Data Acquisition, Processing and Analysis for Distributed Decision Support

Broad Contents

Chapter 3

Processing of Acquired Data

3.1 Objectives of Data Processing

Data processing or data pre-processing, as it is normally known in the field of Nondestructive Testing, has a number of procedures and methods, all of which aim at bringing every bit of available data to the same level, for an objective comparison. These procedures and methods attempt to remove experimental disparities and extraneous noise while preserving the original signature of the data. They hoist the data, often obtained at different times, having different sizes, on the same platform thereby removing unwanted experimental variations, and sometimes, errors.

Depending upon the type of data that is being studied (signal, image, etc.), the domain of discourse (eddy current testing, ultrasonic testing, seismology, etc.), and the objective of the study (classification of data, prediction of data, controlling a device, etc.), the methods of pre-processing and their sequence will vary.

Typical parameters that need to be made uniform before any meaningful comparison among data can take place include: (a) identical data length (for column vector data), (b) identical gain settings while acquiring data, (c) identical sampling rates, (d) identical experimental conditions, if possible, while performing the experiments, (e) low noise conditions, (f) identical "depth" of each pixel (in the case of image data; 8-bit, 12-bit, etc.), (g) identical image sizes, if possible (128 x 128 pixels, for example), and so on.

In spite of the strict, controlled fashion in which experimental data could be gathered, there are situations (e.g., data acquisition from a running plant, or data acquisition from plants / components situated at different geographical locations, where the data is acquired by different teams using equipments of different makes, data acquisition tasks separated in time where environmental conditions are different, etc.) where data pre-processing becomes a necessity.

It is not necessary that the pre-processing procedures and methods must stick on to the same domain in which the original data was acquired. For example, if a vibration signal is acquired in time domain, from which if we wish to remove a known extraneous noise (e.g., a repetitive ball-bearing abrasion noise) this could well be achieved in the frequency domain. This approach is particularly useful in situations where the desirable "signature" and the undesirable "noise" are well separated in the frequency domain, rather than in the time domain. In such a case, the original time data is "transformed" to the frequency domain, where the noise is removed and the resulting frequency data is "inverse transformed" back to the time domain for further analysis / interpretation.

This Chapter describes some of these *pre-processing procedures and methods*. Wherever possible, the skeletal-listing of the programming / scripting code for such procedures is presented with relevant explanations. Though these code snippets have been tested extensively, Readers are urged to examine them carefully before using them. Note that while using**++** the code snippets, the prefix / include files and function declarations must be added along with any memory allocation or de-allocation functions******.

Once again, it must be emphasised that the concepts below are presented more from a practical point of view. A detailed and comprehensive treatment of each of these concepts is beyond the scope of this Monograph, for which, Readers are encouraged to refer other published papers and textbooks. The next Chapter, Chapter 4, describes the concepts, methods and procedures of *Data Analysis*, **after** necessary pre-processing.

3.2 Methods of Data Processing

Conceptually, the pre-processing methods used for image (2-d data) analysis are similar to those used for signal (1-d signal data is referred to as column vectors in this Monograph) analysis. Hence, we shall restrict ourselves mostly to the 1-d case, (except the last example in this Series, as listed below) and point out the differences with respect to the 2-d data as we go along. The following pre-processing methods are described in this Section:

- Bias Removal
- Normalisation with respect to a Column Vector
- Normalisation with respect to Columns in several Column Vectors
- Normalisation with respect to Rows in several Column Vectors
- Windowing
- Smoothening or Averaging
- Fixing the Record Length of Column Vector data to a specified 2-power-N value
- Time Shifting
- Signal Averaging (Record-wise averaging)
- Polarity Thresholding
- Typical Image Pre-processing Sequence

3.2.1 Bias Removal

It is quite possible that during data acquisition, particularly from certain digital storage oscilloscopes (DSO), the entire signal value is "lifted" by a certain positive value. In effect, the DSO imparts a positive DC bias to the whole signal, which will interfere with the subsequent analysis of the signal.

Figure 3.1 shows a sinusoidal waveform which has a DC bias of value 1.0

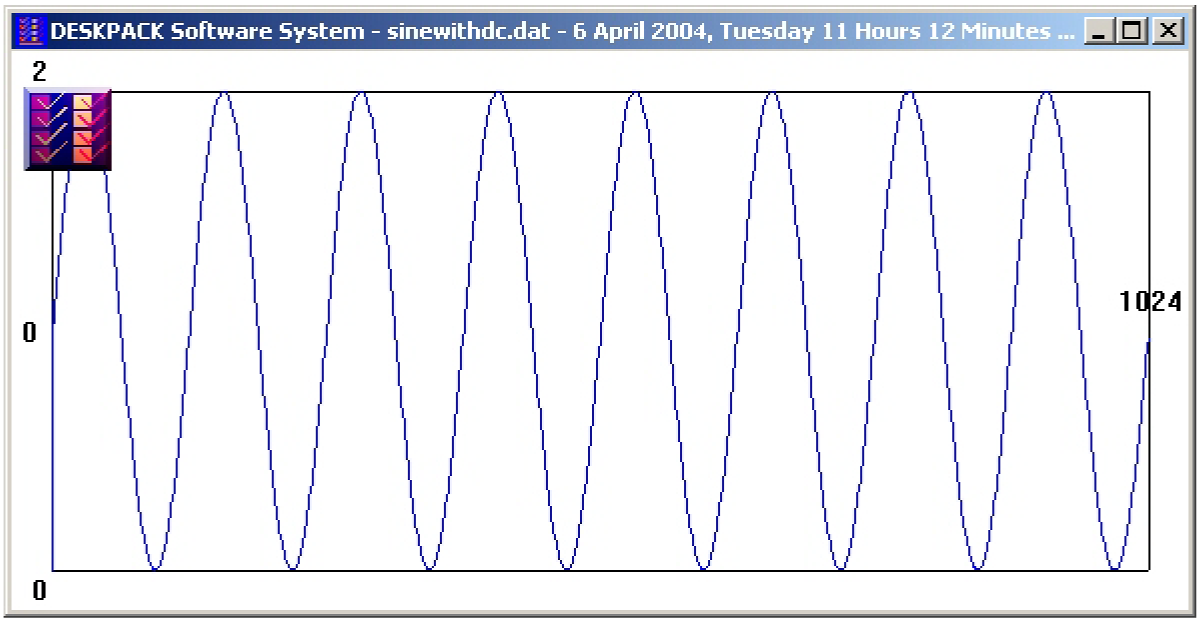


Figure 3.1 Sinusoidal waveform with a positive (1.0) DC Bias

Note in Fig.3.1 that the lowest value of the signal is 0.0 instead of -1.0 and the highest value is 2.0 instead of 1.0.

Figure 3.2 shows the auto-correlation waveform of the above sinusoidal signal with the DC bias present in it.

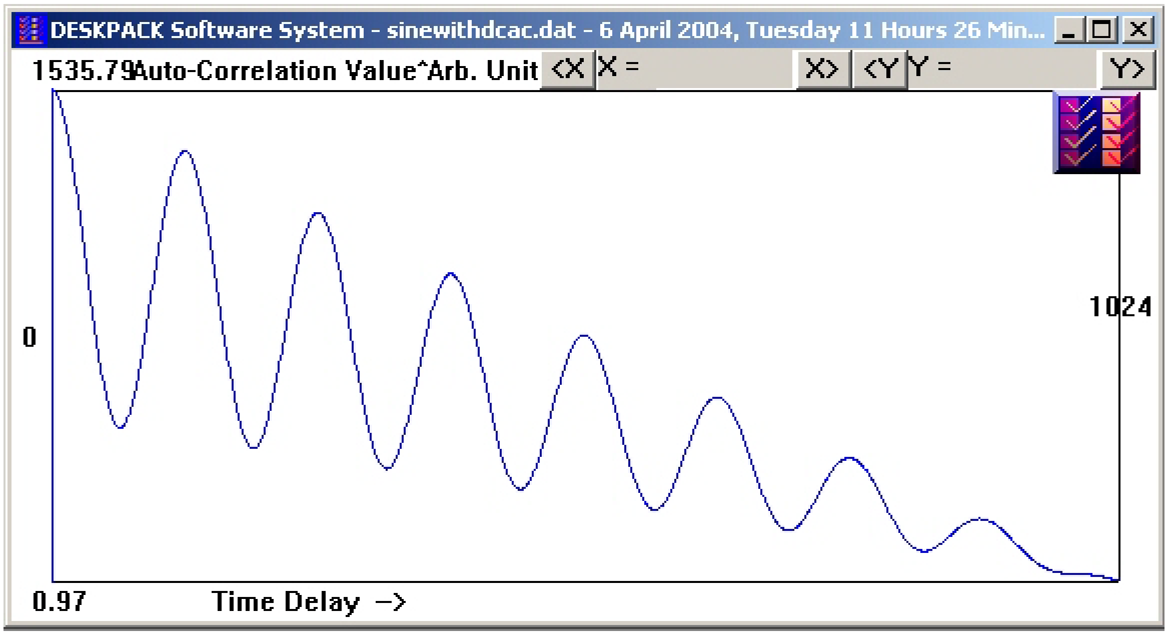


Figure 3.2 Auto-Correlation of the Sine Signal with DC bias present

Note in Figure 3.2 that a "downward trend" is seen in the auto-correlation signal which is illusory, and has come as a result of the non-removal of the DC bias before analysis.

The DC bias can be removed in a simple case# by finding the mean of the sinusoidal signal and removing it from the entire signal. Code listing 3.1 shows this process.

```
void rembiasdata(float *data, int rlength)
{
int i;
float sum;

sum = 0.0;
for(i=0; i<rlength; i++)
    {
    sum = sum + data[i];
    }
sum = sum/rlength;
for(i=0; i<rlength; i++)
    {
    data[i] = data[i] - sum;
    }
}
```

Code Listing 3.1 Code to Remove DC Bias in a Simple# Case

Once the DC bias is removed from the sinusoidal signal of Figure 1, using the code above, the signal looks like in Figure 3.3.

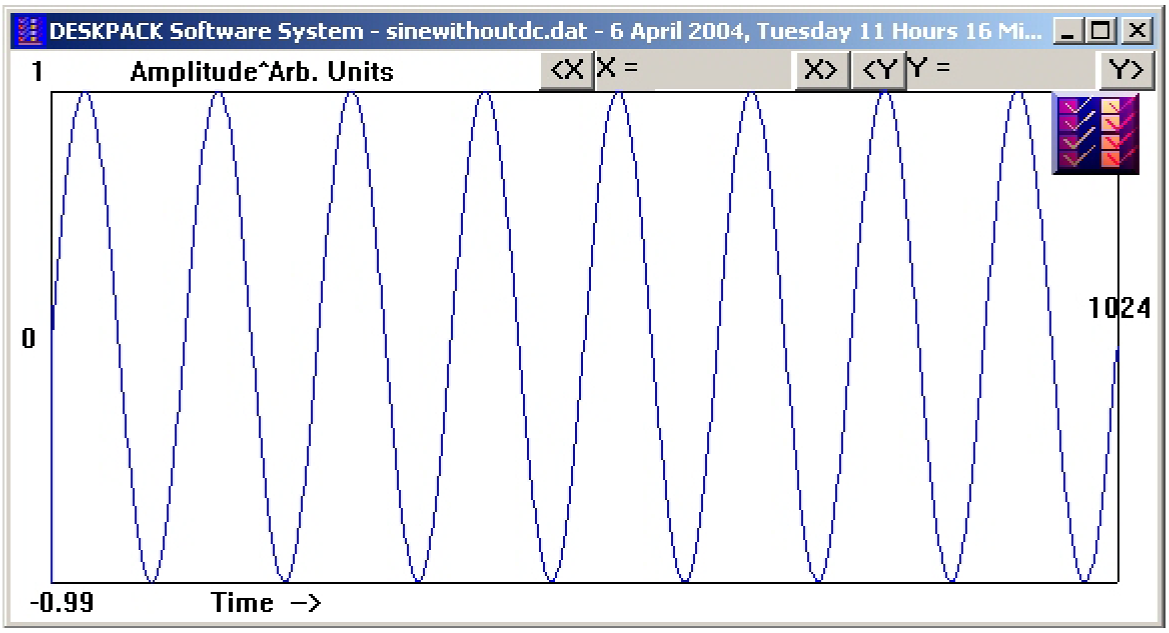


Figure 3.3 Sinusoidal waveform of Fig.1, after DC bias removal

If we now find the auto-correlation of this processed sinusoidal signal, we get the correct function as in Figure 3.4.

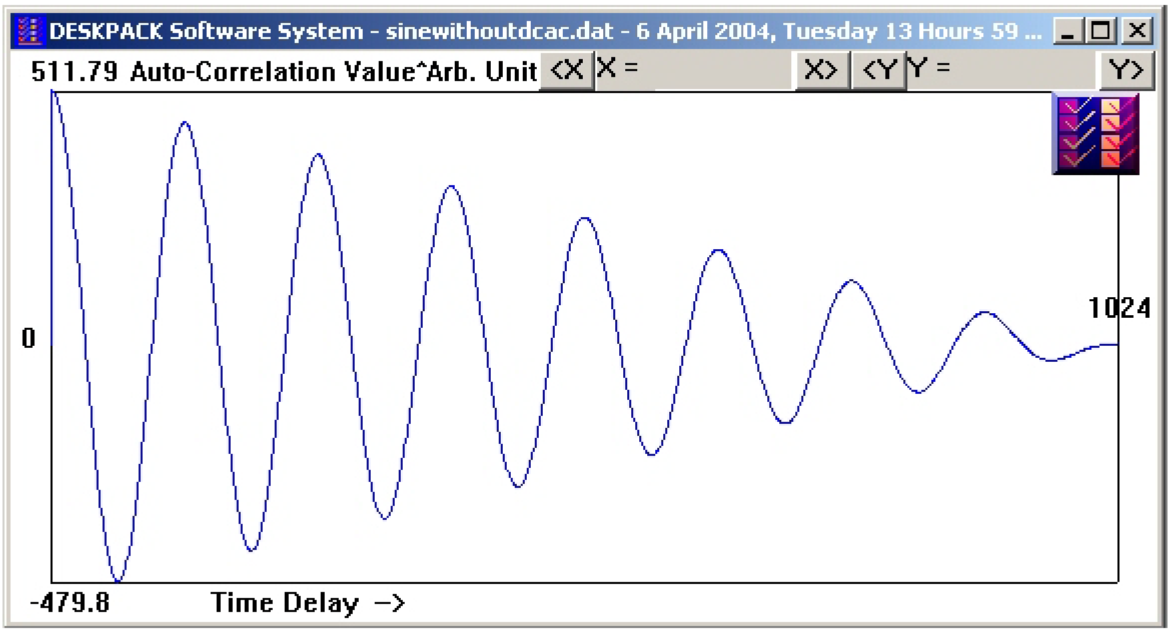


Figure 3.4 Auto-correlation function of a DC bias removed Sine waveform

It worthwhile to compare the Figures 3.2 and 3.4 shown above. Note that the apparent "downward trend" is absent in Figure 3.4, showing the correct auto-correlation function of the sine wave.

3.2.2 Normalisation – Single Column Vector

At times, it becomes essential to "map" a range of values (represented, may be, by a column vector or a signal), from their existing extremes to say, between 1 and 0. Example situations where these might be required is when one wants to plot the waveform within a given Window, or when one wants to feed the original data to a neural network that is constrained to accept values between 1.0 and 0.0

In such cases, there are at least two different ways in which this normalisation can be done: (a) zero-mean unit-variance method and (b) direct mapping method.

The zero-mean unit-variance is the standard method, whose code listing is shown in Code Listing 3.2. In this method, for a given column vector, its (vector's) mean is subtracted from each data point, and is divided by its (vector's) variance.

```
void stdnormsingledata(float *rawdata, int r_length)
{
int count;         /* Just counters ! */
float mean;
float variance;

/* Values are Read and the 'mean' is found */
mean = 0.0;
for(count=0; count<r_length; count++)      /* count is the counter for the data
points in a file */
   { /* index is the counter used for referring to data points later in the
Program */
   mean = mean + rawdata[count];
   }
mean = mean/r_length;

/* Variance is found */
variance = 0.0;
for(count=0; count<r_length; count++)
   {
   variance = variance + (rawdata[count] - mean)*(rawdata[count] - mean);
   }
variance = sqrt(variance/r_length);

/* Values are transformed into zero-mean, unit-variance - and are written back
to file FI */
for(count=0; count<r_length; count++)      /* count is the counter for the data
points in a file */
   { /* index is the counter used for referring to data points later in the
Program */
   rawdata[count] = (rawdata[count] - mean)/variance;
   }
}
```

Code Listing 3.2 Zero-Mean Unit-Variance Normalisation Code

In the direct mapping method, a range is fixed with the allowable max and min values. The column vector values are then directly mapped in such a way that the column vector's maximum value is set at max and the column vector's minimum value is set to the min value. This process results in a certain scaling factor, which governs this mapping.

Once the max, min and the scaling factors are computed, we can perform either the forward transform (Column Vector to Pre-defined Range), or the reverse transform (Pre-defined Range to the Column

Vector). In certain situations the reverse transform becomes essential, where we need to get back to the original domain either for interpretation or for controlling a device. A good example of such a situation is the use of multi-layered perceptron artificial neural network for prediction of data values.

The following Code Listing (Code Listing 3.3) shows this process.

```
void maprangedata(float **rawdata, float *sf, int n_files, int i_nodes, float
mn, float mx, int reverse, int column)
{
float *maxx;
int index;
int count;
float range;

range = (mx - mn);


/* This fragment computes the scaling factors, if forward mapping is requested
*/
if(column)
  {
   if(reverse) {} /* Does nothing for reverse mapping */
   else
       {
       sf[0] = mn;
       sf[1] = mx;
       for(index=0; index<n_files; index++)
         {
          maxx[index] = -33000.0;
          for(count=0; count<i_nodes; count++)
             {
             if(rawdata[index][count] >= maxx[index]) {maxx[index] =
rawdata[index][count];}
             }
           sf[index+2] = range/maxx[index];
         }
       }
/* Computation of Scaling factors ends here - stored in sf[count] */

/* Data values are mapped within the selected range here */
   if(reverse) /* Reverse mapping done here */
     {
     for(index=0; index<n_files; index++)
            {
            for(count=0; count<i_nodes; count++)
               {
               rawdata[index][count] = (rawdata[index][count]- mn)/sf[index+2];
               }
            }
     }
     else /* Forward mapping done here */
        {
        for(index=0; index<n_files; index++)
           {
           for(count=0; count<i_nodes; count++)
              {
              rawdata[index][count] = (rawdata[index][count]*sf[index+2]) + mn;
              }
           }
        }
   }
else
   {
   if(reverse) {} /* Does nothing for reverse mapping */
   else
      {
      sf[0] = mn;
      sf[1] = mx;
      for(count=0; count<i_nodes; count++)
        {
        maxx[count] = -33000.0;
        for(index=0; index<n_files; index++)
          {
           if(rawdata[index][count] >= maxx[count]) {maxx[count] =
rawdata[index][count];}
          }
          sf[count+2] = range/maxx[count];
        }
      }
/* Computation of Scaling factors ends here - stored in sf[count] */

/* Data values are mapped within the selected range here */
   if(reverse) /* Reverse mapping done here */
     {
     for(count=0; count<i_nodes; count++)
            {
            for(index=0; index<n_files; index++)
               {
               rawdata[index][count] = (rawdata[index][count]- mn)/sf[count+2];
               }
            }
     }
     else /* Forward mapping done here */
        {
        for(count=0; count<i_nodes; count++)
           {
           for(index=0; index<n_files; index++)
              {
              rawdata[index][count] = (rawdata[index][count]*sf[count+2]) + mn;
              }
           }
        }
   }
/* Mapping of data values ends here */


}
```

Code Listing 3.3 Code Snippet that shows both forward and reverse Mapping within a given Range, fixed by Maximum and Minimum values

3.2.3 Normalisation – Several Column Vectors

In this case, conceptually, however, the procedure is the same as above, except that many column vectors are involved. In such cases where multiple column vectors are to normalised (zero-mean, unit-variance method), the Code Listing 3.2 is repeatedly applied for all the column vectors.

3.2.4 Normalisation – Row-wise, of Several Column Vectors

This is an important case of normalisation, where a set of column vectors (which can be visualised as a matrix) are to be normalised, across the rows of the matrix. This type of normalisation is required when a set of feature vectors are to be normalised. For example, if a feature vector A has features f1A, f2A, f3A…..fnA, and another feature vector B has features f1B, f2B, f3B…..fnB, then we need to normalise f1A and f1B; f2A and f2B; and so on.

If there are more feature vectors, say, A,B,C,D,……each in a separate file, up to `n_files`, then the procedure to normalise that matrix is shown in Code Listing 3.4

```
void stdnormalise(float **rawdata, int n_files, int i_nodes)
{
float *mean;
float *variance;
int index;
int count;


/* Mean of all the features, in every feature-vector is calculated */
   for(count=0; count<i_nodes; count++)
      {
      mean[count] = 0.0;
      for(index=0; index<n_files; index++)
         {
         mean[count] = mean[count] + rawdata[index][count];
         }
      mean[count] = mean[count]/n_files;
      }

/* Variance of all the features, in every feature-vector is calculated */
   for(count=0; count<i_nodes; count++)
      {
      variance[count] = 0.0;
      for(index=0; index<n_files; index++)
         {
         variance[count] = variance[count] + (rawdata[index][count] -
mean[count])*(rawdata[index][count] - mean[count]);
         }
      variance[count] = sqrt(variance[count]/n_files);
      }

/* Feature vectors are transformed into zero-mean, unit-variance normalised form
here */
   for(count=0; count<i_nodes; count++)
      {
      for(index=0; index<n_files; index++)
         {
         rawdata[index][count] = (rawdata[index][count] -
mean[count])/variance[count];
         }
      }

}
```

Code Listing 3.4 Code Snippet for Row-wise normalisation of a Matrix (or a set of column vectors)

3.2.5 Windowing

While acquiring data, because of the requirement of finite amount data per record, it is quite possible that the ends of the data record are not smooth (say equal to zero at both ends) but discontinuous. This is particularly true if a rectangular window is used (as is normally the case) during data acquisition. Certain analysis procedures, particularly the Fourier analysis assumes that the signals extend beyond the record length on both sides, as if the truncated data record is placed one after the other in an infinite sequence, and that they are periodic.

Under this assumption, discontinuous records (at the edges of the rectangular window) may result in wrong Fourier spectra and may lead to spectral leakage and wrong interpretation. The failure of Fourier series to converge at discontinuities is also called the Gibbs phenomena.

In order to avoid this signal data records are "windowed" effectively reducing their end points to zero. Three types of Windows are normally used, viz., Bartlett, Hann and Welch. This process is equivalent to "weighting" the data values. One can visualise rectangular window as one where every sample in the data record has a weight = 1; in other windows, this weight is a function that reduces the discontinuities at the edges to nearly zero.

Such weightings (a) reduce the discontinuity to zero, (b) modulate the signal data record by the shape of the window, (c) reduce the side lobe height (i.e., reduce spectral leakage) in the frequency domain, and (d) increase the effective bandwidth.

The code listings for these three windows are shown below:

```
/* Implementation of the module that finds the Welch Window of a signal IN AN
ARRAY */
void welchdata(float *data, int rlength)
{
int i;
float N;

N = rlength;
for(i=0; i<rlength; i++)
   {
   data[i] = data[i]*(1.0 - pow(((i-N/2.0)/(N/2.0)),2.0));
   }
}

/* Implementation of the module that finds the Hann Window of a signal IN AN
ARRAY */
void hanndata(float *data, int rlength)
{
int i;
float N;

N = rlength;
for(i=0; i<rlength; i++)
   {
   data[i] = data[i]*0.5*(1.0 - cos((2.0*22.0*i)/(7.0*N)));
   }
}
```

```
/* Implementation of the module that finds the Bartlett Window of a signal IN AN
ARRAY */
void bartlettdata(float *data, int rlength)
{
int i;
float N,j;

N = rlength;
for(i=0; i<rlength; i++)
   {
   j = i;
   data[i] = data[i]*(1.0 - fabs((j-N*0.5)/(N*0.5)));
   }
}
```

Code Listing 3.5 Code Snippets Listing the Welch, Hann and Bartlett Windows

The following three figures (Figs. 3.5, 3.6 and 3.7) show the Welch, Hann and Bartlett windows of the sine wave data shown in Figure 3.3.

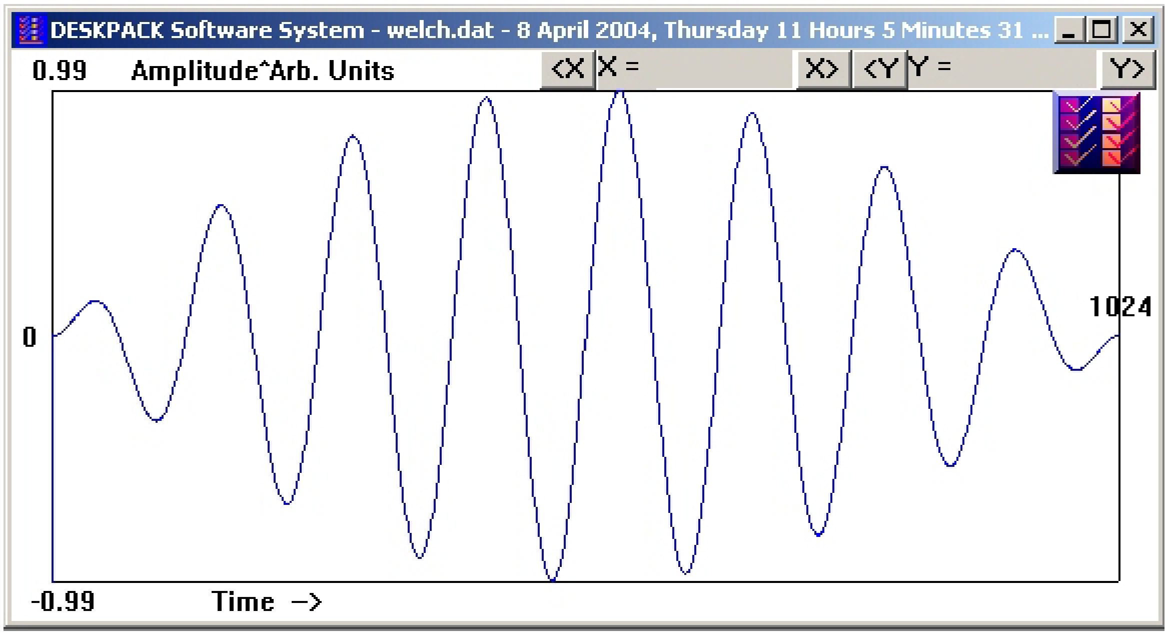


Figure 3.5 Sine Wave modified by a Welch Window

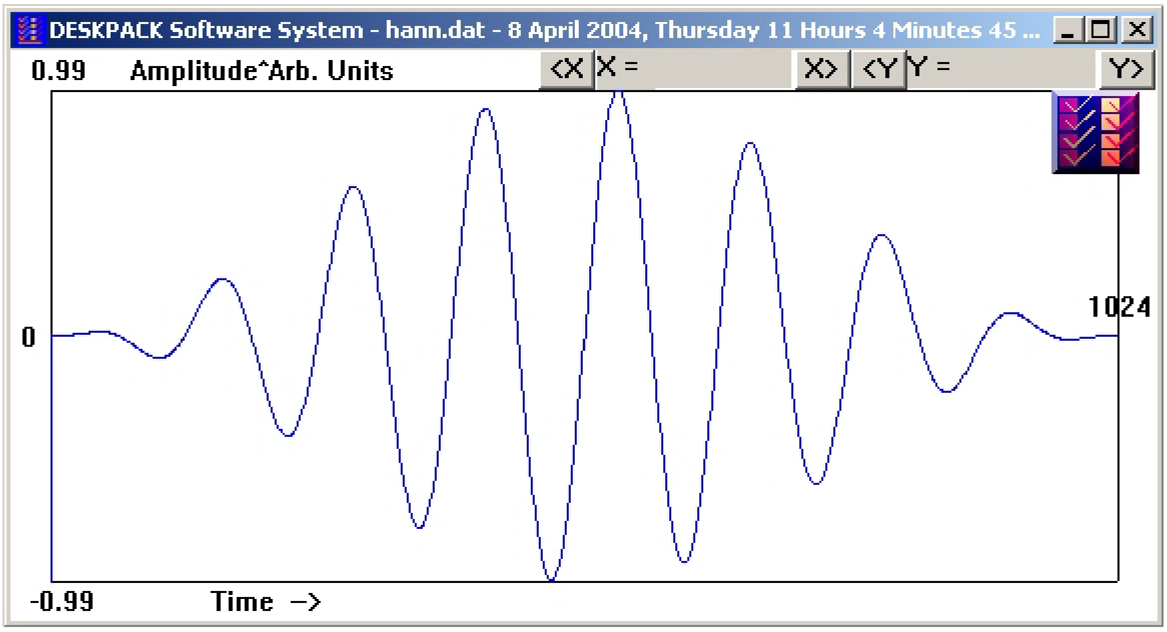

DESKPACK Software System - hann.dat - 8 April 2004, Thursday 11 Hours 4 Minutes 45 ...
0.99
Amplitude^Arb. Units
<X
X =
X>
<Y
Y =
Y>
0
1024
-0.99
Time ->

Figure 3.6 Sine Wave modified by a Hann Window

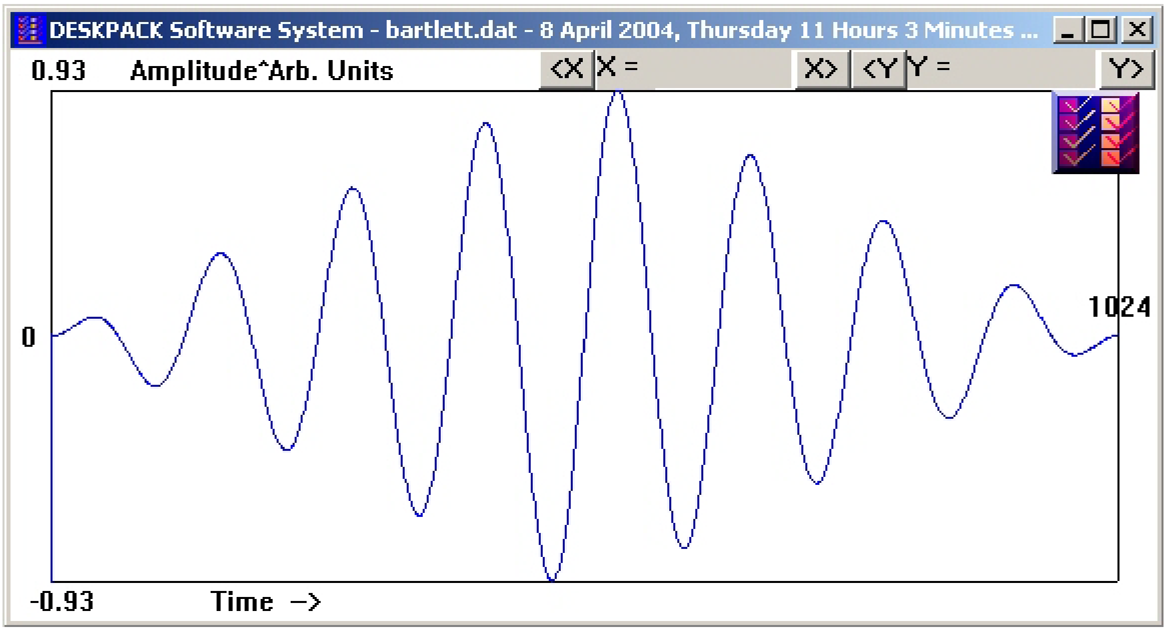

DESKPACK Software System - bartlett.dat - 8 April 2004, Thursday 11 Hours 3 Minutes ...
0.93
Amplitude^Arb. Units
<X
X =
X>
<Y
Y =
Y>
0
1024
-0.93
Time ->

Figure 3.7 Sine Wave modified by a Bartlett Window

If required, it is possible to apply a certain Window function repeatedly on a signal. Figure 3.8 shows the repeated application of Welch Window on a sinusoidal signal, thrice. Note the decrease in amplitude, with respect to the original sinusoidal signal of Fig.3.3.

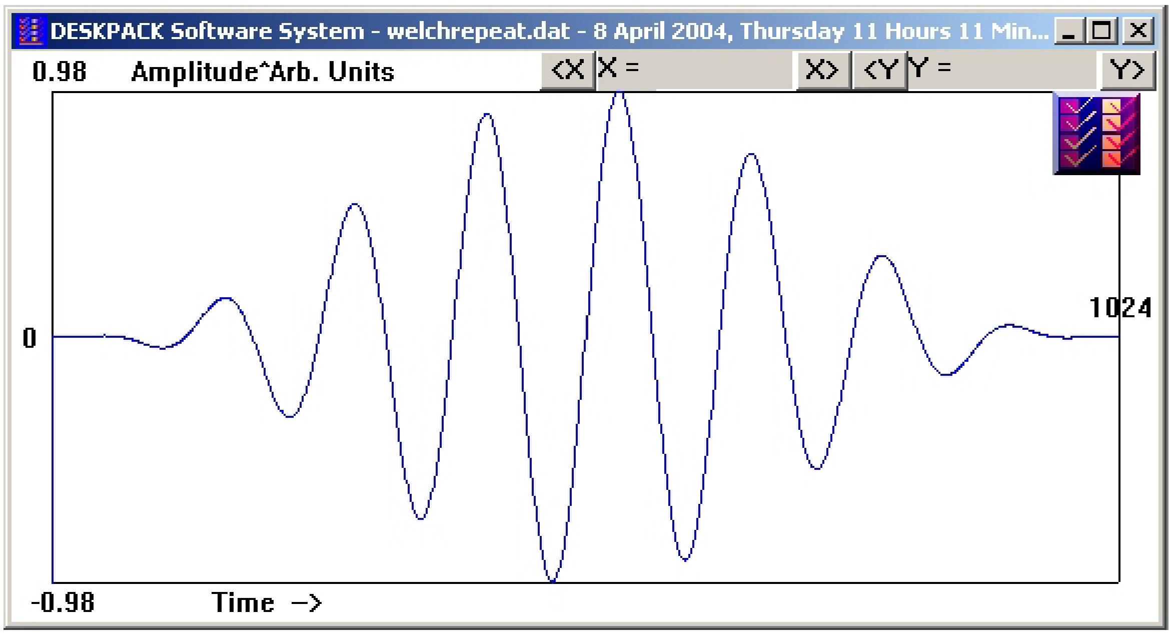


Figure 3.8 Welch Window applied thrice on a Sinusoidal signal

In each of the windowing method, note that the resultant data is nearly zero at the two edges of the signal record (at the start and at the end). Windowing methods are used in such situations where the edges are strongly discontinuous, requiring pre-processing to avoid errors particularly while using the fast Fourier transform.

3.2.6 Smoothening or Averaging

Smoothening a single signal record is different from record-wise averaging discussed in Section 3.2.9. In smoothening a signal record, usually a "moving-window average" is performed which has an equivalent effect of a low-pass filter. The length of the "moving-window" is usually a variable, fixed as 3, 4 or 5 points – or even more – depending upon the nature of smoothening required.

The code listing for this operation is fairly straightforward and is given in Code Listing 3.6, below.

```
void smoothdata(float *data, int rlength, int n)
{
float *y;
int a,i,j,k;
float sum = 0.0;

/* Initialise the array with zero */
for(i=0; i<rlength; i++)
   {
   y[i] = 0.0;
   }


/* Find average for first rlength-n values */
for(i=0; i<rlength - n; i++)
   {
   j=0;
   sum = 0.0;
   while(j < n)
        {
        sum = sum + data[i+j];
        j++;
        }
   y[i] = sum/n;
   }


/* Find average for the remaining values by taking samples from the beginning */
for(; i<rlength; i++)
   {
   a=0; j=0; sum=0.0; k=i;
   while(j<n)
        {
        if(k<rlength)
          {
          sum= sum + data[k];
          k++;
          }
        else
          {
          sum = sum + data[a];
          a++;
          }
        j++;
        }
        y[i] = sum/n;
    }
    y[i]='\0';

/* Store smoothened 'y' values in data array */
for(i=0; i<rlength; i++)
   {
   data[i] = y[i];
   }
}
```

Code Listing 3.6 Code for smoothening a single data record using "moving-window" averaging

The following two figures show the effect of such smoothening based on "moving-window" averaging. Figure 3.9 shows a sinusoidal signal corrupted by high frequency noise. This signal data is subjected to a "moving-window" average smoothening with a window length of 4 data points. The resulting smoothened data is shown in Figure 3.10.

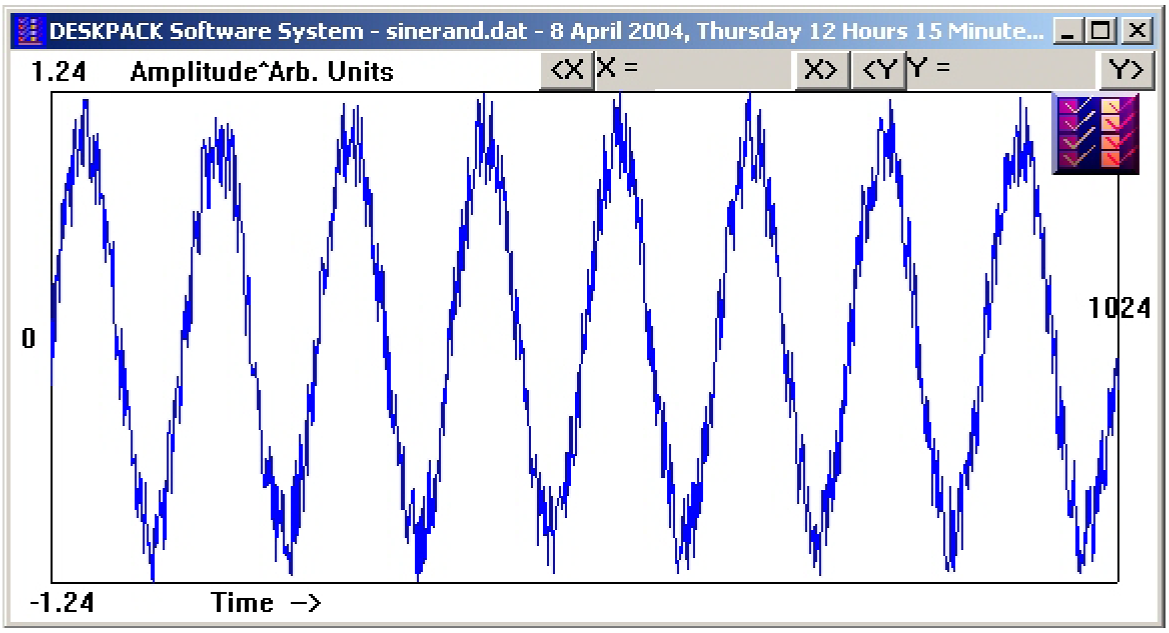

DESKPACK Software System - sinerand.dat - 8 April 2004, Thursday 12 Hours 15 Minute...
1.24
Amplitude^Arb. Units
<X
X =
X>
<Y
Y =
Y>
0
1024
-1.24
Time ->

Figure 3.9 Sine data corrupted by high-frequency noise

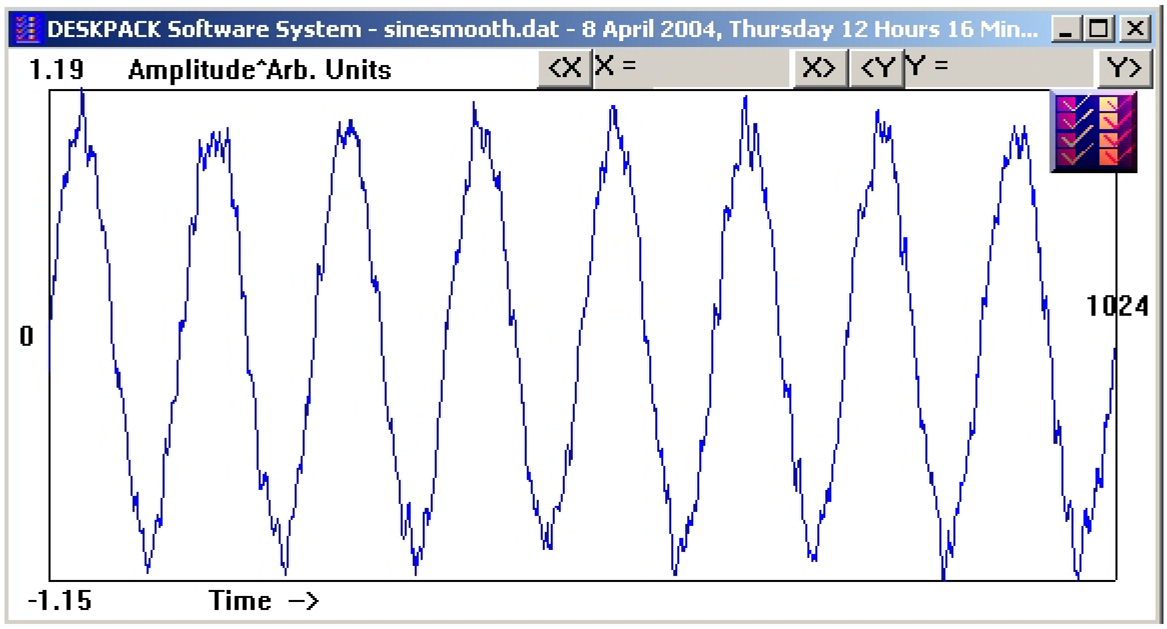

DESKPACK Software System - sinesmooth.dat - 8 April 2004, Thursday 12 Hours 16 Min...
1.19
Amplitude^Arb. Units
<X
X =
X>
<Y
Y =
Y>
0
1024
-1.15
Time ->

Figure 3.10 Smoothened waveform of the noise-corrupted sine data

Notice that during the process of smoothening, the high-frequency noise has reduced, and so is the overall amplitude of the sine wave. This is the result of a single-pass smoothening. Further smoothening is possible by additional passes.

3.2.7 Fixing the Record Length

There are many experimental situations where the acquired data's length will not be equal to 2^N where N is an integer. This unusual length may lead to problems while analysing / transforming the data using algorithms or methods that assume this conditionality. One such method is the fast Fourier transform.

In such cases, it is necessary to either remove certain data points or add a certain number of data points (usually zeros are added, in a process called zero-padding), to fix the length of the signal record to be equal to 2^N. Normally used data lengths are 256, 512, 1024, 2048, 4096 and so on.

The following program fragment shows the general methodology used:

```
setdatazero(data,rlength);
tlength = getfflength(f_name);
if(tlength<rlength) {load_data(data,tlength,f_name);}
if(tlength>=rlength) {load_data(data,rlength,f_name);}
save_data(data,rlength,f_name);
```

where, the function `setdatazero` is

```
void setdatazero(float *data, int rlength)
{
int i;

for(i=0; i<rlength; i++)
   {
   data[i] = 0.0;
   }
}
```

Code Listing 3.7 Program fragment showing the method for fixing the data length

3.2.8 Time-Shifting

In a time-domain signal record, the x-axis represents Time and the y-axis represents the amplitude of the physical process that is being measured or monitored.

In situations where a direct comparison in time is required, it might be necessary to move the data in time and synchronize it with the time instant of a reference point. In such cases, the data is time-shifted either in the positive direction or negative direction, as required by the reference point.

Figures 3.11 show a typical pulse signal (Figure 3.8) which has been time-shifted to the left by about 256 points.

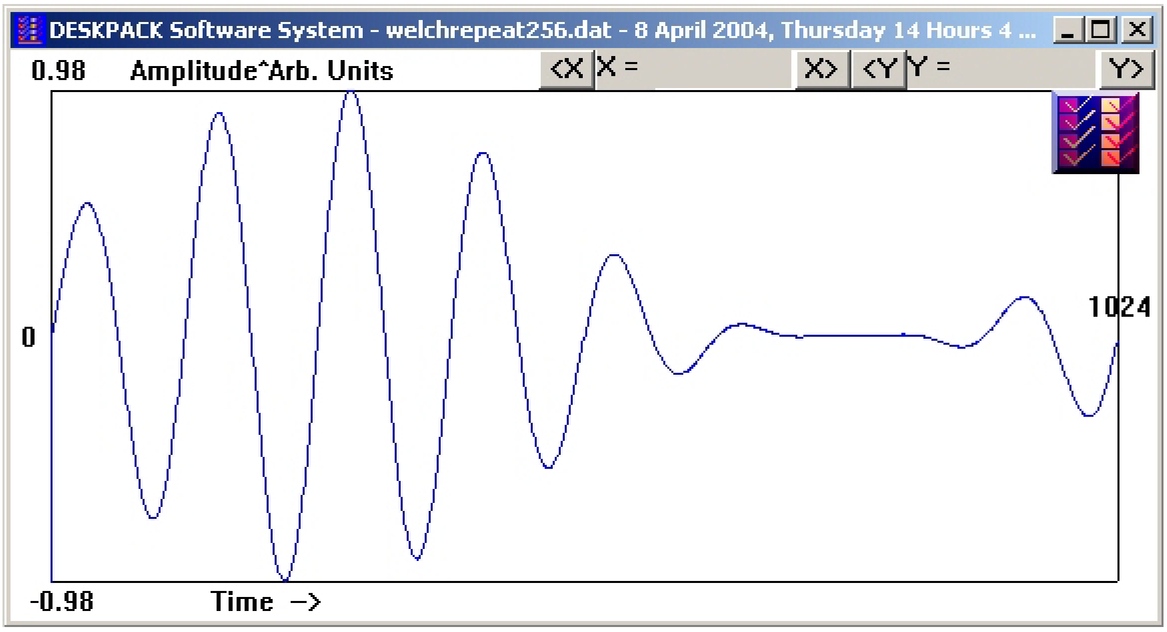

DESKPACK Software System - welchrepeat256.dat - 8 April 2004, Thursday 14 Hours 4 ...
0.98
Amplitude^Arb. Units
<X
X =
X>
<Y
Y =
Y>
0
1024
-0.98
Time ->

Figure 3.11 Time-shifted waveform of Figure 3.8, to the left, by 256 points

The following code listing shows the procedure to accomplish this for a given file that contains the data in a column vector format. In this program listing, the variable `dir` represents the direction of shift.

```
void timeshiftfile(char Infile[128], char LF[128], int dir, int tsp, int *cerr)
{
float *data, *y;
int rlength;
int i,j, sl;
FILE *fpl;
char shfn[128] = "";
char extdat[] = ".dat";
char intstr[5];

sl = strlen(Infile);
sl = sl - 4;
strncat(shfn,Infile,sl);
itoa(tsp, intstr, 10);
strncat(shfn,intstr,3);
strncat(shfn,extdat,4);

if((fpl=fopen(LF,"a")) == NULL) {*cerr = 54; exit(1);}
fprintf(fpl,"%s\n","Opened Log file from within 'timeshift' C Module");
fprintf(fpl,"%s\n","Ready to Execute the Program");

rlength = getfflength(Infile);
load_data(data,rlength,Infile);

if(dir)
  {
  for(i=0;i<rlength-tsp;i++)
     {
     y[i] = data[i+tsp];
     }
  j = 0;
  for(i=rlength-tsp;i<rlength;i++)
     {
     y[i] = data[j];
```

```
      j = j + 1;
      }
  }
else
  {

  }
save_data(y,rlength,shfn);

fprintf(fpl,"%s\n","Completed 'timeshiftfile' C Module");
fclose(fpl);
*cerr = 0;
}
```

Code Listing 3.8 Program that shifts a data record by a certain number of points

Note that even though the program function is called `timeshiftfile`, this can be used for any stream of data, irrespective of its domain. However, care must be taken while interpreting the "time-shifted" data record.

3.2.9 Signal Averaging (Record-wise Averaging)

This is a standard method used to average the noise out of an ensemble of data records. This method removes un-correlated noise effectively for an ensemble of simple# data records, having **equal** record lengths. Note that if the records present in the ensemble have different record lengths, then this procedure is not applicable.

This procedure is also common in averaging-enabled hardware, such as in modern digital storage oscilloscopes. The more the number of records in the ensemble that are averaged in this fashion, the better the "estimate" of the true data record would be.

The following code listing shows a procedure to accomplish this task.

```
  void sigrecordavg(char Infile[128],char Outfile[128],char LF[128],int *cerr)
  {

    int nfiles,rlength,i,j,r;
    float *avg;
    float **v;
    float sum;
    char *fname;
    FILE *fp;
    FILE *fpn;
    if((fp=fopen(LF,"a"))==NULL) {*cerr= 37 ;exit(1);}
     fprintf(fp,"%s\n","Opened Logfile from within 'SIGNAL RECORD AVERAGING' C
module");
     fprintf(fp,"%s\n","Ready to execute the program");
     fclose(fp);

    nfiles = getsflength(Infile);

    i=0;
    fpn=fopen(Infile,"r");
    fscanf(fpn,"%s",fname);
    rlength=getfflength(fname);
    r=rlength;

    /* check for uniform length */
    for(i=1;i<nfiles;++i)
    {
      fscanf(fpn,"%s",fname);
      rlength=getfflength(fname);

      if(rlength == r)
        r=rlength;
      else
        {
         if((fp=fopen(LF,"a"))==NULL) {*cerr= 37 ;exit(1);}
         fprintf(fp,"%s","Cannot find average since the filelength for all the
files is not uniform ");
         fclose(fp);
         exit(0);
         }
    }

    fclose(fpn);

     if((fpn=fopen(Infile,"r"))==NULL) {*cerr= 37 ;exit(1);}


  /*** load data from the files inseparate arrays ***/
     for(i=0;i<nfiles;++i)
       {
         fscanf(fpn,"%s",fname);
         load_data(v[i],rlength,fname);
       }

     fclose(fpn);



     for(i=0;i<rlength;++i)
       avg[i]=0;
     for(i=0;i<rlength;i++)
       {
          sum=0.0;

    /*** vary the files and add the corresponding values ***/
          for(j=0;j<nfiles;++j)
           {
            sum=sum+v[j][i];
            }

          avg[i]=sum/nfiles;              /*find average*/
       }

      save_data(avg,rlength,Outfile);

    if((fp=fopen(LF,"a"))==NULL) {*cerr= 37 ;exit(1);}
    fprintf(fp,"%s\n","Completed 'Signal Record Averaging' C Module");
    fclose(fp);

  *cerr = 0;
   }
```

Code Listing 3.9 Program listing that averages the noise out of an ensemble of signal records

3.2.10 Polarity Thresholding

Polarity thresholding (PL) is another noise removal technique, similar to the record averaging discussed in Section 3.2.9. However, instead of taking the average of an ensemble of signal records, PL scans a point in time in all the records of an ensemble and finds if the amplitude at that time in all the records have the same sign (either positive or negative). If for any point of time, the amplitude varies in sign for one or more records in an ensemble, the function replaces the value for that point of time in all the records as zero.

The basic assumption here is that if there is a genuine signal amplitude at that point of time (which is **not** uncorrelated noise), then it must maintain its polarity (either positive or negative) in all the records of the ensemble. Only noise would vary its polarity at a given point in several records.

This method is usually followed by an algorithm called Minimization, which after scanning a certain point of time in all the records of an ensemble, finds the least value (positive) or the value closest to zero (negative) and re-generates a single data record from a given ensemble.

Both Polarity Thresholding and Minimization have been used in conjunction with a popular data processing method called the Split Spectrum processing to weed out known, uncorrelated noise.

The following code listing shows the methods for both Polarity Thresholding and Minimization.

```
void    polarthreshold(char    Infile[128],char    Outfile[128],char    LF[128],int
rbias,int *cerr)
   {
     int nfiles,rlength,i,j,r,count=0;
     FILE *fp;
     FILE *fpn;
     char *fname;
     float *thres;
     float **v;
     float posmin=35000.0,negmax=-35000.0;

     if((fp=fopen(LF,"a")) == NULL) { *cerr = 37; exit(1); }
     fprintf(fp,"%s\n","Opened  Log  File  from  within  'POLARITY  THRESHOLD'   C
module ");
     fprintf(fp,"%s%d\n","Ready To execute the program ",rbias);
     fclose(fp);

  /*** get the no of files ***/
     nfiles=getsflength(Infile);

     fpn=fopen(Infile,"r");
     fscanf(fpn,"%s",fname);
     rlength=getfflength(fname);
     r=rlength;

  /*** check for uniform length ***/
     for(i=1;i<nfiles;++i)
      {
       fscanf(fpn,"%s",fname);
       rlength=getfflength(fname);

       if(rlength == r)
        r=rlength;
       else
        {
         if((fp=fopen(LF,"a")) == NULL) { *cerr = 37; exit(1); }
          fprintf(fp,"%s\n","cannot  find  threshold  value  since  the  file  length
for all the files is not uniform");
```

```
          fclose(fp);
          exit(1);
         }
       }

       fclose(fpn);

 /*** load the data in separate arrays ***/
       fpn=fopen(Infile,"r");
       for(i=0;i<nfiles;++i)
        {
         fscanf(fpn,"%s",fname);
         load_data(v[i],rlength,fname);
        }

        fclose(fpn);

        if(rbias==1)
         {
          if((fp=fopen(LF,"a")) == NULL) { *cerr = 37; exit(1); }
          fprintf(fp,"%s\n","Remove bias");
          fclose(fp);

           removebias(v,nfiles,rlength);
          if((fp=fopen(LF,"a")) == NULL) { *cerr = 37; exit(1); }
          fprintf(fp,"%s\n","Remove bias completed");
          fclose(fp);

         }

 /*** find the positive min & negative max ***/
        for(i=0;i<rlength;++i)
         {
          count=0;
          posmin=35000.0; negmax=-35000.0;
          for(j=0;j<nfiles;++j)
           {
            if(v[j][i] > 0)
             {
              if(v[j][i] < posmin )
               {  posmin = v[j][i];  }
              count++;
             }
            else if(v[j][i] < 0)
             {
              if (v[j][i] > negmax)
               { negmax = v[j][i]; }
              count--;
             }
           }
/* if all the values are +ve then take posmin and if
      all are -ve take negmax else put zero */
           if(count == nfiles)
            thres[i] = posmin;
           else if(count == -nfiles)
            thres[i] = negmax;
           else
            thres[i]=0;
          }
          save_data(thres,rlength,Outfile);
          }
```

Code Listing 3.10 Program fragment that implements Polarity Thresholding and Minimisation

3.2.11 Typical Image Pre-Processing Sequence

In this subsection we take a typical image and subject it to common image pre-processing methods. The following analysis and the Scripts listed herein pertain to the free UTHSCSA ImageTool program (developed at the University of Texas Health Science Center at San Antonio, Texas, USA, and available freely from the Internet by anonymous FTP from ftp://maxrad6.uthscsa.edu).

In order to highlight the pre-processing methods, we describe the following sequence: (a) Consider an x-ray image of a pipe weld that has defects, (b) convert the image to a grayscale image using ImageTool, (c) Threshold the grey-scale manually to highlight the defect, and (d) find the properties of the defect.

The entire process can be automated using Scripts, which are shown below in the Script Listings 3.1 to 3.3.

The "colour" image that we start with is shown in Figure 3.12.

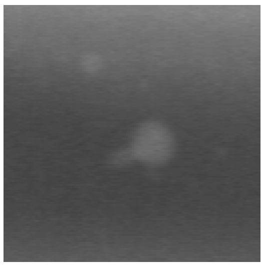


Figure 3.12 "Colour" X-Ray Image of a pipe weld defect

We now transform the image to a grey scale image, which can be done either manually or by using the example Script listed below:

```
macro 'Samples|GrayThresMedian';

var

index, count, no, low, high, limit, step : integer;
filename, txttitle                       : string;

begin
    FOR index:= 1 TO 1 DO BEGIN
    filename := GetOpenFileName('Enter New Image File','BMP','');
    Open(filename);
    PlugIn('C:\\Program_Files\\UTHSCSA\\ImageTool\\Plug-Ins\\makegray.dll');
    Delete(filename,1,78);
    SaveAs(concat('D:\\Raja\\Rough Work\\','G',filename));
    Close;
    END;
end;
```

Script Listing 3.1 Script fragment to convert an image to gray scale image in ImageTool

The gray scale image is shown in Figure 3.13.

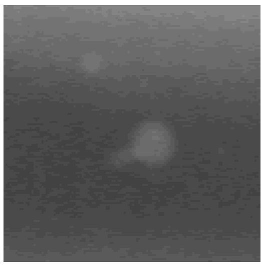


Figure 3.13 Gray scale image of Figure 3.12 (X-ray Image of Defect in a Pipe Weld)

Even though Figure 3.12 and 3.13 look similar, the latter is only a gray scale image. This can be verified by measuring the sizes occupied by these images. Figure 3.12 has a size of 49kB whereas that of Figure 3.13 is just 18kB.

We can now study the properties of the defect artefact (object), by programmatically identifying it and analysing the object. The following example Script converts the image to gray scale and fixes the upper and lower threshold limits to identify the object.

```
macro 'Samples|GrayThresMedian';

var

index, count, no, low, high, limit, step : integer;
filename, txttitle                        : string;

begin
    FOR index:= 1 TO 2 DO BEGIN
    filename := GetOpenFileName('Enter New Image File','BMP','');
    Open(filename);
    PlugIn('C:\\Program_Files\\UTHSCSA\\ImageTool\\Plug-Ins\\makegray.dll');
    SetDensitySlice(76,82);
    PlugIn('C:\\Program_Files\\UTHSCSA\\ImageTool\\Plug-Ins\\median.dll');
    Delete(filename,1,77);
    SaveAs(concat('D:\\Raja\\Rough Work\\',filename));
    Close;
    END;
end;
```

Script Listing 3.2 Script fragment that sets the upper and lower threshold limits

On the basis of the upper and lower threshold limits, the objects present in an image can be isolated for further study. The following figure (Fig.3.14) shows the objects identified in such a manner in Figure 3.13. A total of 55 Objects were identified in this procedure. Note that for a given gray scale image, this number will crucially depend upon the upper and lower threshold limits set in the Script Listing 3.2.

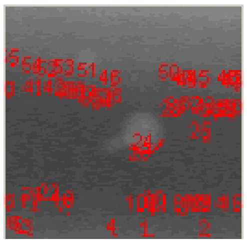


Figure 3.14 Objects identified by the Thresholding Script (Script Listing 3.2) – (the red markings are Object numbers, whose resolution is not clear enough in this screen grab)

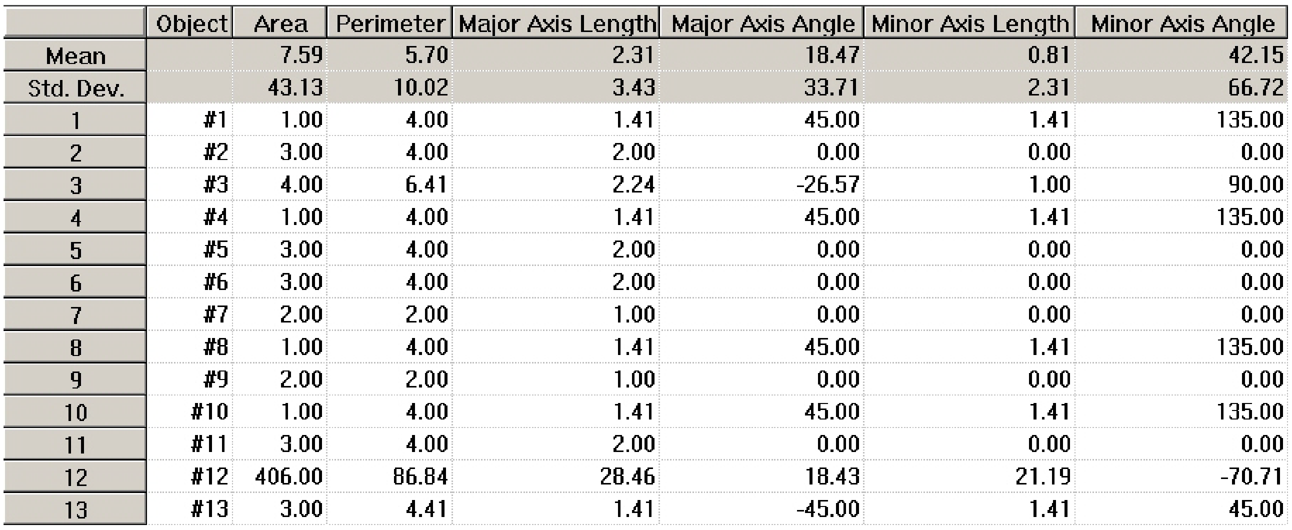
The following example Script fragment shows the implementation of analysing the individual objects identified in an image, by upper and lower threshold settings.

```
macro 'Samples|DSS ObjAnal';

var

index, count, no : integer;
filename         : string;

begin
    FOR index:= 1 TO 100 DO BEGIN
    count := 0;
    filename := GetOpenFileName('Enter New Image File','BMP','');
    Open(filename);
    PlugIn('C:\\Program_Files\\UTHSCSA\\ImageTool\\Plug-Ins\\makegray.dll');
    Delete(filename,1,39);
    SaveAs(concat('D:\\Raja\\My Files\\SAFT Analysis\\','G',filename));
    Delete(filename,12,3);
    SetDensitySlice(76,82);
    FindObjects;
    PlugIn('C:\\Program_Files\\UTHSCSA\\ImageTool\\Plug-Ins\\objanal.dll');
    Close;
    ShowResults(true);
    MakeNewTextWindow;
    Write(GetResults('Area',-2));
    Write(',');
    Write(GetResults('Perimeter',-2));
    Write(',');
    Write(GetResults('Major Axis Length',-2));
    Write(',');
    Write(GetResults('Major Axis Angle',-2));
    Write(',');
    Write(GetResults('Minor Axis Length',-2));
    Write(',');
    Write(GetResults('Minor Axis Angle',-2));
    Write(',');
    WriteLn(GetResults('Int. Dens.',-2));
    Write(GetResults('Area',-1));
    Write(',');
    Write(GetResults('Perimeter',-1));
    Write(',');
    Write(GetResults('Major Axis Length',-1));
    Write(',');
    Write(GetResults('Major Axis Angle',-1));
    Write(',');
    Write(GetResults('Minor Axis Length',-1));
    Write(',');
    Write(GetResults('Minor Axis Angle',-1));
    Write(',');
    WriteLn(GetResults('Int. Dens.',-1));
    no := GetNumber('Total No. of Objects',count);
    FOR count:= 1 TO no DO BEGIN
    Write(GetResults('Object',count));
    Write(',');
    Write(GetResults('Area',count));
    Write(',');
    Write(GetResults('Perimeter',count));
    Write(',');
    Write(GetResults('Major Axis Length',count));
    Write(',');
    Write(GetResults('Major Axis Angle',count));
    Write(',');
    Write(GetResults('Minor Axis Length',count));
    Write(',');
    Write(GetResults('Minor Axis Angle',count));
    Write(',');
    Write(GetResults('Centroid',count));
    Write(',');
    Write(GetResults('Gray Centroid',count));
    Write(',');
    WriteLn(GetResults('Int. Dens.',count));
    END;
    SaveAs(concat('D:\\Raja\\My Files\\SAFT
Analysis\\Data\\','R',filename,'DAT'));
    DiscardResults;
    Closeall;
    END;
end;
```

Script Listing 3.3 Script fragment that finds the Objects present in an image and their Properties

Though not shown fully in the Script Listing 3.3, the following properties can be obtained for each Object in an image: Object Number, Area, Perimeter, Major Axis Length, Major Axis Angle, Minor Axis Length, Minor Axis Angle, Elongation, Roundness, Ferret Diameter, Compactness, Centroid, Gray Centroid, Density, Minimum Level, Maximum Level, Mean, Median, Mode and Standard Deviation.

Figure 3.15 shows a typical listing of Object properties for a few Objects of Figure 3.14:



	Object	Area	Perimeter	Major Axis Length	Major Axis Angle	Minor Axis Length	Minor Axis Angle
Mean		7.59	5.70	2.31	18.47	0.81	42.15
Std. Dev.		43.13	10.02	3.43	33.71	2.31	66.72
1	#1	1.00	4.00	1.41	45.00	1.41	135.00
2	#2	3.00	4.00	2.00	0.00	0.00	0.00
3	#3	4.00	6.41	2.24	-26.57	1.00	90.00
4	#4	1.00	4.00	1.41	45.00	1.41	135.00
5	#5	3.00	4.00	2.00	0.00	0.00	0.00
6	#6	3.00	4.00	2.00	0.00	0.00	0.00
7	#7	2.00	2.00	1.00	0.00	0.00	0.00
8	#8	1.00	4.00	1.41	45.00	1.41	135.00
9	#9	2.00	2.00	1.00	0.00	0.00	0.00
10	#10	1.00	4.00	1.41	45.00	1.41	135.00
11	#11	3.00	4.00	2.00	0.00	0.00	0.00
12	#12	406.00	86.84	28.46	18.43	21.19	-70.71
13	#13	3.00	4.41	1.41	-45.00	1.41	45.00

Figure 3.15 Typical List of Properties for a few Objects in an Image (for Figure 3.14)

These properties of the Objects, can then be used either directly to study them or can be fed to a Decision Support System to arrive at conclusions on these Objects. These aspects would be discussed in detail in Chapter 5.

We have seen just the rudiments of image pre-processing in this Section, though most methods described in this Section would be applicable in a variety of image processing tasks.

3.3 Various Domains of Data Processing

As mentioned earlier, the data pre-processing need not be confined only to the time domain, in which normally the data is acquired. It could be in any other domain, say in the frequency domain or even by expanding the data using the "basis" of the domain and then reducing it, a good example of which could be using principal component analysis. We have seen the use of pre-processing in the frequency domain. We shall now see a more challenging domain, viz., the eigenspace domain for pre-processing, popularly known as principal component analysis.

One of the main objectives of using the principal component analysis is to find the major components, or those "basis" functions that predominantly determine the form of the signal data. One can accomplish this by creating a covariance matrix from a signal record, finding the eigenvalues and eigenvectors of the covariance matrix, and then studying the eigenvalue "spectrum" for further processing.

Consider a set of M 1-d data records, each of equal length N. One data covariance matrix of size N x N is obtained from each 1-d signal. The matrix thus obtained is a real, symmetric matrix, as the values of the 1-d signal are real (physically measured values). The diagonal elements of the matrix thus obtained are closely related to the auto-correlation function (described in Chapter 4) of the corresponding 1-d signal. If a class of M signals are to be considered for generating the data covariance matrix, then, for each 1-d signal in that class the matrix is generated, and the average of M such matrices are computed to obtain the final data covariance matrix. The procedure to obtain the data covariance matrix is listed in a step-by-step fashion, below:

1. Consider a signal [a,b,c,d] where a,b,c,d are points of the signal.
2. Assume that the signal can be represented by a vector **V** (**V** = [a,b,c,d])
3. Find $\mathbf{V}^T$ i.e., the transpose of the vector **V**.
4. Find the matrix M = $\mathbf{V}^T\mathbf{V}$
5. If we have a number of signals, $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$,…..$\mathbf{V}_N$, find all the matrices M_1, M_2, M_3,…..M_N, as described in step 4.
6. Find the matrix M_{AVG} which is the average all the matrices.
7. Find the eigen values and eigen vectors of the matrix M_{AVG}, using any of the standard methods.
8. Analyse the eigenvalues and eigenvectors, dictated by the objectives of pre-processing.

Note that the input 1-d signals to obtain the covariance matrix can be from any domain. Irrespective of the type of inputs, the eigenvalue system treats them as mere numbers – we need to interpret the results keeping in mind the domain of the input 1-d signal.

While using this approach for pre-processing of data, particularly 1-d data, the following observations may be kept in mind:

- This procedure, because of eigenvalue extraction from the data covariance matrix, is computationally intensive
- If we use a single input signal, the eigenvector corresponding to the largest eigenvalue, is exactly equivalent to the input

- If we use a set of input signals belonging to a Class, then the eigenvector of the largest eigenvalue (of the average of the all data covariance matrices) is the "estimate" of the input Class
- This behaviour holds good irrespective of the input data's domain
- The eigenvector of the non-largest eigenvalues does not reflect the input data's behaviour, even if they are positive and non-zero values
- The eigenvalue spectrum changes with to time and phase shifts
- The value of the largest eigenvalue is (obviously) sensitive to the amplification of the input signals – the EV's value increases accordingly.
- Under identical conditions, it is possible to distinguish the signal sources, and hence the defects, from the eigenvalue spectrum itself

The following skeletal code listing gives a method to accomplish this task:

```
void evev(char FI[128], char LF[128], int DType, int Size, int Sort, int *cerr)
{
int i,j,k,r,sl,rlength,nf,nfiles,coun=0;
float *value,mean,**cov,**covar,*eigval,**eigvec;
float avg,min,max,sum=0.0;
char fname[128];
char evalfn[128] = "";
char evecfn[128] = "";
char evcmfn[128] = "";
char extdat[] = ".dat";
char extlst[] = ".lst";
char intstr[5];
char evecfnn[128] = "";

FILE *fp;
FILE *fpn;
FILE *fpr;

if((fp=fopen(LF,"a"))==NULL) {*cerr= 53 ;exit(1);}
fprintf(fp,"%s\n","Opened logfile from within 'EVEV' (eigenvalues & vectors) C
Module");
fprintf(fp,"%s\n","Ready to execute the program");

sl = strlen(FI);
sl = sl - 4;
strncat(evalfn,FI,sl);
strncat(evalfn,"lst",3);
strncat(evalfn,"eval",4);
strncat(evalfn,extdat,4);
strncat(evecfn,FI,sl);
strncat(evecfn,"evec",4);
strncat(evcmfn,FI,sl);
strncat(evcmfn,"evcm",4);
strncat(evcmfn,extdat,4);

if(DType == 58)
{
nfiles=getsflength(FI);
fprintf(fp,"%s%s\n","Length of the input file list obtained from the file ",FI);
fprintf(fp,"%s%d\n","No. of files = ",nfiles);

fpn=fopen(FI,"r");
fscanf(fpn,"%s",&fname);
rlength=getfflength(fname);
fclose(fpn);
```

```
for(i=0;i<rlength;++i)
   {
   for(j=0;j<rlength;++j)
      {
      covar[i][j] = 0.0;
      }
   }
fprintf(fp,"%s\n","Covariance Matrix initialised...");

fpn=fopen(FI,"r");
rewind(fpn);
for(nf=0;nf<nfiles;++nf)
   {
   fscanf(fpn,"%s",&fname);
   value=vector(0,rlength-1);
   load_data(value,rlength,fname);
   rembiasdata(value,rlength);

   /******** finding covariance matrix AA' *******/
   for(i=0;i<rlength;++i)
      {
       for(j=0;j<rlength;++j)
          {
           cov[i][j]=0.0;
           cov[i][j] = (value[i] * value[j]);
          }
      } /* Completed finding Covariance Matrix for ONE file */

   for(i=0;i<rlength;++i)
      {
      for(j=0;j<rlength;++j)
         {
         covar[i][j] = covar[i][j] + cov[i][j];
         }
      } /* Completed adding the just found Covariance Matrix for ONE file */
   } /* Completed finding the Covariance Matrix for all the files */
   fclose(fpn);

   /* finding average covariance matrix */

if((fpr=fopen(evcmfn,"w"))==NULL) {*cerr= 53 ;exit(1);}
fprintf(fp,"%s%s\n","Completed computing the Covaraince Matrix(CM). The CM
values are stored in file ",evcmfn);

for(i=0;i<rlength;++i)
   {
   for(j=0;j<rlength;++j)
      {
      covar[i][j] = covar[i][j] / nfiles;
      /* if(i==j) {covar[i][j] = 0.0;}  Done as a Test. To be Removed or
commented out later */
      fprintf(fpr,"%d%s%d%s%f\n",i,"  ",j,"  ",covar[i][j]);
      }
   }
fclose(fpr);
}
else
{

fpn=fopen(FI,"r");
for(i=0;i<Size;++i)
   {
```

```
   for(j=0;j<Size;++j)
      {
      fscanf(fpn,"%f",&covar[i][j]);
      }
   }
fprintf(fp,"%s%s\n","Covariance Matrix read from the input file ",FI);
fclose(fpn);
}

/********** Set 'rlength' equal to 'Size', if the covriance matrix itself is
input as data  *******/
if(DType == 57)
  {
  rlength = Size;

/********  Storing the Covariance Matrix in a separate file  ********/
if((fpr=fopen(evcmfn,"w"))==NULL) {*cerr= 53 ;exit(1);}
fprintf(fp,"%s%s\n","Completed reading the Covaraince Matrix(CM). The CM values
are stored again in the file ",evcmfn);

for(i=0;i<rlength;++i)
   {
   for(j=0;j<rlength;++j)
      {
      fprintf(fpr,"%d%s%d%s%f\n",i,"  ",j,"  ",covar[i][j]);
      }
   }
fclose(fpr);
  }

/*** calculate eigen values & eigen vectors$ ***/
/* Use the covariance matrix ‘covar’, ‘rlength’ defines the size of the
   covariance matrix; the eigenvalues that are found are stored in ‘eigval’,
   and the eigenvectors found are stored in ‘eigvec’ */


/*** if(Sort == 55) Sort the eigenvlues and eigenvectors if requested  ***/


if((fpr=fopen(evalfn,"w"))==NULL) {*cerr= 53 ;exit(1);}
fprintf(fp,"%s\n","Completed finding the eigenvalues and eigenvectors.");
fprintf(fp,"%s%s\n","The eigenvalues are stored in file ",evalfn);
fprintf(fp,"%s%s\n","The eigenvectors are stored in file ",evecfn);

/* Finding the prominent Eigen values */

for(i=0;i<rlength;++i)
   {
   fprintf(fpr,"%f\n",eigval[i]);
   }
fclose(fpr);



for(i=0;i<rlength;++i)
   {
   itoa(i, intstr, 10);
   strcpy(evecfnn,evecfn);
   strncat(evecfnn,intstr,4);
   strncat(evecfnn,extdat,4);
   if((fpr=fopen(evecfnn,"w"))==NULL) {*cerr= 53 ;exit(1);}
   for(j=0;j<rlength;++j)
      {
      fprintf(fpr,"%f\n",eigvec[j][i]);
```

```
        }
    fclose(fpr);
    }
fclose(fp);


}
```

Code Listing 3.11 Program fragment to find the covariance matrix in order to find the eigenvalues and eigenvectors

3.4 Questions

- Are there any real-life situations where data pre-processing is **not** necessary?
- What types of data pre-processing do we human beings perform? How effective are they?
- Are there other ways to normalise a data record, without loss of information?
- Why can't we avoid windowing by acquiring an infinite sequence of data?
- Can we think of the eigenvectors of a data's covariance matrix as its "basis"?
- If we find the eigenvectors / eigenvalues of a pure sine wave, how will they look like?
- Many of the data pre-processing methods discussed, have assumed that the signal data is ergodic and stationary. Why is this assumption required?
- Can you give some real-life examples of non-stationary signals?
- For a 2-d data (e.g., a colour image), what additional pre-processing issues need be considered? Why?

[#] It is assumed that the signals are ergodic and stationary.

$ See, for example, W.H.Press, S.A.Teukolsky, W.T.Vetterling and B.P.Flannery, "Numerical Recipes in C", Second Edition, Cambridge University Press, 1992.

** Readers may check for the applicability and completeness of the Code before using them; Scripts require the ImageTool Software (developed at the University of Texas Health Science Center at San Antonio, Texas, USA, and available freely from the Internet by anonymous FTP from ftp://maxrad6.uthscsa.edu).

++ Usage of Code and other material from this Monograph is governed by the Disclaimer found at http://deskpack.tripod.com/disclaim.html