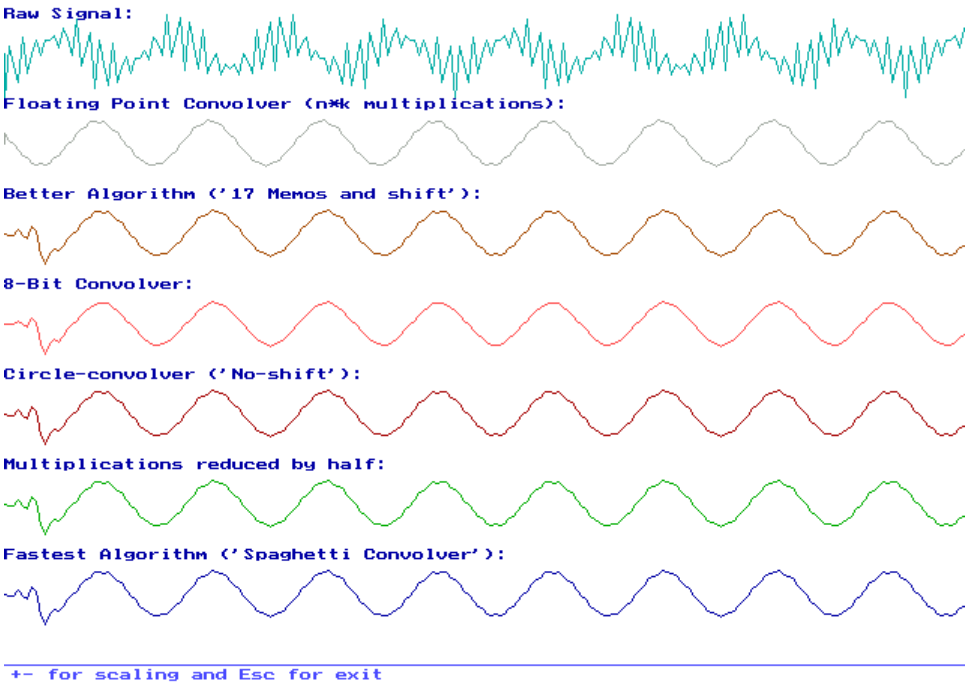
Lab-Report
Digital Signal Processing

FIR-Filters

Model of a fixed and floating point convolver



Name:	Dirk Becker
Course:	BEng 2
Group:	A
Student No.:	9801351
Date:	30/10/1998

UNIVERSITY of EAST LONDON

1. Contents

1. CONTENTS 2

2. INTRODUCTION 3

3. FINITE IMPULSE RESPONSE (FIR) FILTERS 3

a) Definition of a FIR Filter 3

b) Convolution 4

4. CONVOLVER ALGORITHMS 5

a) Standard Shifting Floating Point Convolver 5

b) 8-Bit fixed point convolver 7

c) Convolver with half multiplications 8

d) Line by line Convolver 9

e) Circular - Convolver - Convolver without shifting 11

2. Introduction

Digital Signal Processing (DSP) is the processing of digital signals or the digital processing of signals, which means the same. Fast microprocessors, especially designed for "number crunching" are getting cheaper and cheaper and make DSP a new very important part of engineer's work.
A filter is a system, that is designed to remove some component or modify some charecteristics of a signal. With digital systems almost ideal filters can be realised.

3. Finite impulse response (FIR) Filters

a) Definition of a FIR Filter

FIR Filters are systems for which each output is a weighted average of a finite number of samples of the input sequence.
Following equation defines the causal FIR filter. Causal means, that only positive numbers for k are allowed, so the output depends only on the present and previous values of the input.
The output doesn't change before the input changes from zero.

$$y(n) = \sum_{k=0}^{N-1} h_k x(n-k) \quad \text{where } n = \text{impulse response duration}$$

The transfer function of a FIR filter H(z) is given by:

$$H(z) = \sum_{n=0}^{N-1} h(n) z^{-n}$$

The multiplication by z^{-1} means a delay of 1 in the time domain.
The lowpass filter which was to simulate in the Lab can be described by the equation:

$$y(n) = \sum_{k=0}^{N-1} h(k) x(n-k) \quad \text{where } h(k) \text{ are the filter coefficents and } x(n-k) \text{ the sampled input values}$$

The Filter coefficients h[k] were given by h[n]={0.0234, 0.0267, -0.0505, 0.0, 0.0757, -0.0624,-0.0935, 0.3027, 0.6, 0.3027, -0.0935, -0.0624, 0.0757, 0.0, -0.0505, 0.0267, 0.0234}

For simulation of the digital filter as a behavioural model the following function was to filter:

$x[n]=\cos(2\pi 0.04n)+ \cos(2\pi 0.35n)+ \cos(2\pi 0.4n)$

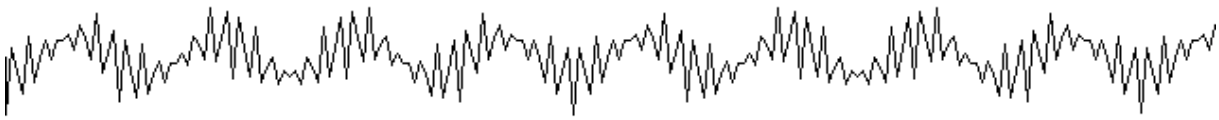


figure 1

Figure 1 shows a plot of the function which was to filter. It consists of three cosines with different radian frequencies.

b) Convolution

Coming from the global definition of the low pass filter we can say:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) = h(k) * x(n) \text{ where } * \text{ means convolution}$$

$$y(n) = h(0)x(n) + h(1)x(n-1) + h(2)x(n-2) + \cdots + h(N-1)n(n-N+1)$$

The output is just a linear weighted sum of present and past inputs. So the FIR filter is called a "Running average filter".

The filter function can be described by the following block diagram, where z^{-1} represents the unit delay:

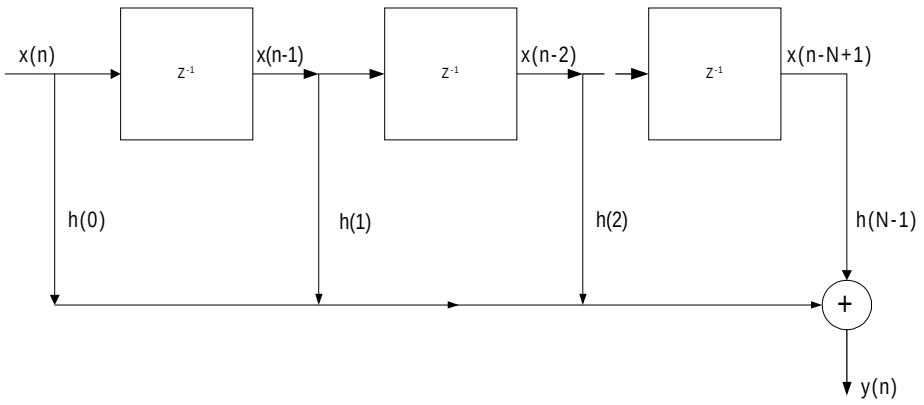


figure 2

The input x(n) is multiplied by the coefficient h(n). Then x(n) is delayed by one step and multiplied with the next coefficient h(n+1). After that x(n) is delayed again and multiplied with the next filter coefficient h(n+2). All this single multiplications are added together and one run of an x(n) value trough all the filter coefficients (here 17) results in one output y(n). So it takes at least 16 steps, till the filter is on correct work. It's the time which the 1st value needs to come to the output of the filter.
This filter method is done by means of a convolver, which stores and shifts the x-values and multiplies them by the correct filter coefficients.
Figure 2 shows the standard block diagram of a transversal filter.

4. Convolver Algorithms

The convolver function can be done via very different algorithms. The main difference between every algorithm is the way of shifting (delaying) and multiplying the different values. As a result of this the efficiency of the algorithms can be very dissimilar.

a) Standard Shifting Floating Point Convolver

First convolver to realise was a standard shifting convolver, which exactly works like the block diagram of figure 2.
The convolver is realised by means of a C-function. The h(n) filter coefficients are handed over by the main program. The other parameters are only for optical improvements and adjustments like ypos gives the absolute position of the function on the screen, scale is the scale factor in horizontal position (zooming) and colour is the colour of the filtered function.
First part of the convolver function is to initialise all the 17 memories (xstore[n]) for the convolver.
Next loop is the main counter loop for the simulation function x(n). It counts from 0 to n_{max} (right end of screen). It's only for producing new values of x(n).
The main convolver implementation is done in the 2nd loop which uses k as loop counter. The actual values of the stored x-value and the filter coefficients are multiplied and added to the accumulator y_output. After each calculation the stored x-values are shifted down by one, the oldest one is getting lost (xstore[0]) and at the end of the loop a new value of x is written into the newest memory (xstore[16]).
Following figure shows the function of the convolver better than an explanation:

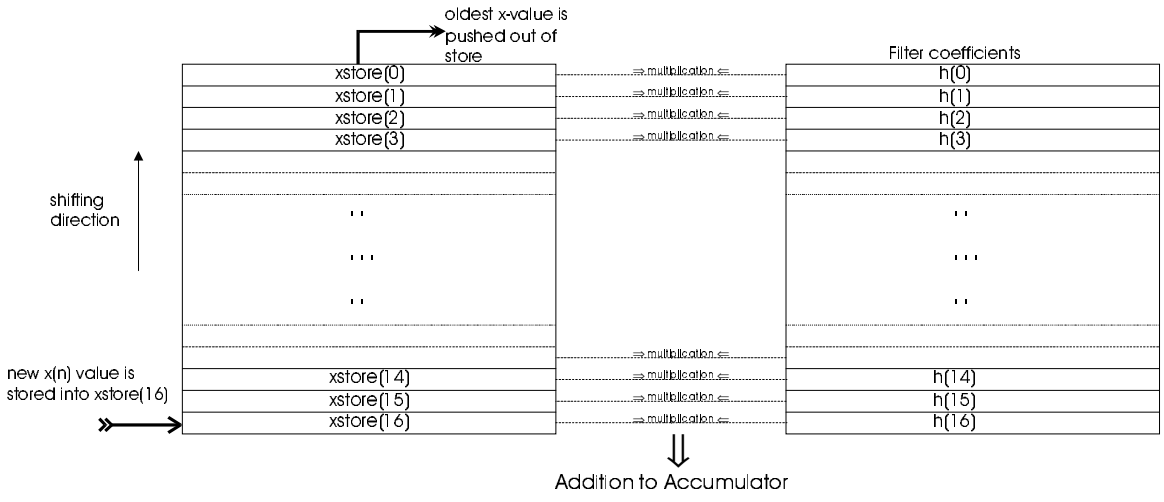


figure 3 Working model of a convolver

The following listing is the code of the above explained standard shifting convolver.

```
/* ------------------------------------------------------------ */
/* ---------- Shifting Convolver ------------------------------ */
void shifting_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k;
     float xstore[20], xvalue, y_output;
     setcolor(WHITE);
     outtextxy(0, ypos-30, "Better Algorithm ('17 Memos and shift'):");
     moveto(0,ypos);

     for(n=0; n<=16; n++)  /* initialising array of x(n) */
      {
        xstore[n]=0;
      }
     setcolor(colour);

     for(n=0; n<=(end/scale); n++)     /* loop for n */
      {

      y_output=0;
      for(k=0; k<=16; k++)                 /* convolver loop */
          {
            y_output+=xstore[k]*h[k];    /* convolving */
            xstore[k]=xstore[k+1];       /* shifting array */
          }
      lineto(n*scale,y_output*15+ypos);   /* drawing output  */
      xstore[16]=x(n);    /* writing new x(n) value */
      }
}
```

figure 4

Figure 4 shows the output of the simulated filter. It can be easily seen, that two of the three cosine oscillations have been removed by means of the FIR filter → Lowpass filtering function.. The higher frequencies are removed.
First part of the plot shows the duration till all the convolver stores are filled and the convolver starts it's filtering work.

b) 8-Bit fixed point convolver

Next part of the lab was to simulate an 8 Bit fixed point convolver. The working model of the convolver didn't need any change, but the variables of the stored x-values and the variables for the coefficients had to be converted into 8-Bit integers (in C named "signed char").
As accumulator (summation of the multiplied x and h values) an usual 16-bit integer was chosen.
For the output this 16-bit integer must be changed into in 8-bit integer, because most Digital to Analogue Converters (DAC) are 8-bit converters. Therefor the 16-bit output of the convolver is divided by 128 by right shifting of 7 Bits (2^7=128).
First part of the program changes the floating point parameters of h(n) into their 8-bit equivalents called h_8bit(n) by multiplying each of them by 127. Same loop initialises the 8-bit variables for storing the different x(n) values.
In the main loop the new xstore values are multiplied by (127/3) to get a maximum value of 127 (signed char: -128..127 max. values).

```
/* ------------------------------------------------------------ */
/* ---------- Shifting 8-Bit Convolver ---------------------- */
void convolver_8bit(int ypos, int scale, int colour, float *h)
{
   int n, y_output;      /* n and accumulator are aigned integers */
   signed char xstore[17], k, h_8bit[17], bit=127, y_dac;
   setcolor(WHITE);                        /* optics */
   outtextxy(0, ypos-30, "8-Bit Convolver:");
   moveto(0,ypos);
   for(n=0; n<=16; n++)  /* initalising and converting h(n) to 8-Bit int */
         {
         xstore[n]=0;
         h_8bit[n]=h[n]*bit;
    }
   setcolor(colour);
   for(n=0; n<=(end/scale); n++)         /* loop for n */
    {

         y_output=0;
         for(k=0; k<=16; k++)             /* convolver loop */
            {
                  y_output+=(xstore[k]*h_8bit[k]); /* convolving into accu */
                  xstore[k]=xstore[k+1];          /* shifting            */
            }
         y_dac=y_output>>7;       /* converting accu to 8-bit for DAC */
         lineto(n*scale,(3*15*y_dac)/(bit) +ypos); /* Drawing & scaling */
         xstore[16]=(bit/3)*x(n);      /* storing new  x(n) value */
                                       /* max value=127          */
     }
}
```

figure 5

Figure 5 shows the output plot of the 8-Bit convolver.
The result is the of course same than the output of the other convolvers.
Opportunity of the 8-bit convolver is the short calculation time, and it is the nearest to a real hardware filter, because digital signal processors with more than 8-bit are very expensive and not commonly used.
The 8-bit convolver does nearly the same work than the others, but with much less calculation expenditure.

c) Convolver with half multiplications

In the above convolver algorithms the output sum is done by multiplying every x(n) by the h-coefficient belonging to it. The number of multiplications can be reduced by half.
The first multiplication in the convolver loop is h(0) multiplied with x(0), the last multiplication is h(16) with x(16) like shown in the following figure.

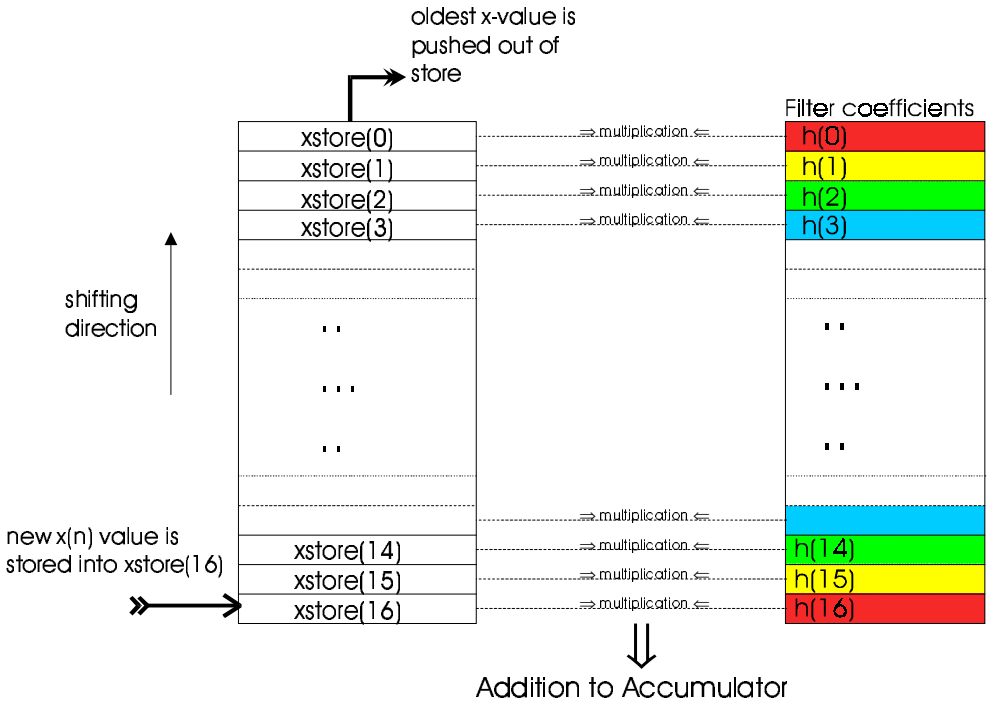


figure 6

The h(n) parameters are symmetrical to the middle value h(8). For reducing the number of multiplications the values of x(k) and x(16-k) can be added and multiplied with only one h coefficient.

Proof of reducing multiplications:
y(n)=x(0)h(0)+x(1)h(1)+x(2)h(2) +...+ x(14)h(14)+x(15)h(15)+x(16)h(16)
y(n)=x(0)h(0)+x(1)h(1)+x(2)h(2) +...+ x(14)h(2) +x(15)h(1) +x(16)h(0)
y(n)=h(0)[x(0)+x(16)]+h(1)[x(1)+x(15)]+h(2)[x(2)+x(14)]+...+h(8)[x(8)+x(8)]

So the 16 multiplications are reduced to 8 multiplications and 8 additions.
The middle coefficient h(8) has to be divided by 2 or summed after the convolver loop, in order to calculate this value only once, because the number of coefficients is odd.
The shifting (delaying) of the stored x-values is done by means of a separate loop.
It can be done in the same loop, but is more complicated because 2 different values have to be stored and tit shouldn't be shown here.

The output of the filter is printed in the following plot:

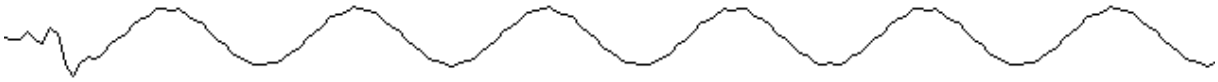


figure 7

There is no difference in the output to the other convolvers, but the duration time is shorter, because adding is much faster than multiplying.

d) Line by line Convolver

Another way to increase the speed of the convolver is to replace the convolver loop by discrete written code. One for each loop run means 34 different statements, 17 for shifting, 17 for convolving.
Rest of the function works like the standard shifting convolver.
Listing of the discrete written convolver:

```
/* ------------------------------------------------------------ */
/* ---------- Spaghetti Convolver ------------------------------ */
void spaghetti_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k;
     float xstore[20], xvalue, y_output;
     setcolor(BLUE);
     outtextxy(0, ypos-30, "Fastest Algorithm ('Spaggetti Convolver'):");
     moveto(0,ypos);

     for(n=0; n<=16; n++)  /* initialising array of x(n) */
      {
        xstore[n]=0;
      }
     setcolor(colour);
     for(n=0; n<=(end/scale); n++)
      {

      y_output=0;

      /* Spaghetti - convolver repeats 17 times same function */

         y_output+=xstore[0]*h[0];    /* convolving */
         xstore[0]=xstore[1];       /* shifting array */
         y_output+=xstore[1]*h[1];    /* convolving */
         xstore[1]=xstore[2];       /* shifting array */
         y_output+=xstore[2]*h[2];    /* convolving */
         xstore[2]=xstore[3];       /* shifting array */
         y_output+=xstore[3]*h[3];    /* convolving */
         xstore[3]=xstore[4];       /* shifting array */
```

```
            y_output+=xstore[4]*h[4];    /* convolving */
            xstore[4]=xstore[5];       /* shifting array */
            y_output+=xstore[5]*h[5];    /* convolving */
            xstore[5]=xstore[6];       /* shifting array */
            y_output+=xstore[6]*h[6];    /* convolving */
            xstore[6]=xstore[7];       /* shifting array */
            y_output+=xstore[7]*h[7];    /* convolving */
            xstore[7]=xstore[8];       /* shifting array */
            y_output+=xstore[8]*h[8];    /* convolving */
            xstore[8]=xstore[9];       /* shifting array */
            y_output+=xstore[9]*h[9];    /* convolving */
            xstore[9]=xstore[10];       /* shifting array */
            y_output+=xstore[10]*h[10];    /* convolving */
            xstore[10]=xstore[11];       /* shifting array */
            y_output+=xstore[11]*h[11];    /* convolving */
            xstore[11]=xstore[12];       /* shifting array */
            y_output+=xstore[12]*h[12];    /* convolving */
            xstore[12]=xstore[13];       /* shifting array */
            y_output+=xstore[13]*h[13];    /* convolving */
            xstore[13]=xstore[14];       /* shifting array */
            y_output+=xstore[14]*h[14];    /* convolving */
            xstore[14]=xstore[15];       /* shifting array */
            y_output+=xstore[15]*h[15];    /* convolving */
            xstore[15]=xstore[16];       /* shifting array */
            y_output+=xstore[16]*h[16];    /* convolving */

         /*  --- END of spaghetti convolver */

         lineto(n*scale,y_output*15+ypos);
         xstore[16]=x(n);    /* writing new x(n) value */
         }
}
```

The screen output, which means the filtered signal is of course the same again.

e) Circular - Convolver - Convolver without shifting

Last way to improve the speed of the convolver is to reduce or avoid the shifting of the array with the stored x(n) values. This can be done by means of a circular convolver.
The stores for the x-values are stored in a circular array. Only the start (=end) position of this array is to be known. Now instead of shifting the array, only the starting point of the circular array is increased by one. This results in the same procedure like shifting the linear array by one. Automatically the x-values are multiplied with the NEXT coefficients.

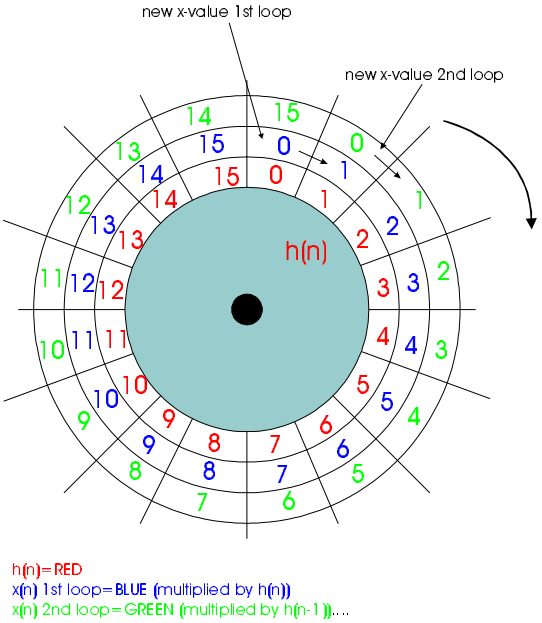


figure 8

Figure 8 shows the main function of the circular convolver.

The code of the circular convolver is the most complex because the start and end points of the circular array have to be converted into a conventional linear array. So the starting and end point have to be stored in special variables because the end point of the linear array is not the end point of the circular array.

Code of the circular convolver function:

```
/* ------------------------------------------------------------ */
/* ---------- Circle Convolver ------------------------------- */
void circle_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k, i=0, j=0;
     float xstore[20], xvalue, y_output;
     setcolor(BLUE);
     outtextxy(0, ypos-30, "Circle-convolver ('No-shift'):");
     moveto(0,ypos);
     for(n=0; n<=16; n++) /* initalising array */
      {
        xstore[n]=0;
      }
     setcolor(colour);
     for(n=0; n<=(end/scale); n++)
```

```
      {
      y_output=0; /* initialising output      */
      k=i;j=0;    /* convolver start position */
      do {        /* convolver loop           */

              y_output+=xstore[k]*h[j]; /* convolving         */
              k++;j++;                  /* increasing counter */
              if (k>=17) {k=0;};        /* circling (k)       */

          } while(k!=i);
      lineto(n*scale,y_output*15+ypos);
      xstore[i]=x(n);
      i++;                  /* circle start (i)        */
      if(i>=17) {i=0;};     /*     and resetting start */
      }
}
```

5. Conclusion

The simulation of a FIR filter in C shows the different possibilities of realising the same function. The simulation shows the different ways how to optimise a code. Either in direction of speed, or in direction of small code size.
Fastest code in this case would be an 8-bit convolver with half multiplication written in “line by line” code without loops. The filter which is the closest one to a real hardware realisation is the simulation of the 8-Bit converter. Cheap and fast Analogue to Digital and Digital to Analogue Converters are mostly 8-bit converters. So the input and output of the filter has to be realised by means of 8-bit functions

f) Complete Listing

The different function are put together in one listing. This makes it easier to compare the efficiency of the different codes and the outputs of the various functions can be displayed together.
It can be easily obtained, that the output of the 8-bit convolver is only NEARLY the same than the output of the other convolvers.

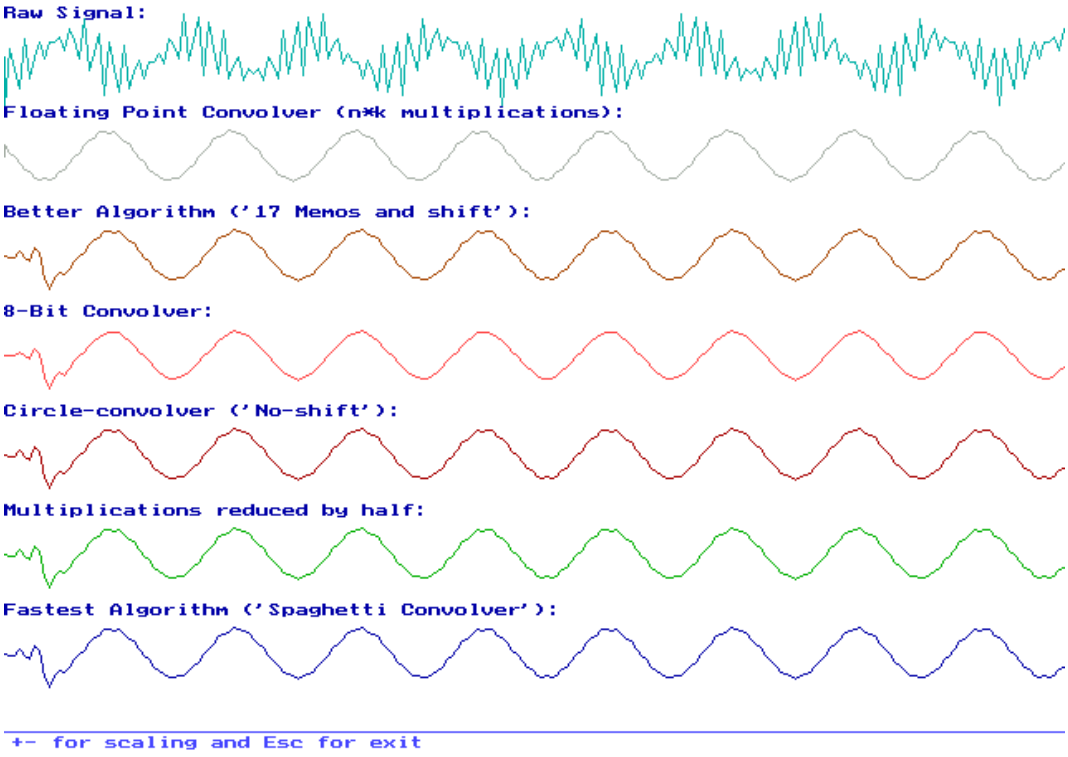


figure 9

Figure 9 shows the output of all the convolver simulations, the listing is printed on the following pages.

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>


#define pi 3.14159
#define end 640


float x(int n);
void std_convolver(int ypos, int scale, int colour, float *);
void shifting_convolver(int ypos, int scale, int colour, float *);
void convolver_8bit(int ypos, int scale, int colour, float *);
void half_convolver(int ypos, int scale, int colour, float *);
void circle_convolver(int ypos, int scale, int colour, float *);
void spaghetti_convolver(int ypos, int scale, int colour, float *);
void signraw(int ypos, int scale, int colour);


int main(void)
{
   /* request autodetection */
   int gdriver = DETECT, gmode, errorcode, scale=3;
   char key;
   /* filter coefficients */
   float h[17]={0.0234, 0.0267, -0.0505, 0.0, 0.0757, -0.0624,
            -0.0935, 0.3027, 0.6, 0.3027, -0.0935, -0.0624,
             0.0757, 0.0, -0.0505, 0.0267, 0.0234};
   /* initialize graphics and local variables */
   initgraph(&gdriver, &gmode, "");
   do {
      /*  start convolvers */
      setbkcolor(WHITE);
      signraw(40, scale, 19);
      std_convolver(100, scale, 7, &h[0]);
      shifting_convolver(160, scale, 6, &h[0]);
      convolver_8bit(220, scale, LIGHTRED, &h[0]);
      circle_convolver(280, scale, RED, &h[0]);
      half_convolver(340, scale, GREEN, &h[0]);
      spaghetti_convolver(400, scale, BLUE, &h[0]);
      setcolor(LIGHTBLUE);
      moveto(0,447);lineto(639,447);
      outtextxy(5,450,"+- for scaling and Esc for exit");
      key=getch();
      if (key=='+') {scale++;};
      if (key=='-') {scale--;};
      if (scale<=0) {scale=1;};
       cleardevice();
      } while (key!=27);

   /* closegraph (clean up) */
   closegraph();
   return 0;
}

/* ---------- END OF MAIN FUNCTION -------------------------- */
```

```
/* ------------------------------------------------------------ */
/*----------- Definition of x(n) ---------------------------- */
float x(int n)
{
   float x;
   x=cos(2*pi*0.04*n)+cos(2*pi*0.35*n)+cos(2*pi*0.4*n);
   return(x);
}

/* ------------------------------------------------------------ */
/* ---------- Standart Convolver ---------------------------- */
void std_convolver(int ypos, int scale, int colour, float *h)
{
    int n, k;
    float xvalue, y_output;
    setcolor(BLUE);
    outtextxy(0, ypos-30, "Floating Point Convolver (n*k
multiplications):");
    moveto(0,ypos);
    setcolor(colour);

    for(n=0; n<=(end/scale); n++)
     {
      y_output=0;
      for(k=0; k<=16; k++)
          {
            y_output+=h[k]*x(n-k);     /* convolving */
          }
      lineto(n*scale,y_output*15+ypos);
     }

}

/* ------------------------------------------------------------ */
/* ---------- Shifting Convolver ---------------------------- */
void shifting_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k;
     float xstore[20], xvalue, y_output;
     setcolor(BLUE);
     outtextxy(0, ypos-30, "Better Algorithm ('17 Memos and shift'):");
     moveto(0,ypos);

     for(n=0; n<=16; n++)  /* initialising array of x(n) */
      {
        xstore[n]=0;
      }
     setcolor(colour);

     for(n=0; n<=(end/scale); n++)     /* loop for n */
      {

      y_output=0;
      for(k=0; k<=16; k++)                 /* convolver loop */
          {
            y_output+=xstore[k]*h[k];    /* convolving */
            xstore[k]=xstore[k+1];       /* shifting array */
          }
      lineto(n*scale,y_output*15+ypos);   /* drawing output  */
      xstore[16]=x(n);    /* writing new x(n) value */
      }
}
```

```
/* ------------------------------------------------------------ */
/* ---------- Convolver with half multiplications------------- */
void half_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k;
     float xstore[20], xvalue, y_output;
     setcolor(BLUE);
     outtextxy(0, ypos-30, "Multiplications reduced by half:");
     moveto(0,ypos);

     for(n=0; n<=16; n++)   /* initialising array  */
      {
        xstore[n]=0;
      }
     setcolor(colour);
     h[8]/=2;        /* no double multiplication of middle coefficient  */
     for(n=0; n<=(end/scale); n++)
      {

      y_output=0;

      for(k=0; k<=8; k++)
          {
            y_output+=(h[k]*(xstore[k]+xstore[16-k])); /* convolving */
          }
       lineto(n*scale,y_output*15+ypos);
       for(k=0; k<=15; k++)    /* shifting loop */
          {
             xstore[k]=xstore[k+1];
          }
        xstore[16]=x(n);    /* writing new x(n) value */
      }
      h[8]*=2;   /* Setting old value of middle coefficient again*/
}
/* ------------------------------------------------------------ */
/* ---------- Circle Convolver ------------------------------- */
void circle_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k, i=0, j=0;
     float xstore[20], xvalue, y_output;
     setcolor(BLUE);
     outtextxy(0, ypos-30, "Circle-convolver ('No-shift'):");
     moveto(0,ypos);
     for(n=0; n<=16; n++) /* initalising array */
      {
        xstore[n]=0;
      }
     setcolor(colour);
     for(n=0; n<=(end/scale); n++)
      {
      y_output=0; /* initialising output      */
      k=i;j=0;    /* convolver start position */
      do {        /* convolver loop           */

              y_output+=xstore[k]*h[j]; /* convolving         */
              k++;j++;                  /* increasing counter */
              if (k>=17) {k=0;};        /* circling (k)       */

          } while(k!=i);
      lineto(n*scale,y_output*15+ypos);
      xstore[i]=x(n);
      i++;                  /* circle start (i)       */
```

```
      if(i>=17) {i=0;};     /*     and resetting start */
      }
}

/* ------------------------------------------------------------ */
/* ---------- Shifting 8-Bit Convolver ---------------------- */
void convolver_8bit(int ypos, int scale, int colour, float *h)
{
    int n, y_output;      /* n and accumulator are aigned integers */
    signed char xstore[17], k, h_8bit[17], bit=127, y_dac;
    setcolor(BLUE);                              /* optics */
    outtextxy(0, ypos-30, "8-Bit Convolver:");
    moveto(0,ypos);
    for(n=0; n<=16; n++)  /* initalising and converting h(n) to 8-Bit int
*/
      {
      xstore[n]=0;
      h_8bit[n]=h[n]*bit;
     }
    setcolor(colour);
    for(n=0; n<=(end/scale); n++)          /* loop for n */
     {

      y_output=0;
      for(k=0; k<=16; k++)              /* convolver loop */
          {
            y_output+=(xstore[k]*h_8bit[k]); /* convolving into accu */
            xstore[k]=xstore[k+1];           /* shifting             */
          }
      y_dac=y_output>>7;       /* converting accu to 8-bit for DAC */
      lineto(n*scale,(3*15*y_dac)/(bit) +ypos); /* Drawing & scaling */
      xstore[16]=(bit/3)*x(n);      /* storing new  x(n) value */
                             /* max value=127           */
      }


}
/* ------------------------------------------------------------ */
/* ---------- Spaghetti Convolver ---------------------------- */
void spaghetti_convolver(int ypos, int scale, int colour, float *h)
{
     int n, k;
     float xstore[20], xvalue, y_output;
     setcolor(BLUE);
     outtextxy(0, ypos-30, "Fastest Algorithm ('Spaggetti Convolver'):");
     moveto(0,ypos);

     for(n=0; n<=16; n++)  /* initialising array of x(n) */
      {
        xstore[n]=0;
      }
     setcolor(colour);
     for(n=0; n<=(end/scale); n++)
      {

      y_output=0;

      /* Spaghetti - convolver repeats 17 times same function */

         y_output+=xstore[0]*h[0];    /* convolving */
         xstore[0]=xstore[1];       /* shifting array */
         y_output+=xstore[1]*h[1];    /* convolving */
         xstore[1]=xstore[2];       /* shifting array */
```

```
          y_output+=xstore[2]*h[2];    /* convolving */
          xstore[2]=xstore[3];       /* shifting array */
          y_output+=xstore[3]*h[3];    /* convolving */
          xstore[3]=xstore[4];       /* shifting array */
          y_output+=xstore[4]*h[4];    /* convolving */
          xstore[4]=xstore[5];       /* shifting array */
          y_output+=xstore[5]*h[5];    /* convolving */
          xstore[5]=xstore[6];       /* shifting array */
          y_output+=xstore[6]*h[6];    /* convolving */
          xstore[6]=xstore[7];       /* shifting array */
          y_output+=xstore[7]*h[7];    /* convolving */
          xstore[7]=xstore[8];       /* shifting array */
          y_output+=xstore[8]*h[8];    /* convolving */
          xstore[8]=xstore[9];       /* shifting array */
          y_output+=xstore[9]*h[9];    /* convolving */
          xstore[9]=xstore[10];       /* shifting array */
          y_output+=xstore[10]*h[10];    /* convolving */
          xstore[10]=xstore[11];       /* shifting array */
          y_output+=xstore[11]*h[11];    /* convolving */
          xstore[11]=xstore[12];       /* shifting array */
          y_output+=xstore[12]*h[12];    /* convolving */
          xstore[12]=xstore[13];       /* shifting array */
          y_output+=xstore[13]*h[13];    /* convolving */
          xstore[13]=xstore[14];       /* shifting array */
          y_output+=xstore[14]*h[14];    /* convolving */
          xstore[14]=xstore[15];       /* shifting array */
          y_output+=xstore[15]*h[15];    /* convolving */
          xstore[15]=xstore[16];       /* shifting array */
          y_output+=xstore[16]*h[16];    /* convolving */

        /*  --- END of spaghetti convolver */

        lineto(n*scale,y_output*15+ypos);
        xstore[16]=x(n);    /* writing new x(n) value */
        }
}
/* ------------------------------------------------------------ */
/* ---------- Draw raw function x(n)------------------------- */
void signraw(int ypos, int  scale, int colour)
{
   float xvalue;
   int n;
   setcolor(BLUE);
   outtextxy(0, ypos-30, "Raw Signal:");
   moveto(0,ypos);
   setcolor(colour);
   for(n=0; n<=(end/scale); n++)
    {
       xvalue=x(n);   /* calculating new value */
       lineto(n*scale,xvalue*10+ypos);
    }
}
```