Imperial College London

Department of Electrical and Electronic Engineering Imperial College London

Real-time digital signal processing

Report on "*Real time implementation of IIR filters*" (laboratory 5)

22 pages total

Report written by:

Quentin McGaw,	CID 00746622	Username QDM12
Alexandra Rouhana,	CID 00736752	Username AR4412

1. Designing a discrete time single-pole filter

1.1. Objective

The analogue filter required to design in a discrete time version is shown below.

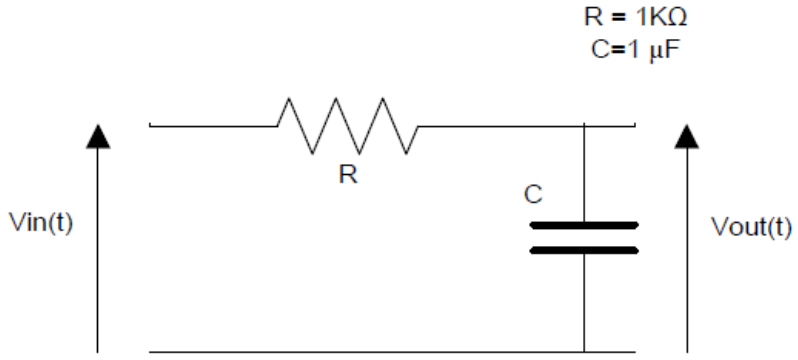

Figure 1.1: Analogue filter circuit diagram

This is a low-pass single-pole filter, with a time constant of $\mathrm{RC} = 10^{3} \times 10^{-6} = 10^{-3} = 1\ ms$

1.2. Laplace domain transfer function

The Laplace domain transfer function of this filter can be obtained by analysing the analogue circuit with Kirchhoff current laws. The following relationship can hence be obtained:

$$\frac{\mathrm{Vout} - 0}{(sC)^{-1}} + \frac{\mathrm{Vout} - \mathrm{Vin}}{\mathrm{R}} = 0$$

$$\Rightarrow \mathrm{R} \times \mathrm{Vout} + (sC)^{-1} \times (\mathrm{Vout} - \mathrm{Vin}) = 0$$

$$\Rightarrow \mathrm{Vout} \times (\mathrm{R} + (sC)^{-1}) = \mathrm{Vin} \times (sC)^{-1}$$

$$\Rightarrow \frac{\mathrm{Vout}}{\mathrm{Vin}} = \frac{(sC)^{-1}}{(\mathrm{R} + (sC)^{-1})}$$

$$\Rightarrow \frac{\mathrm{Vout}}{\mathrm{Vin}} = \frac{1}{(sRC + 1)}$$

With R = 1 KΩ and C = 1 µF, the transfer function of this filter is given by:

$$\mathrm{H(s)} = \frac{1}{(0.001s + 1)}$$

This transfer function has a single pole at s = - 1000, and the cut off frequency is $\omega_0 = 1000\ rad/s$ or $f_0 = 159\ Hz$.

1.3. Z-domain transfer function

The Z-domain is the discrete time equivalent of the Laplace domain.
To obtain the system Z-domain transfer function H(z) from the Laplace domain transfer function H(s) previously found, the following method is used.
Let T_s be the sampling time, so we have:

$$\mathrm{z} = e^{sT_s}$$
$$\Rightarrow \mathrm{s} = \frac{1}{T_s} \times \ln(z)$$

According to the Taylor series, the natural logarithm of z is given by:

$$\ln(z) = 2 \times \left[\frac{z-1}{z+1} + \frac{1}{2}\left(\frac{z-1}{z+1}\right)^2 + \frac{1}{3}\left(\frac{z-1}{z+1}\right)^3 + \cdots\right]$$

Approximating the natural logarithm of z to $2 \times \frac{z-1}{z+1}$ the following is obtained:

$$\mathrm{s} = \frac{2}{T_s} \times \frac{z-1}{z+1}$$

This relationship is better known as the **Tustin transform** (or *bilinear transform*) and allows to transform a Laplace domain transfer function to a Z-domain transfer function.
In our case, the sampling frequency of the discrete filter will be 8 KHz, so the sampling time is given by $T_s = \frac{1}{8000} = 1.25 \times 10^{-4}\ seconds = 125$ µsecond

so we have $\mathrm{s} = 16000 \times \frac{z-1}{z+1}$ thus *H(z)* is given by:

$$\mathrm{H}(z) = \frac{1}{\left(0.001 \times 16000 \times \frac{z-1}{z+1} + 1\right)}$$
$$\Rightarrow \mathrm{H}(z) = \frac{z+1}{(16 \times (z-1) + \mathrm{z} + 1)}$$
$$\Rightarrow \mathrm{H}(z) = \frac{z+1}{17z - 15}$$
$$\Rightarrow \mathrm{H}(z) = \frac{1 + z^{-1}}{17 - 15z^{-1}}$$
$$\Rightarrow \mathrm{H}(z) = \frac{\frac{1}{17} + \frac{1}{17}z^{-1}}{1 - \frac{15}{17}z^{-1}}$$

The last expression of *H(z)* is in the form of the IIR filter Z-domain transfer function $\mathrm{H}(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}$ where $b_0 = b_1 = \frac{1}{17}$ and $a_1 = -\frac{15}{17}$.

This corresponds to the time domain difference equation

$$y(n) = b_0 x(n) + b_1 x(n-1) - a_1 y(n-1)$$
$$= \frac{1}{17}x(n) + \frac{1}{17}x(n-1) + \frac{15}{17}y(n-1)$$

$$= \frac{1}{17}(x(n) + x(n-1) + 15y(n-1))$$

With T_s = 1.25 x 10^{-4} second and ω_p being the frequency in the discrete time, the analogue domain frequency is then given by:

$$\omega = \frac{2}{T_s}\tan(\frac{\omega_p T_s}{2})$$
$$\Rightarrow \frac{T_s}{2}\omega = \tan(\frac{\omega_p T_s}{2})$$
$$\Rightarrow \frac{2}{T_s}\tan^{-1}(\frac{T_s}{2}\omega) = \omega_p$$

Now, for an analogue domain cut off frequency of 1000 rad/second, the corresponding discrete domain cut off frequency would then be

$$\omega_p = \frac{2}{1.25 \text{ x } 10^{-4}}\tan^{-1}(\frac{1.25 \text{ x } 10^{-4}}{2}1000)$$
$$= 16000 \tan^{-1}(0.0625)$$
$$= 998.7\ rad/second$$
$$= 159\ Hertz$$

So the analogue domain and discrete domain cut off frequencies are approximately the same even after undertaking the Tustin transform. This is mostly due to the fact that the sampling frequency (8 KHz) is high compared to the value of the corner frequency. The product $\omega_p T_s$ is thus small, making the continuous frequency very close to being equal to the discrete frequency.

1.4. Direct form 1 implementation

1.4.1. C code implementation (direct form 1)

The first implementation of the IIR filter will be in the direct form 1. The time domain signal flow diagram is shown in figure 1.4.1.a and its corresponding z-domain diagram is shown in the next figure 1.4.1.b.

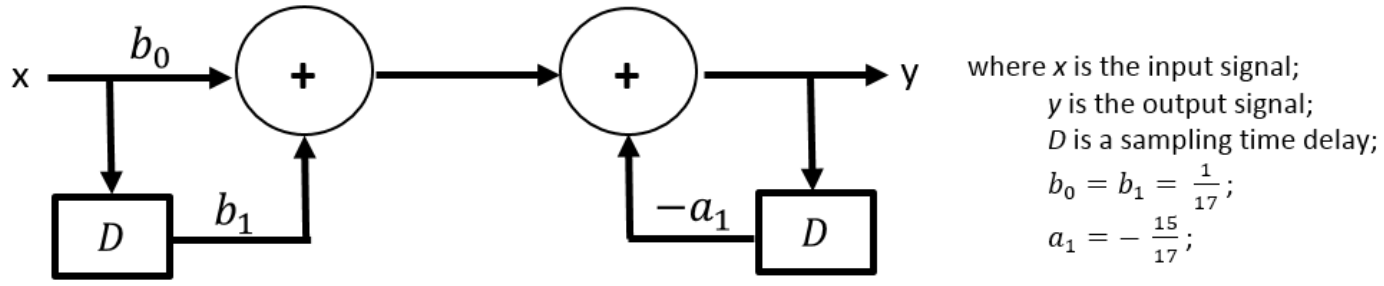

Figure 1.4.1.a: Time domain signal flow diagram

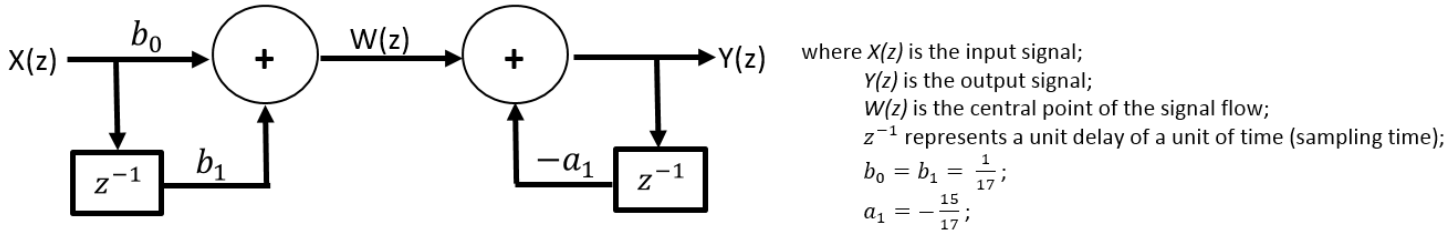

Figure 1.4.1.b: Corresponding Z-domain signal flow diagram

To implement this on the DSK board, the file *intio.c* is modified. It first includes the coefficients *a* and *b* with the pre-processor statement `#include "coef.txt"` from the file *coef.txt*, which has the following content:

```
const float b[] = {(1/(double)17), (1/(double)17)};
const float a[] = {1.0000, - 15/(double)17};
```

The *const* declarations ensure the coefficients included will not be changed by the program and will remain constants. The *(double)* casting in the initialisation of the arrays is needed for the numerator and/or the denominator in order to perform a double type resulting division. The first element (*1.0000*) of the array *a* is never used but is inserted to make the code clearer.

The program then defines the order of the filter *N* as the number of elements of the array of coefficients *a* with `#define N ((sizeof(a)/sizeof(a[0]))-1)` and declares the global arrays of floats *x* and *y* with `float x[N+1] = {0}, y[N+1] = {0};` . These arrays will serve as delay buffers for the input *x* and the output *y* in the interrupt service routine (ISR).

The last modification concerns the ISR, as described below.

```
/************************** INTERRUPT SERVICE ROUTINE ******************************/

void ISR_AIC(void)
{
    x[0] = mono_read_16Bit();
    filter();
    mono_write_16Bit((int)y[0]);
}
```

where the function *filter* is defined as:

```
void filter(void)

{
    int n;
    y[0] = b[0] * x[0];
    for(n = N; n > 0; n--)
    {
        y[0] += x[n]*b[n] - y[n]*a[n];
        y[n] = y[n-1];
        x[n] = x[n-1];
    }
}
```

This filter function requires 100 and 32 clock cycles respectively without optimisation and with the –o2 optimisation level option.

The DSK memory is configured through the DSP_bios tool such that a heap memory is created. This section of memory is not natively managed by the DSK and can be managed by the user using *calloc()*. This function allocates a section of memory in the heap and returns a pointer to the first element of this block. This will be used to allocate memory dynamically in order to avoid changing the size of the arrays separately. It is achieved with the following global variable declarations:

```
int N = (sizeof(a)/sizeof(a[0]))-1; //order of filter
float *x, *y; //pointers to floats
```

and the following code executed in the *main()*, before the hardware interrupt initialization:

```
x = (float *) calloc(N, sizeof(float));
y = (float *) calloc(N, sizeof(float));
```

This implementation allows to use any order of IIR filter without changing the code. This ISR now requires 120 and 78 clock cycles respectively without optimisation and with the –o2 optimisation level option. This code is slower but more flexible and will hence be used in what follows.

1.4.2. Testing and measurements (RC circuit)

The input line (left and right) of the DSK board is connected to the computer output mini-jack port, and its output port is connected to the channel 1 of the oscilloscope. The channel 2 of the oscilloscope is connected to the input port of the DSK board.

1.4.2.1. Frequency response of the RC circuit

The signal generator program is used to produce a variable frequency sine wave of 3.42 volts peak-to-peak. The measurements obtained with the oscilloscope are shown in the table of appendix 1 and are plotted in the two following figures.

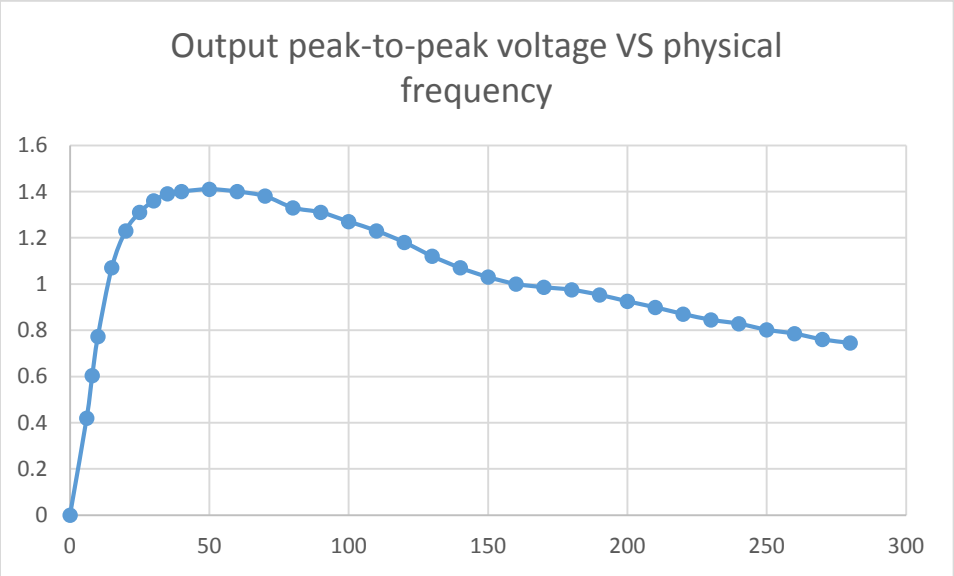

Figure 1.4.2.1.a: Output peak-to-peak voltage (V) VS physical frequency (Hz)

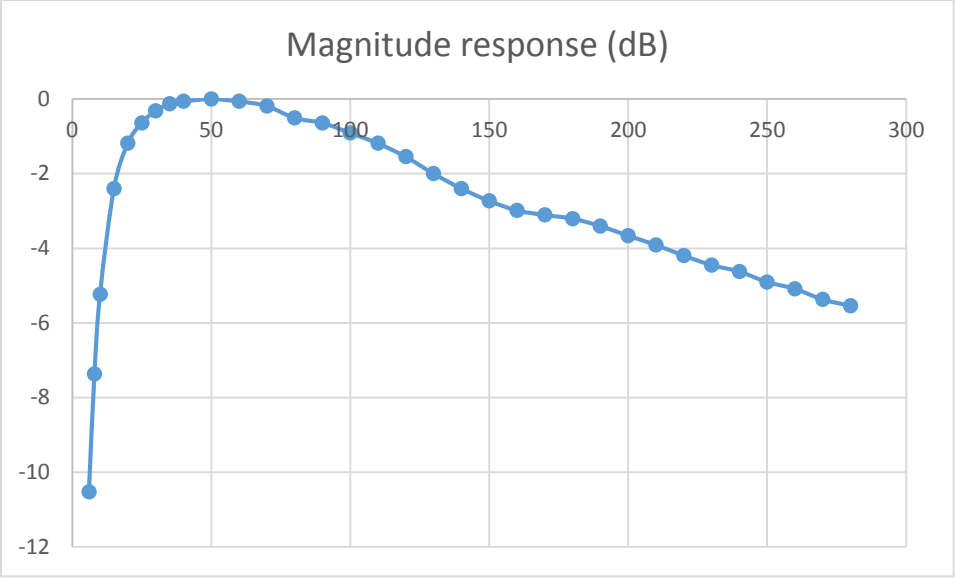

Figure 1.4.2.1.b: Magnitude gain (dB) VS physical frequency (Hz)

From the plot of figure 1.4.2.1.b, a gain of -3 dB (linear gain of 0.707) is attained at about 160 Hz, which was the cut off frequency expected.

1.4.2.2. Measurement of the time constant of the RC circuit

The signal generator program is used to produce a square wave at 25 Hz (not lower to avoid too much distortion from the DSK built-in high pass filter at the input port). The relevant observation are shown in the figure below.

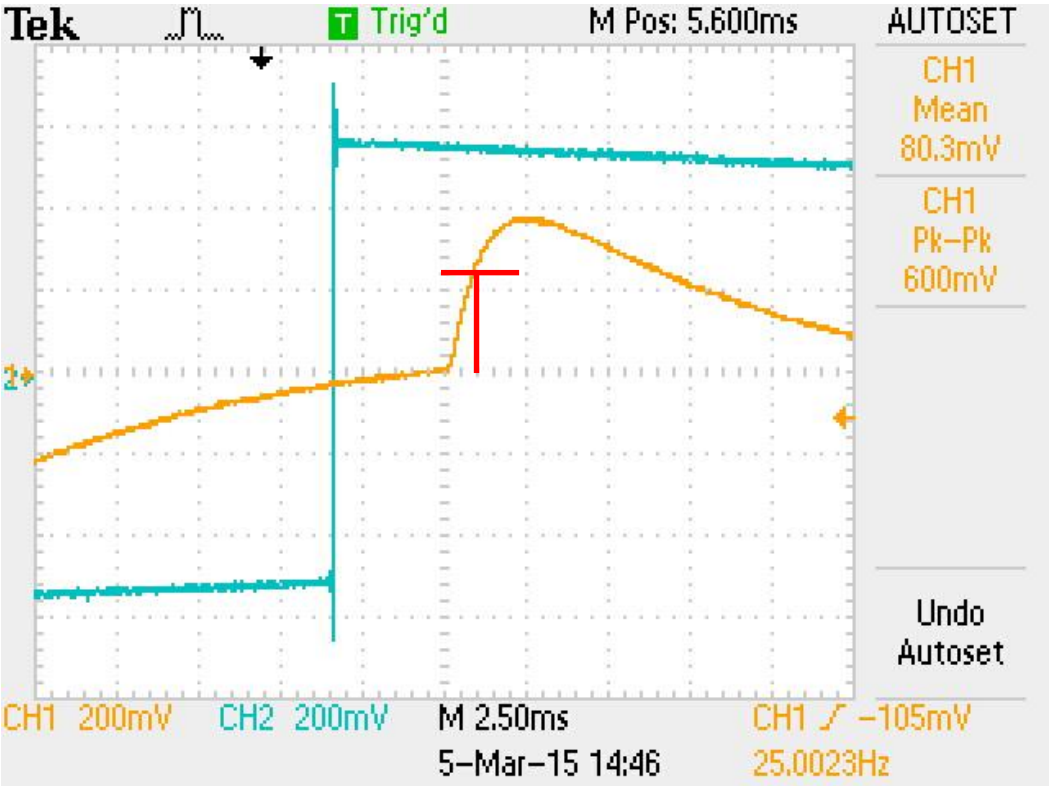

Figure 1.4.2.2: Oscilloscope oberservation for time constant measurement of the digital filter

The time constant is the time required to charge the capacitor to 63.2% of the difference between the initial voltage value and the final one. In this case, the difference is *d* = 1.9 div x 200mV – 0.0 div x 200mV = 380mV and 63.2% of it is then 0.632 x 380mV = 240.16mV which corresponds to 1.2 divisions. The time needed to reach 1.2 voltage division is 0.3 time division which corresponds to 0.4 div x 2.5mS = 1 mS. This time constant corresponds to the previously calculated time constant (product R by C).

The implementation of this single pole filter (RC circuit) is thus giving the expected theoretical results.

1.5. Direct form 2 implementation

In terms of control, it can be shown that the direct form 1 signal flow diagram which was implemented in 1.4. is equivalent to a more efficient signal flow diagram called direct form 2. This equivalence is due to the linearity and shift invariance of the system.

The original signal flow diagram which has been implemented is shown below.

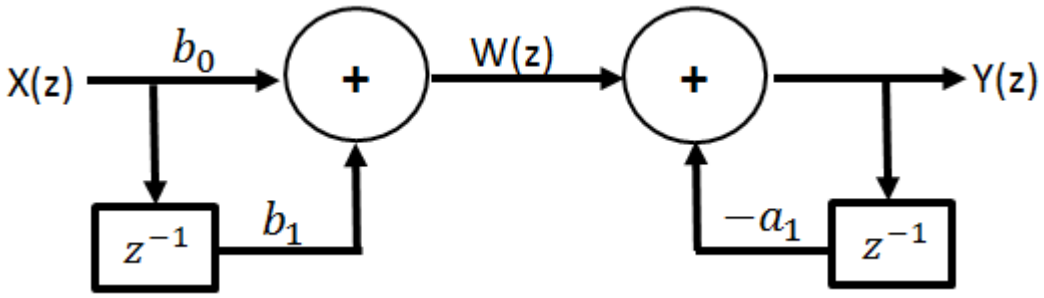

Figure 1.5.1: Signal flow diagram of direct form 1 for the 1 pole filter

The order of the two parts of the transfer function can be swapped round without changing the transfer from input to output. This results in the following signal flow diagram:

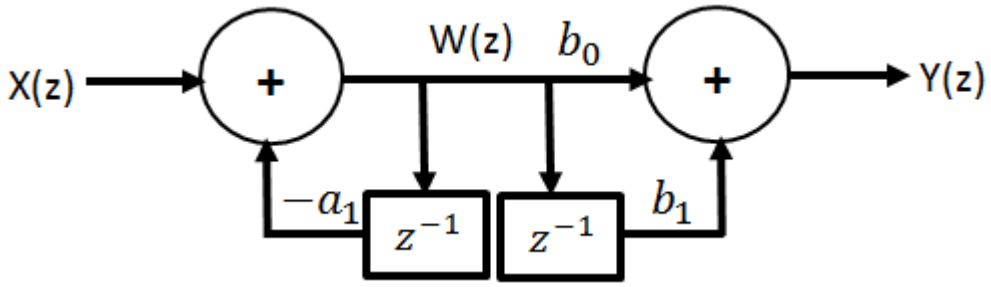

Figure 1.5.2: Signal flow diagram of for the 1 pole filter

This diagram can then be simplified by using only one delay element as shown in figure 2.5.3.

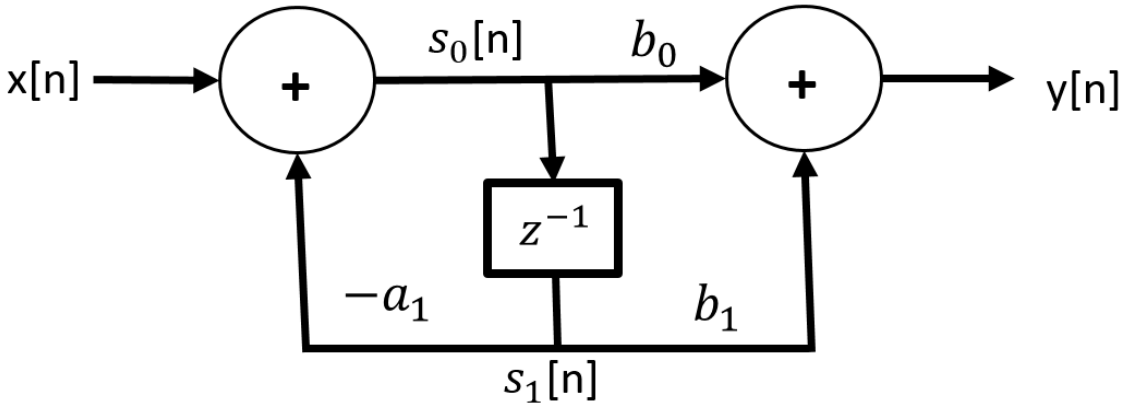

Figure 1.5.3: Signal flow diagram of direct form 2 for the 1 pole filter

From this diagram, the following relations are deduced.
s_0[n] = x[n] $-a_1 s_1$[n]
y[n] = $b_0 s_0$[n] + $b_1 s_1$[n]
where x[n] and y[n] are respectively the input and output signals, s_0[n] is the signal at the centre and s_1[n] is the delayed version signal of s_0[n].

In a more general case, for an IIR filter of order *N* these equations become
s_0[n] = x[n] $-a_1 s_1$[n] $-a_2 s_2$[n]− … − $a_{N+1} s_{N+1}$[n]
y[n] = $b_0 s_0$[n] $+b_1 s_1$[n] $+b_2 s_2$[n]+ … + $b_{N+1} s_{N+1}$[n]
and the state variables s_i have to be updated according to the following equations:
s_{N+1}[n+1] = s_N[n]
s_N[n+1] = s_{N-1}[n]
…
s_2[n+1] = s_1[n]s
s_1[n+1] = $s_0[n]$

The modifications needed to implement this in the C code are described in the following. Only one array of floats is now needed and declared with a global float pointer by `float *s;` The array is then allocated using `s = (float *) calloc(N, sizeof(float));` in the *main().*

The *ISR_AIC()* interrupt routine is modified as follows:

```
/************************* INTERRUPT SERVICE ROUTINE ******************************/

void ISR_AIC(void)
{
    float y;
    s[0] = mono_read_16Bit();
    y = filter();
    mono_write_16Bit((int)y);
}
```

where the function *filter* is defined as:

```
float filter(void)

{
    int n;
    float y = 0;
    for(n = N; n > 0; n--)
    {
        y += b[n] * s[n]; //sums the output
        s[0] -= a[n]*s[n];
        s[n] = s[n-1]; //does the delay
    }
    y += b[0] * s[0];
    return y;
}
```

This implements the direct form II and gives the same experimental results as for the direct form I implementation. It consumes 122 and 66 clock cycles respectively without optimisation and with the –o2 optimisation option level.

1.6. Direct form 2 transposed implementation

Applying the following rules to the direct form 2 signal flow diagram:

A network is unchanged in behaviour if:
-the direction of each branch is reversed
-the branch divisions are interchanged with branch summations
-the input and output are swapped round

we end up with the direct form II transposed form, as shown in the signal flow diagram below.

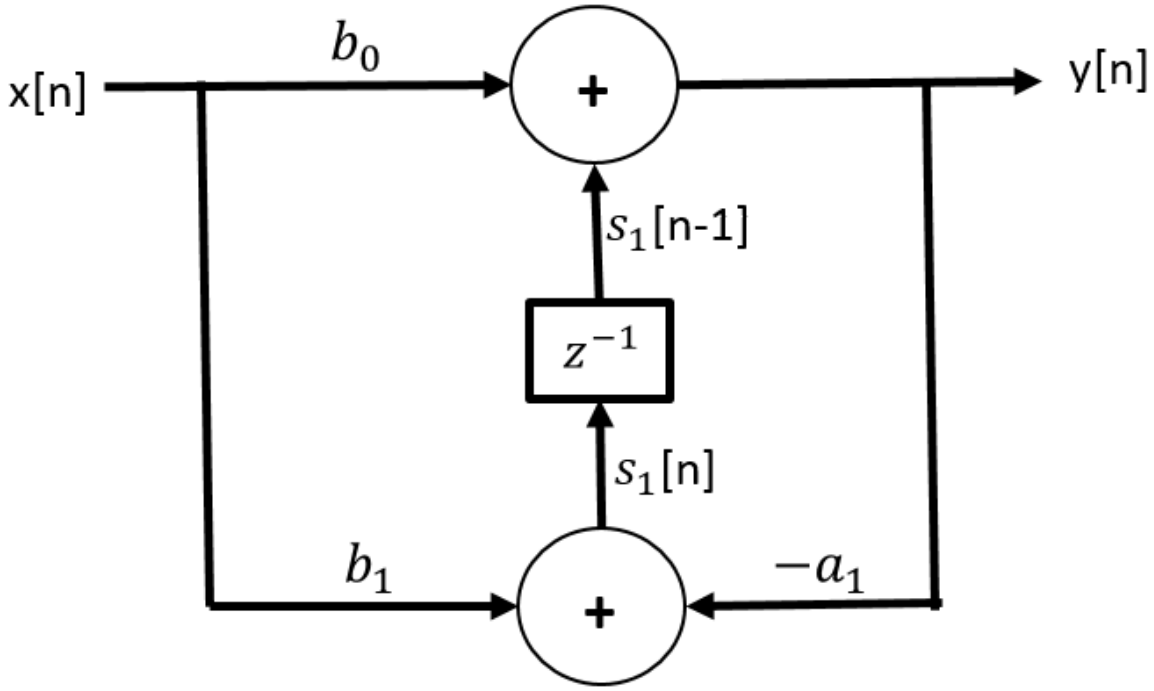

Figure 1.6.1: Signal flow diagram of direct form II transposed for the 1 pole filter

The implementation of this form will directly be made to support any order of filter *N*.
For a IIR filter of order *N* = 3, the signal flow diagram for this direct form II transposed then becomes

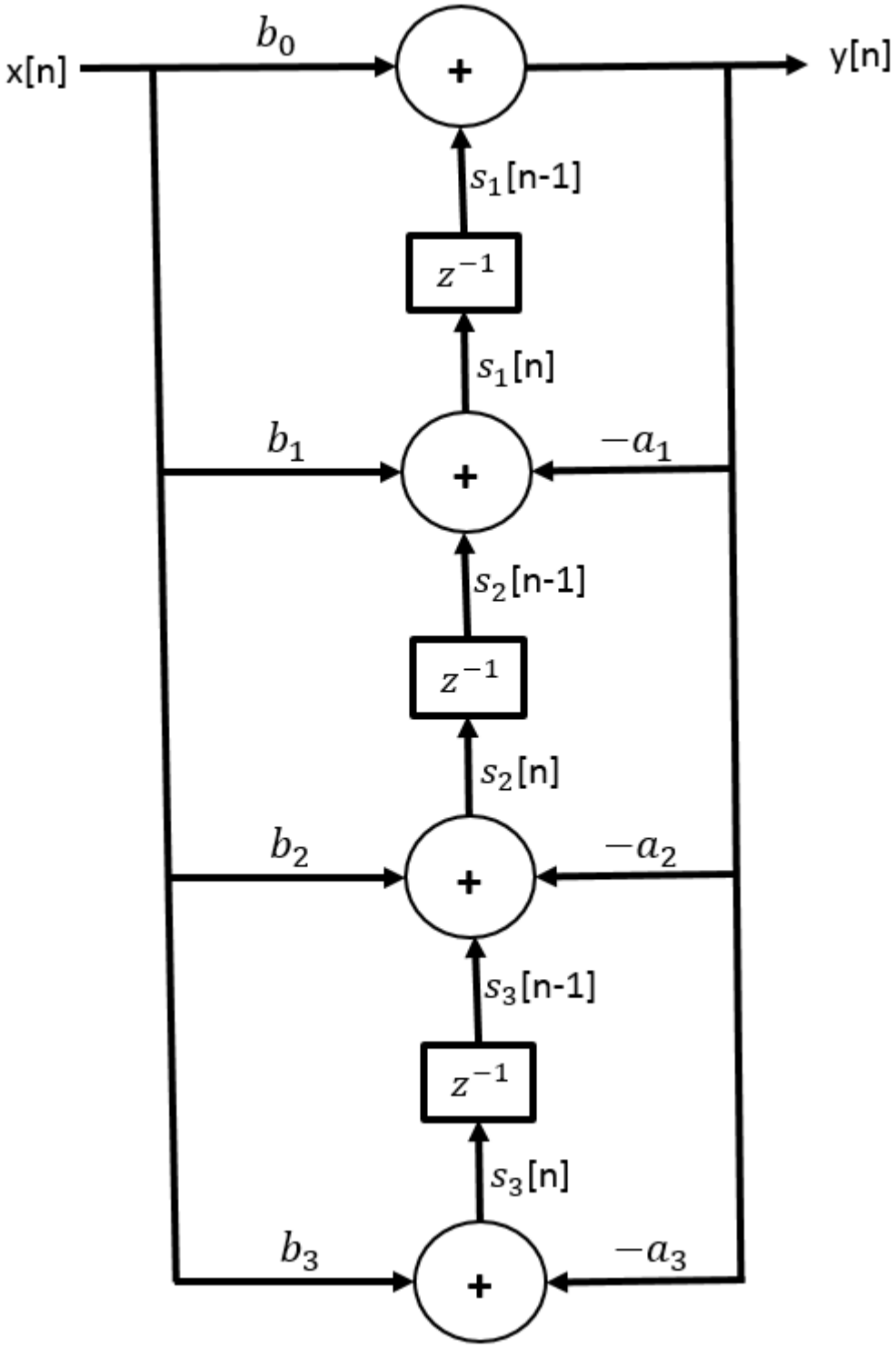

Figure 1.6.2: Signal flow diagram of direct form II transposed for an IIR filter of order 3

The easiest way to implement this form of IIR filter is by using a for loop and by keeping a record of the previous value of the output.

The code used for the direct form II is modified as described below.

A global variable `float y;` is used to store the previous value of the output (done in the interrupt routine). As before, an array *s* is declared in the *main()* by `s = (float *) calloc(N, sizeof(float));`

In the *main()*, the array *s* is allocated to contain N+1 element, where its last element is set to 0 (never changed):

```
    s = (float *) calloc(N+1, sizeof(float));
    s[N+1] = 0;
```

This last 'N+1'th element is set to zero in order to perform all the calculation into the for loop of the *filter()* function described in the following.

The ISR_AIC() is modified slightly

```
/************************** INTERRUPT SERVICE ROUTINE ******************************/

void ISR_AIC(void)
{
    float x = mono_read_16Bit();
    filter(x);
    mono_write_16Bit((int)y);
}
```

and the *filter()* function does the calculation of the equations derived from the signal flow diagram of figure 2.6.2.

```
void filter(int x)

{
    int i;
    for(i = 1; i <= N; i++)
    {
        s[i] = b[i]*x - a[i]*y + s[i+1];
    }
    y = s[1] + b[0]*x;
}
```

Here *y* represents the previous value of the output during the calculations and is then assigned the new value of the output (to write). The array *s* represents the summation results of the products as from the signal flow diagram shown before.

This implements the direct form II transposed and gives the same experimental results as before. It uses 100 and 77 clock cycles respectively without optimisation and with the –o2 optimisation option level.

The following table is a sum up of performance, measured in clock cycles, of each implementation realised for this single pole filter.

Form implemented	No optimisation	Optimisation level option -o2
Direct form I	100	32
Direct form I with calloc	120	78
Direct form II with calloc	122	66
Direct form II transposed with calloc	100	77

Figure 1.6.3: Performance summary, in clock cycles (1 pole filter)

2. Designing a discrete time bandpass filter

2.1. Objective

The elliptic bandpass filter required to design in a discrete time version must satisfy the following specifications:
Order: 4^{th}
Passband: 280 Hz to 470 Hz
Passband ripple: 0.5 dB
Stopband attenuation: 25 dB
The coefficients *a* and *b* needed to implement this filter will be generated with a MatLab script (using the function ellip).

2.2. MatLab IIR design script

To design the required filter, the following script is used:

```
%% ************* MATLAB IIR design script ************* %%

%% Authors: Alexandra Rouhana and Quentin McGaw %%
%% Date: February-March 2015 %%


% ~~~~~~~~~~~~~~~~ P A R A M E T E R S ~~~~~~~~~~~~~~~~
% ~~~ Filter parameters ~~~
order = 4;
% ~~~ Frequency parameters ~~~
Fsampling = 8000; %sampling frequency (C6713 sampling)
Fpassband = [280 470]; %Pass band frequency range
% ~~~ Gain parameters ~~~
PB_ripple=0.5; %pass band ripple (dB)
SB_attenuation=25; %stop band attenuation (dB)

% ~~~~~~~~~~~~~~ C A L C U L A T I O N S ~~~~~~~~~~~~~~
Nyquist = Fsampling/2; %Nyquist frequency
Fnormalized = Fpassband/Nyquist;
%normalized frequencies to the Nyquist frequency
[b,a] = ellip(order/2,PB_ripple,SB_attenuation,Fnormalized);

% ~~~~~~~~~~~~~~~~~~ R E S U L T S ~~~~~~~~~~~~~~~~~~
freqz(b,a);
str_a='const float a[] = {';
str_b='const float b[] = {';
for i = 1:order
    str_a = [str_a num2str(a(i)) ','];
    str_b = [str_b num2str(b(i)) ','];
end
str_a = [str_a num2str(a(order+1)) '};'];
str_b = [str_b num2str(b(order+1)) '};'];
str_all = [str_a '\n' str_b '\n' '\n'];
dlmwrite('coef.txt',str_all,'');
```

This script generates the coefficients *a* and *b* in the text file *coef.txt* where `float a[] = {1,-3.6305,5.0962,-3.2741,0.81432};float b[] = {0.057901,-0.19414,0.2728,-0.19414,0.057901};` as well as the MatLab frequency plot as shown in the figure 3.2.a below.

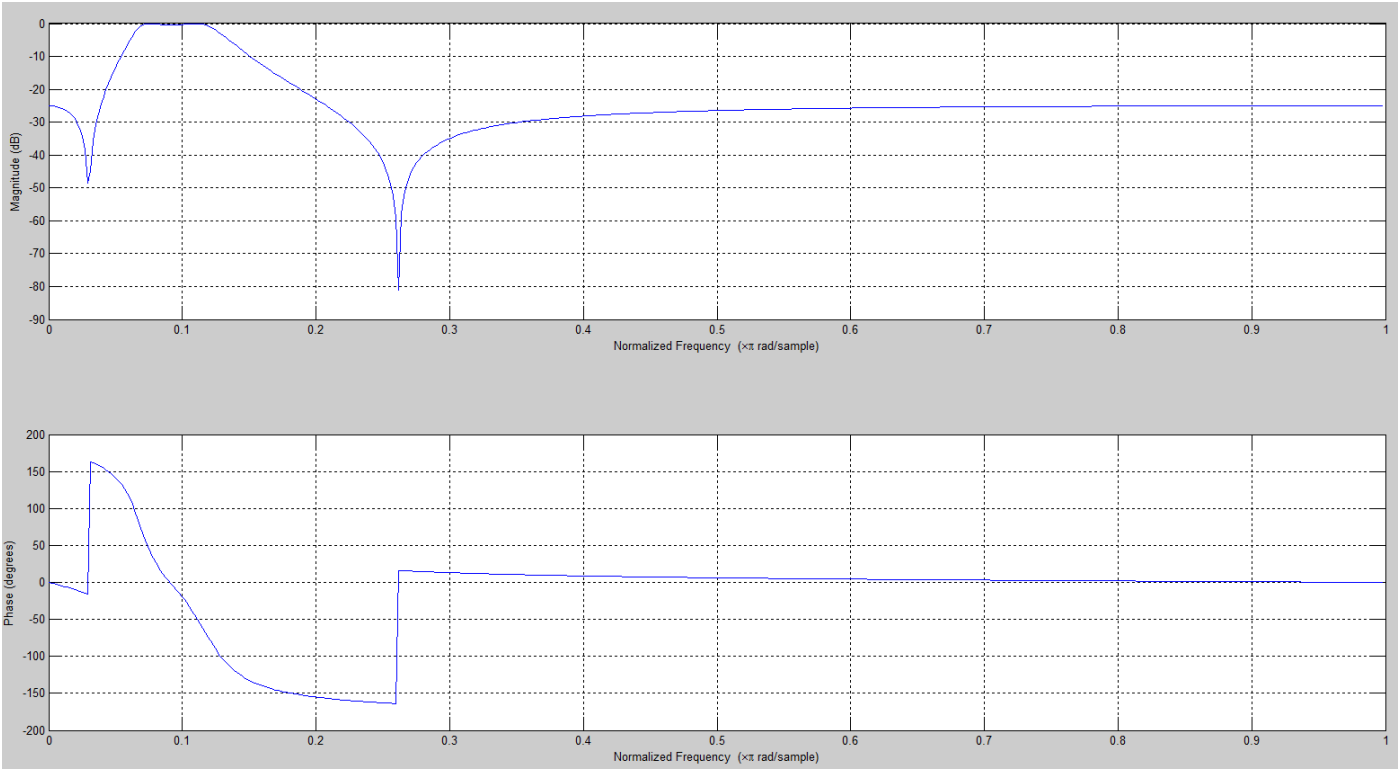

Figure 2.2.a: MatLab frequency plot response for the elliptic filter

The magnitude response corresponds to the required specifications (passband, ripple, stopband, attenuation).

2.3. Direct form II implementation

To implement this 4th order IIR filter, the code written in part 2.5 will be used as it works with any order of filter. As a reminder, the interrupt routine for the direct form II is coded as follow:

```
/************************** INTERRUPT SERVICE ROUTINE ******************************/

void ISR_AIC(void)
{
    float y;
    s[0] = mono_read_16Bit();
    y = filter();
    mono_write_16Bit(y);
}

float filter(void)
{
    int n;
    float y = 0;
    for(n = N; n > 0; n--)
    {
        y += b[n] * s[n]; //sums the output
        s[0] -= a[n]*s[n];
        s[n] = s[n-1]; //does the delay
    }
    y += b[0] * s[0];
    return y;
}
```

The function *filter()* uses 458 and 214 clock cycles respectively without optimisation and with the –o2 optimisation level option. With the single pole filter, these values were 122 and 66 clock cycles.

It can thus be deduced that for a filter of order *n*, each sample requires $\frac{458-122}{4-1}n = \frac{336}{3}n = 112n$ clock cycles without optimisation. The instruction cycles per sample needed in the form *A + Bn* is thus 10 + 112*n* with no optimisation option.

Similarly, with the –o2 optimisation level option, each sample requires $\frac{214-66}{4-1}n = \frac{148}{3}n = 50n$ instruction cycles. In this case, the instruction cycles per sample needed is 16 + 55*n.*

2.4. Frequency response measurement using APX515

The APX515 spectrum analyser is connected to the DSK board running the code of the direct form II with the coefficients *a* and *b* corresponding to the 4th order IIR filter.

The following plot is obtained concerning the magnitude response of the designed digital filter.

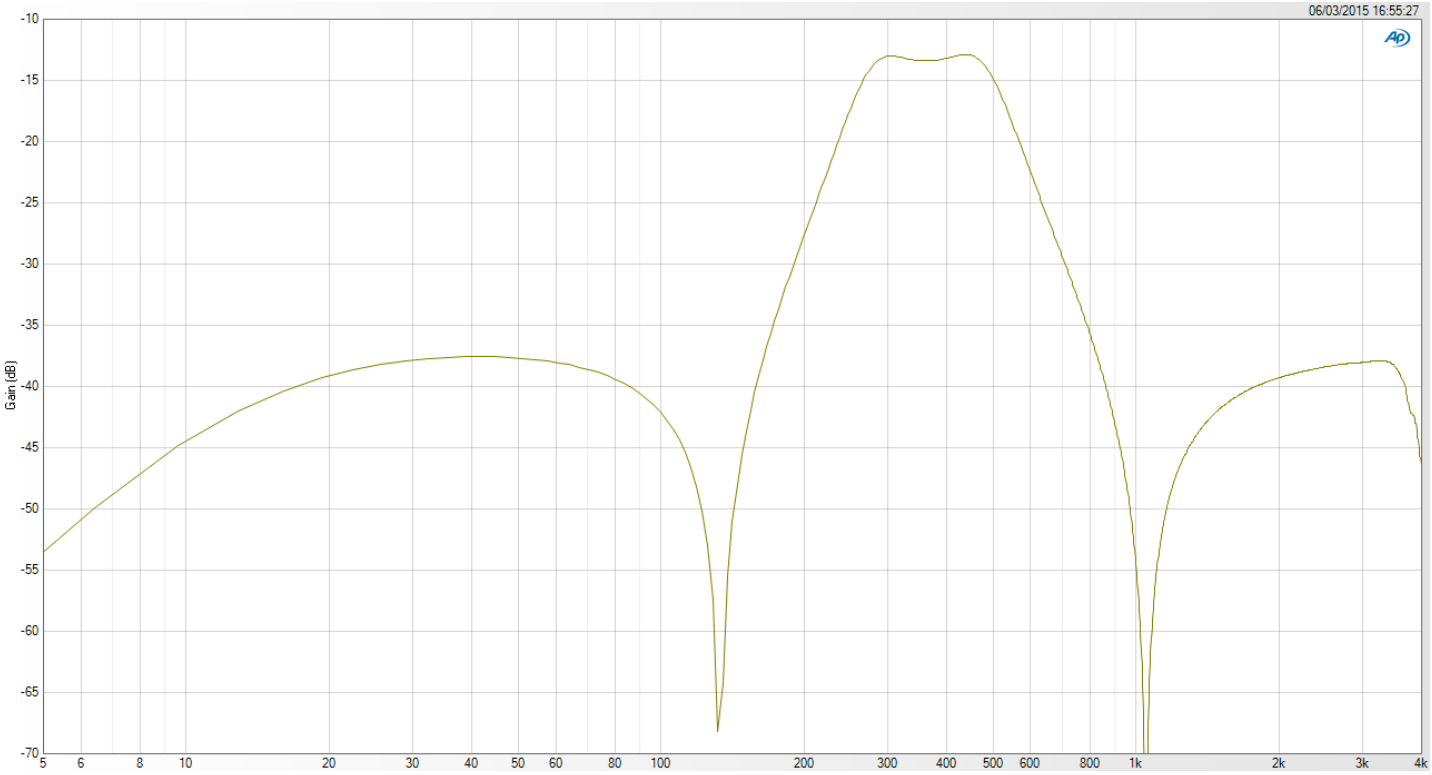

Figure 2.4.a: Gain response of the digital filter, using APX500 software

The passband corresponds to the specifications (280 Hz to 470 Hz), as does the passband ripple (0.5 dB) and the stopband attenuation. Please again **note** all of this gain plot is shifted by -12.5 dB because of the attenuators at the input ports which divide the input signal by 4.

2.5. Direct form II transposed implementation

To implement this 4th order IIR filter, the code written in part 2.6 will be used as it works with any order of filter. Please refer to section 2.6 for the C code description. As a quick reminder, the interrupt routine is as follows:

```
/************************** INTERRUPT SERVICE ROUTINE ******************************/

void ISR_AIC(void)
{
    float x = mono_read_16Bit();
    filter(x);
    mono_write_16Bit((int)y);
}

void filter(int x)
{
    int i;
    for(i = 1; i <= N; i++)
    {
        s[i] = b[i]*x - a[i]*y + s[i+1];
    }
    y = s[1] + b[0]*x;
}
```

The function *filter()* uses 229 and 90 clock cycles respectively without optimisation and with the –o2 optimisation level option. With the single pole filter, these values were 100 and 77 clock cycles.

It can thus be deduced that for a filter of order *n*, each sample requires $\frac{229-100}{4-1}n = \frac{129}{3}n = 43n$ clock cycles without optimisation. The instruction cycles per sample needed in the form *A + Bn* is thus 67 + 43*n* with no optimisation option.

Similarly, with the –o2 optimisation level option, each sample requires $\frac{90-77}{4-1}n = \frac{13}{3}n = 5n$ instruction cycles. In this case, the instruction cycles per sample needed is 72 + 5*n.*

The following table is a sum up of performance, measured in clock cycles, of each implementation realised for this 4th order IIR filter.

Form implemented	No optimisation	Optimisation level option -o2
Direct form I	366	117
Direct form I with calloc	392	252
Direct form II with calloc	458	214
Direct form II transposed with calloc	229	90

Figure 2.5.a: Performance summary, in clock cycles (Passband filter)

Conclusion

The following table is a summary of the instruction cycles required for an IIR filter of order n in the form *A* + *Bn*.

Form implemented	A	B
Direct form II	10	112
Direct form II transposed	67	43
Direct form II with -o2	16	55
Direct form II transposed with -o2	72	5

This clearly shows the best solution is the direct form II transposed with –o2 for any order.

This lab have demonstrated how to implement a digital IIR filter in various forms and with different optimisation option levels. It is essential to understand the importance of the signal flow diagram manipulations which allow to make an algorithm way faster.

Appendix 1: Measurements results for single-pole IIR filter (RC circuit)

Physical Frequency (Hz)	Output p-p voltage (V)	Linear Gain	Gain (dB)
0	0	0.0001	-80
6	0.42	0.29787234	-10.51939645
8	0.604	0.428368794	-7.363643481
10	0.772	0.54751773	-5.232036246
15	1.07	0.758865248	-2.396706699
20	1.23	0.872340426	-1.186280024
25	1.31	0.929078014	-0.63895634
30	1.36	0.964539007	-0.313604086
35	1.39	0.985815603	-0.124086248
40	1.4	0.992907801	-0.06182154
50	1.41	1	0
60	1.4	0.992907801	-0.06182154
70	1.38	0.978723404	-0.186800525
80	1.33	0.943262411	-0.507349434
90	1.31	0.929078014	-0.63895634
100	1.27	0.90070922	-0.908307834
110	1.23	0.872340426	-1.186280024
120	1.18	0.836879433	-1.546742107
130	1.12	0.794326241	-2.0000218
140	1.07	0.758865248	-2.396706699
150	1.03	0.730496454	-2.727637759
160	1	0.709219858	-2.984382253
170	0.986	0.69929078	-3.106843954
180	0.975	0.691489362	-3.204289939
190	0.953	0.675886525	-3.40252424
200	0.925	0.656028369	-3.661547598
210	0.899	0.637588652	-3.909188418
220	0.87	0.617021277	-4.193997201
230	0.845	0.59929078	-4.447248074
240	0.828	0.587234043	-4.623775517
250	0.802	0.568794326	-4.900894887
260	0.785	0.556737589	-5.086989118
270	0.76	0.539007092	-5.368110407
280	0.745	0.528368794	-5.541256798