Imv1, Rev2, Imv2);

 /*for(k = 0; k < 180; k++)
 {
  angle = (double)k*pi/180.0;

  ReP = Rev0*cos(wav*cos(angle)) + Imv0*sin(wav*cos(angle))
      + Rev1*cos(wav*cos(angle+2*pi/3)) + Imv1*sin(wav*cos(angle+2*pi/3))
      + Rev2*cos(wav*cos(angle-2*pi/3)) + Imv2*sin(wav*cos(angle-2*pi/3));

  ImP = Imv0*cos(wav*cos(angle)) - Rev0*sin(wav*cos(angle))
      + Imv1*cos(wav*cos(angle+2*pi/3)) - Rev1*sin(wav*cos(angle+2*pi/3))
      + Imv2*cos(wav*cos(angle-2*pi/3)) - Rev2*sin(wav*cos(angle-2*pi/3));

  P = ReP * ReP + ImP * ImP;
```

```
    if (k == 0) valmin = P;
    if(P < valmin)
    {
      valmin = P; thetamax = angle;
    }
  }

  thetamax *= 180/pi;
  printf("The DOA is %lf\n", thetamax);
      */
}
```