Yehar's digital sound processing tutorial for the braindead!

version 1998.10.28

www.student.oulu.fi/~oniemita
/DSP/DSPSTUFF.TXT

Chapters:

-- Foreword ---

This is written for the audio digital signal processing enthusiasts (as the title suggests ;) and others who need practical information on the subject. If you don't have this as a "linear reading experience" and encounter difficulties, check if there's something to help you out in the previous chapters. Comments and error reports are welcome. Especially i'd like to hear if you have had trouble understanding something. [NEWS! JUNE/2000 - I'm working on a new book and am no longer updating this text]

In filter frequency response plots, linear frequency and magnitude scales are used. Page changes are designed for 60+ lines/page printers.

Chapter "Shuffling IIR equations" is written by my big brother Kalle. And, thanks to Timo Tossavainen for sharing his DSP knowledge!

by Yehar, Olli Niemitalo, ollinie@freenet.hut.fi

```
----------------------------------------------- About sampled sound ---
```

Note that "sample" can mean (1) a sampled sound or (2) a samplepoint!

Sampled sound data is a pile of samples, amplitude values taken from
the actual sound wave. Sampling rate is the frequency of the "shots".
For example, if the frequency is 44100, 44100 samples have been taken
in one second.

Here's an example of sampling:

```
                                   _0---0_
                                 _/       --0__
0_                __0---0__    _0              -0---0___
  \_          _-0-         -0--                         0-__
+---0==-+--=0---+---+---+---+---+---+---+---+---+---+---+---0==-+---+
       -0--                                                    -0_
                                                                  --0
                                    <--->
                                1/Samplerate
```

The original sound is the curve, and "0"s are the sampled points. The
horizontal straight line is the zero level.

A sampled sound can only represent frequencies up to half the
samplerate. This is called the Nyquist frequency. An easy proof:
You need to have stored at least two samplepoints per wave cycle, the
top and the bottom of the wave to be able to reconstruct it later on:

```
0\     /0\     /0\     /0\     /0\     /0\     /0\     /0\     /0\
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+-|-+
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
   \0/     \0/     \0/     \0/     \0/     \0/     \0/     \0/     \0
```

If you try to include above Nyquist frequencies in your sampled sound,
all you get is extra distortion as they appear as lower frequencies.

```
------------------------------------ Adding two sine waves together ---
```

A Sound consists of frequency components. They all look exactly like
sine waves, but they have different frequencies, phases and amplitudes.
Let's look at a single frequency:

```
        | __        __        __        __      |
        |/  \      /  \      /  \      /  \     |
        /----\----/----\----/----\----/----\----/
        |     \__/      \__/      \__/      \__/|
        |                                       |
```

Now, we take the same frequency from another sound and notice that it
has the same amplitude, but the opposite (rotated 180 degrees) phase.

```
        |      __        __        __        __ |
        |     /  \      /  \      /  \      /  \|
        \----/----\----/----\----/----\----/----\
        |\__/      \__/      \__/      \__/     |
        |                                       |
```

Merging two signals is done simply by adding them together. If we do the same with these two sine waves, the result will be:

```
        |                                       |
        |                                       |
        +=======================================+
        |                                       |
        |                                       |
```

It gets silent. If we think of other cases, where the phase difference is less than 180 degrees, we get sine waves that all have different amplitudes and phases, but the same frequency.

Here's the way to calculate the phase and the amplitude of the resulting sinewave... Convert the amplitude and phase into one complex number, where angle is the phase, and absolute value the amplitude.

amplitude*e^(i*phase) = amplitude*cos(phase)+i*amplitude*sin(phase)

If you do this to both of the sinewaves, you can add them together as complex numbers.

Example:
(Wave A) amplitude 1, phase 0, (Wave B) amplitude 1, phase 90 degrees

```
           _---|---_         _---0---_         _---|---_ 0
          /    |    \       /    |    \       /    |    \
         |     |     |     |     |     |     |     |     |
        -|-----+-----0- + -|-----+-----|- = -|-----+-----|-
         |     |     |     |     |     |     |     |     |
          \_   |   _/       \_   |   _/       \_   |   _/
            ---|---           ---|---           ---|---
```

As you see, the phase of the new sine wave is 45 degrees and the amplitude sqrt(1^2+1^2) = sqrt(2) = about 1.4

It is very important that you understand this, because in many cases, it is more practical to present the amplitude and the phase of a frequency as a complex number.

When adding two sampled sounds together, you may actually wipe out some frequencies, those that had opposite phases and equal amplitudes. The average amplitude of the resulting sound is (for independent originals) sqrt(a^2+b^2) where a and b are the amplitudes of the original signals.

-- What's a filter? ---

The main use of a filter is to scale the amplitudes of the frequency components in a sound. For example, a "lowpass filter" mutes all frequency components above the "cutoff frequency", in other words, multiplies the amplitudes by 0. It lets through all the frequencies below the cutoff frequency unattenuated.

*** Magnitude ***

If you investigate the behaviour of a lowpass filter by driving various sinewaves of different frequencies through it, and measure the amplifications, you get the "magnitude frequency response". Here's a plot of the magnitude frequency response curve of a lowpass filter:

```
        1-+----------------------_                               |
          |                       \                              |
          |       Audible          |          Inaudible          |
          |                        |                             |
          |                        |                             |
          |                         \_                           |
        0-+---------------------------============================
          |                        |                             |
         0Hz                Cutoff frequency                   max
```

Frequency is on the "-" axis and amplification on the "|" axis. As you see, the amplification (= scaling) of the frequencies below the cutoff frequency is 1. So, their amplitudes are not affected in any way. But the amplitudes of frequencies above the cutoff frequency get multiplied by zero so they vanish.

Filters never add any new frequency components to the sound. They can only scale the amplitudes of already existing frequencies. For example, if you have a completely quiet sample, you can't get any sound out of it by filtering. Also, if you have a sine wave sample and filter it, the result will still be the same sine wave, only maybe with different amplitude and phase - no other frequencies can appear.

*** Phase ***

Professionals never get tired of reminding us how important it is not to forget the phase. The frequency components in a sound have their amplitudes and... phases. If we take a sine wave and a cosine wave, we see that they look alike, but they have a phase difference of pi/2, one fourth of a full cycle. Also, when you play them, they sound alike. But, try wearing a headset and play the sinewave on the left channel and the cosine wave on the right channel. Now you hear the difference!

Phase itself doesn't contain important information for us so it's not heard, but the phase difference, of a frequency, between the two ears can be used in estimating the position of the origin of the sound so it's heard.

Filters have a magnitude frequency response, but they also have a
phase frequency response. Here's an example curve that could be from
a lowpass filter:

```
      +pi-+-----------------------------------------------------+
          |                    ___                              |
          |      _________-----   \                             |
        0-+======------------------|---------====================
          |                         \___-----                   |
          |                                                     |
      -pi-+-----------------------------------------------------+
          |                        |                            |
         0Hz                Cutoff frequency                   max
```

If you filter a sound, the values from the phase frequency response
are added to the phases of the frequencies of the original sound.

Linear (straight line) phase is the same thing as a plain delay,
although it may look wild in the plot if it goes around several times.
If your, for example, lowpass filter doesn't have a linear phase
frequency response, you can't turn it into a highpass filter by
simply subtracting its output from the original with equal delay.

*** Complex math with filters ***

The response of a filter for a single frequency can be expressed as a
complex number, where the angle is the phase response of the filter and
the absolute value the magnitude response. When you apply the filter to
a sound, you actually do a complex multiplication of all the frequency
components in the sound by the corresponding filter response values.
(Read chapter "Adding two sinewaves together" if you find this hard to
understand.) Example: The response of a filter is (0,1) at 1000Hz. You
filter a sine wave, with the phase & amplitude information presented as
the complex number (0,1), of the same frequency with it:

```
           Sine wave:         Filter:           Result:
           _---0---_         _---0---_         _---|---_
          /    |    \       /    |    \       /    |    \
         |     |     |     |     |     |     |     |     |
        -|-----+-----|- * -|-----+-----|- = -0-----+-----|-
         |     |     |     |     |     |     |     |     |
          \_   |   _/       \_   |   _/       \_   |   _/
            ---|---           ---|---           ---|---
```

The phase of the sine wave got rotated 90 degrees. No change in the
amplitude.

*** Combining filters ***

```
Serial (A*B):   In --> FILTER A --> FILTER B --> Out
```

The combined response of these two filters put in serial is the response of A multiplied by the response of B (Complex numbers as always!). If you only need to know the magnitude response, you could as well multiply the absolute values.

```
Parallel (A+B):    --> FILTER A -->
                In                  Out
                   --> FILTER B -->
```

In the figure, both filters get their inputs from the same source. Their outputs are then added back together, forming the final output. Now you need to use addition in solving the combined response.

--- Filter types: FIR and IIR ---

FIR = finite impulse response
IIR = infinite impulse response

FIR filter is more straightforward, and easier to understand. Finite impulse response means that when the filter input has remained zero for a certain time, the filter output also becomes zero. An Infinite impulse response filter never fully "settles down" after turning off the input, but it does get quieter and quieter though.

*** FIR ***

A basic FIR filter could be:

```
ouput(t) = a0*input(t) + a1*input(t-1) + a2*input(t-2),
```

where "input" means the sample values fed to the filter.

In this case, people would speak of a "3 tap" filter.

It's up to the coefficients (a0, a1, a2) what this filter will do to the sound. Choosing the coefficient values is the hardest part, and we'll get to that later. To design your own filters, you need to understand some of the math behind and know the right methods.

In the above filter example, only past input values are used. In realtime filters, this is a requirement, because you don't know the future inputs. In sample editors and such, you don't have this limitation, because you have the whole input data ready when you begin.

If your filter is:

```
output(t) = a0*input(t+1) + a1*input(t) + a2*input(t-1),
```

and you need a realtime version of it, just convert it to:

```
output(t) = a0*input(t) + a1*input(t-1) + a2*input(t-2).
```

The only difference is the one sample delay in the realtime filter.

*** IIR ***

Unlike FIR filters, IIR filters also use their previous output values in creating their present output. Here's a simple example:

```
output(t) =   a0*input(t) + a1*input(t-1) + a2*input(t-2)
            + b1*output(t-1) + b2*output(t-2) + b3*output(t-3)
```

This could be called "3 input, 3 output tap" filter.

IIR filters can never use future output values, because such don't yet exist!

There can be several ways of implementing the same IIR filter. Some may be faster than the usual input-output-and-coefficients way. Anyhow, every IIR filter can be written in this form, and it must be used in filter design and examining calculations.

An impulse response (= What the filter will do to a one samplepoint impulse) of an IIR filter often looks more or less like this in the sampledata:

```
            |
            |                time ->
            |
            |
            |      _-_     ___
        ====-\----/---\---/---\===/============================
              \__/     ---
```

Some badly designed IIR filters are unstable. This results in ouput getting louder and louder instead of quieter and quieter. A simple example of this is: output(t) = input(t) + 2*output(t-1). As soon as it gets input data, it gets crazy.

*** FFT ***

The above described filter types process the data sample by sample. FFT, Fast Fourier Transformation, doesn't. It does the work chunk by chunk. A chunk, which must be of length 2^n, is first converted into spectral information - complex numbers representing the phases and amplitudes of the frequency components. In this form, it is very easy to manipulate the spectrum. Then IFFT (Inverse FFT) is used to convert the information back to a chunk of sampledata.

If you just take a chunk of sampledata, it has sharp edges, which is bad for the FFT. Windowing functions are used to smoothen these edges. "Raised cosine", cos(x pi/2)^2, is a well-known windowing function. Here you see what happens when you apply a windowing function to a chunk of sampledata:

```
    Sampledata chunk:  Windowing function:      Result:
      (square wave)      (raised cosine)

 1 -|--   ---   --- |   |     _---_     |   |    _--        |
    |               |   |    /     \    |   |               |
    |               |   |  _/       \_  |   |           \_  |
 0 -|               | * |--           --| = |--_          --|
    |               |   |               |   |   \           |
    |               |   |               |   |    \    _/    |
-1 -|  ---   ---   -|   |               |   |        -      |
```

Overlapping windows (= chunks) are used, so the whole sampledata goes actually twice through the FFT. Here you see how the windows overlap:

```
______ _____________ _____________ ______________ ______________
_____________ _____________ _____________ _______________ ______
```

This is possible because of the symmetrical shape of the windowing function. If you add together overlapping raised cosine windowing functions, you get a flat result.

------------------------------------ Interpolation of sampled sound ---

Sometimes (resampling, precisely defined delay) you need to get samplevalues from between the known samplepoints. That's when you need interpolation. If you don't interpolate, and just throw away the fractional part of your sampleoffset, you get a lot of high frequency distortion:

```
0---                0---0---                0---
    |               |       |               |
    0---        0---        0---        0---
        |       |               |       |
        0---0---                0---0---
```

In the example, the original samplepoints try to represent a sine wave. The closer the interpolated curve is to a sine wave, the better the interpolation algorithm is. The simpliest interpolation method is linear interpolation. Straight lines are drawn between two adjacent samplepoints:

```
0_                 _0---0_                 _0
  -_             _-       -_             _-
    0_         _0           0_         _0
      -_     _-               -_     _-
        0---0                   0---0
```

Still looks quite "edgy" to be a sine wave. However, the improvement to uninterpolated is significant. There's also a drawback - the frequencies just below the Nyquist frequency get attenuated, even more than without interpolation. Here's the formula for linear interpolation: new = old(int)+(old(int+1)-old(int))*fract, where int means the integer part of sample offset and fract the fractional part.

Next step could be Hermite curve, which gives in every way better quality than linear interpolation:

```
                     ___
0                 _-0   0-_                 0
                 _         _
    0_         _0           0_         _0
      _       _               _       _
       -0___0-                 -0___0-
```

With linear interpolation, you needed to know 2 samplepoints at time to be able to draw the line. With Hermite curve, the number is 4. The interpolation curve goes through the two middle points, and the points 1 and 4 are used in shaping the curve. The formula is a cubic:

```
      new = a*fract^3 + b*fract^2 + c*fract + old(int+0), where:

          3 ( old(int) - old(int+1) ) - old(int-1) + old(int+2)
      a = -----------------------------------------------------
                                  2
                                      5 old(int) + old(int+2)
      b = 2 old(int+1) + old(int-1) - -----------------------
                                                 2
          old(int+1) - old(int-1)
      c = -----------------------
                     2
```

And this one here is where a,b,c,d were solved from:

```
  f(x) = ax^3 + bx^2 + cx + d

   / f(0) = y(0)
  |
  |  f(1) = y(1)
 <
  |  f'(0) = (y(1) - y(-1)) / 2
  |
   \ f'(1) = (y(2) - y(0)) / 2
```

A perfect interpolation also exists. By replacing all the sample points with correctly scaled sinc curves, sin(pi x)/(pi x), and by adding them together, you get exact, perfect interpolation. Here is one of the samplepoints replaced with a scaled sinc curve:

```
                _--0--_                          0
         0    _/       \_
            _/           \_
          _/               \_            __-----__
---------/---------+---------\---------/---------\---------/=========
\__   __/                     \__   __/           ---------
   ---                       0    ---                        0
                                       0
```

Sinc curve is endlessly long, so you'd have to use all the samplepoints in calculation of one interpolated value. A practical solution would be to limit the number of samples to say 1000. It will still be too slow for a realtime application, but it'll give great accuracy. If you insist to use sinc in a realtime interpolation algorithm, try using a windowing function and a low number (at least 6) of sinc curves.

*** Downsampling ***

If you want to downsample (decrease the samplerate), you must first filter away the above Nyquist frequencies, or they will appear as distortion in the downsampled sample.

--- About filter design ---

In the process of filter design, you often need to make compromises. To have sharp edges or steep slopes in the magnitude response, you will need a big, and therefore slow filter. In other words, filters with low number of taps practically always have gently sloping magnitude responses.

In the case of IIR filters, sharp edges in magnitude often mean an ugly (very nonlinear) phase frequency response, and close-to-linear phase response a gently sloping magnitude response. With FIR filters, an attempt to create very sharp edges may cause waving in the magnitudes of nearby frequencies.

IIR filters are great for a realtime routine, because they are fast, their properties (for example cutoff frequency) can be quickly changed in the middle of action, and, they sound like real analog filters. :) The nonlinear phase response of IIR filters usually doesn't matter.

FIR filters could be used where the quality and linear phase are important, for example, in a sample editor. People who filter other signals than sound, often desire linear phase frequency response.

With stereo signal, it is important to have identical phase changes on left and right channels.

Some filters and their stylized magnitude frequency responses:

```
      Lowpass:
      ___________________
                         |___________________
      Highpass:
                          ___________________
      ___________________|
      Bandpass or peak:
                         _
      __________________| |__________________
      Notch, bandreject or bandstop:
      __________________   __________________
                        |_|
      Allpass:
      _______________________________________ (Changes in phase only)
```

If you have a symbolic calculation program, i strongly recommend you to use it in the mechanical calculations, just to make your life easier. "Derive" is an old DOS program, but still very useful.

*** White noise ***

White noise means the sort of noise that has flat spectrum. You can easily create it by using random numbers as samplevalues. If you want to know the magnitude frequency response of a filter, apply it on a long sample of white noise and then run a spectrum analysis on the output. What you see is the magnitude frequency response of the filter.

Another way is to send a one-sample impulse, which originally has a flat spectrum. An impulse looks like this in the sampledata:
0, 0, 0, 0, 1, 0, 0, 0, 0 - where the impulse is the "1" in the middle.

From the two, the impulse thingy is faster, but using white noise can give cleaner-looking results, because errors will be less visible.
For much the same reasons, when you are watching videos, a still picture will look more snowy than the running picture. Taking a spectrum analysis on a long sample is usually done by dividing it to smaller pieces, analyzing them separately and then taking the average of all the analyses. My personal choice here would be the program "Cool Edit 96", which is for Windows.

------------------------------------- Pole-zero IIR filter design ---

Pole-zero method is the easiest way of designing fast and simple IIR filters. When you have learned it, you will be able to design filters by yourself.

Here's the complex "Z-plane", the one used in the pole-zero method:

```
                 pi/2                "|" axis: imaginary
             __---|---__             "-" axis: real
           _/     |     \_           (as always!)
          /       |       \
         |        |        |
     pi -|--------+--------|- 0
         |        |        |
          \_      |      _/
            \__   |   __/
               ---|---
                3/2 pi
```

Imagine the frequencies to be wrapped around the unit circle. At angle 0 we have 0Hz, at pi/2 we have samplerate/4, at pi we have samplerate/2, the Nyquist frequency. You shouldn't care about higher frequencies, since they will never appear in the signal, but anyway, at 2pi (full cycle) we have the sampling frequency.

So if you used sampling frequency 44100 Hz, 0 Hz would be at (1,0), 11025 Hz at (0,1) and 22050 Hz at (-1,0).

What are poles and zeros then?

They are cute little things you can place on the Z-plane, like this:

```
             __---|---__   o         x = pole
           _/  o  |     \_           o = zero
          /       |   x   \
         |        |      x |
        -|-o------+--------|-
         |        |      x |
          \_      |   x  _/
            \__o  |   __/
               ---|---     o
```

There are some rules you have to remember. Poles must always be inside the unit circle, never outside or on it. Zeros can be put anywhere. You can use any number of poles and zeros, but they must all have "conjugate pairs", if they are not positioned on the "-" axis. Conjugate pairs means that if you put for example a zero to (0.6, 0.3), you must put another zero to the conjugate coordinate, (0.6,-0.3). And the same thing with poles.

But hey! What do poles and zeros DO?

Poles amplify frequencies, zeros attenuate. The closer a pole is to a frequency (on the unit circle, remember?), the more it gets amplified. The closer a zero is to a frequency, the more it gets attenuated. A zero on the unit circle completely mutes the frequency it is "sitting on".

Now it could be the right time to try this out yourself. There are free filter design programs around that allow you to play with poles and zeros. One candidate could be: "QEDesign 1000 demo" for Windows. It's somewhere on the Internet, you'll find it.

*** Designing a bandpass filter ***

The simpliest filter designed using pole-zero is the following bandpass filter:

```
             __---|---__             x = pole
           _/     |     \_           o = zero
          /       |    x  \
         |        |        |
        -|--------+--------|-
         |        |        |
          \_      |    x _/
            \__   |   __/
               ---|---
```

Poles amplify frequencies, so you could draw the conclusion that the most amplified frequency is the one at the same angle as the pole. And you are almost right! The only problem comes from the conjugate pole, which also gives its own amplification. The effect is strongest at angles close to 0 and pi, where the distance between the two poles is the smallest. But don't let this confuse you, we'll get back to it later.

So the angle of the pole determines the passband frequency. What's the effect of the absolute value (= radius) then?

As stated, poles amplify frequencies, and the amplification is stronger when the pole is closer to a frequency. In our bandpass filter, increasing the radius of the pole causes the magnitude response to become steeper and passband narrower, as you see here:

   Positions of poles:

```
        __---|---__           __---|---__           __---|---__
      _/     |     \_       _/     |     \_       _/     |  x  \_
     /       |       \     /       | x     \     /       |       \
    |        |x       |   |        |        |   |        |        |
   -|--------+--------|- -|--------+--------|- -|--------+--------|-
    |        |x       |   |        |        |   |        |        |
     \_      |      _/     \_      | x    _/     \_      |      _/
       \__   |   __/           \__   |   __/           \__   |  x__/
          ---|---               ---|---               ---|---
```

   Corresponding magnitude frequency response plots (normalized):

```
 1-|------___          | |    _---_          | |      -            |
   |         --___     | |  _/     \         | |     | |           |
   |              -----| |--        \        | |     | |           |
   |                   | |           \_      | |     / \           |
   |                   | |             \_    | |    |   |          |
   |                   | |               \_  | |    /   \          |
   |                   | |                 --| |   /     |         |
   |                   | |                   | |__-       \____    |
 0-|         |         | |                   | |               ----|
   0Hz      SR/4    SR/2  (SR=samplerate)
```

Let's call the radius "r" from now on. (Some of you might remember the letter "q" from analog, "resonant" filters. This is much the same.) In this case we have the limitation: 0 =< r < 1, since poles must be inside the unit circle. So changing r changes steepness, resonance. This "resonance" - it's not a magic thing, just one frequency being amplified more than others.

*** From poles and zeros to filter coefficients ***

There is a transfer function:

$$H(z) = \frac{a0\ (z-z1)\ (z-z2)\ (z-z3)\ (z-z4)\ \ldots}{(z-p1)\ (z-p2)\ (z-p3)\ (z-p4)\ \ldots}$$

where z is frequency, in the (complex) wrapped-around-the-unit-circle coordinate form. H(z) gives the response (complex!) of the filter at the frequency z. P1, p2, p3 and so on are positions of poles and z1, z2, z3 and so on positions of zeros. A0 is the first input coefficient of the filter. Here's the IIR filter formula again, in case you have forgotten:

$$\begin{aligned} output(t) = {} & a0*input(t) + a1*input(t-1) + a2*input(t-2) + \ldots \\ & + b1*output(t-1) + b2*output(t-2) + b3*output(t-3) + \ldots \end{aligned}$$

Our bandpass filter only has one pole, and its conjugate pair, so we can simplify the transfer function:

$$H(z) = \frac{a0}{(z-p1)\ (z-p2)}$$

$$= \frac{a0}{z^2 - p1z - p2z + p1p2} = \frac{a0}{z^2 + (-p1 - p2)z + p1p2}$$

and replace p1 and p2 with the coordinates of the conjugate poles:

$$p1 = (px, py) = px + ipy, \quad p2 = (px, -py) = px - ipy$$

$$H(z) = \frac{a0}{z^2 + (-(px + ipy) - (px - ipy))z + (px - ipy)(px + ipy)}$$

$$= \frac{a0}{z^2 + (-2px)z + (px^2 - i^2\ py^2)}$$

$$= \frac{a0}{z^2 + (-2px)z + (px^2 + py^2)}$$

$$= \frac{a0*z^0}{z^2 + (-2px)z^1 + (px^2 + py^2)z^0}$$

Let's give the divisor a closer look. Say:

```
              z^2 + (-2px)z^1 + (px^2 + py^2)z^0 = 0        | * z^(-2)

        z^0 + (-2px)z^(-1) + (px^2 + py^2)z^(-2) = 0

        z^0 = -(-2px)z^(-1) - (px^2 + py^2)z^(-2)
```

Powers of z here are actually indexes to the output of the filter:

```
        output(t+0) = -(-2px)output(t-1) - (px^2 + py^2)output(t-2)
```

So we know how to calculate the output side coefficients from the position of the pole:

```
        b1 = 2px
        b2 = - (px^2 + py^2)
```

OK! Let's say the passband frequency is at the Z-plane at position ph:

```
      ph = phx + i phy

      phx = cos(2pi f/SR)
      phy = sin(2pi f/SR)
```

The pole is at the same angle as the frequency on the unit circle, but has radius r. Therefore:

```
      p1 = r*ph = r*(phx, phy) = r*(phx + i phy)
      px = r*phx
      py = r*phy
```

Now that we know how the position of the pole depends on the frequency, we can rewrite the output side coefficients:

```
        b1 = 2px
           = 2*r*phx

        b2 = - (px^2 + py^2)
           = - ((r*cos(2pi f/SR))^2 + (r*sin(2pi f/SR)^2)
           = - r^2
```

But we mustn't forget the dividend (of the transfer function), where powers of z are indexes to the input of the filter:

```
        a0 z^0  ->  a0 input(t+0)
```

This must be added to what we already have solved from the output side:

```
        output(t+0) =  a0 input(t)
                      -(-2px) output(t-1) - (px^2 + py^2) output(t-2)
```

Next we have to decide what to put to a0. This is called normalization. The purpose of a0 is just to scale the output of the filter. In our bandpass filter, we want the amplification at the passband frequency to be 1. So we can write the equation:

```
      |H(ph)| = 1

      |       a0        |
      | --------------- | = 1      | / a0
      | (ph-p1) (ph-p2) |

      |        1        |    1
      | --------------- | = ----
      | (ph-p1) (ph-p2) |    a0

      a0 = | (ph-p1) (ph-p2) |

         = | ((phx + i phy) - r*(phx + i phy))
             ((phx + i phy) - r*(phx - i phy)) |

         = | phx^2 (r-1)^2 + phy^2 (r-1) (r+1) + i (-2 phx phy (r-1)) |

         = sqrt(( phx^2 (r-1)^2 + phy^2 (r-1) (r+1) )^2 +
                ( -2 phx phy (r-1)) )^2 )

         = (1-r) * sqrt(r*(r-4*phx+2)+1)
```

There it is! Now the filter is ready:

```
     SR = sampling frequency
     f = passband center frequency
     r = [0,1)

     phx = cos(2pi f/SR)
     a0 = (1-r) * sqrt(r*(r-4*phx+2)+1)
     b1 = 2*r*phx
     b2 = -r^2

     output(t) = a0 * input(t) + b1 * output(t-1) + b2 * output(t-2)
```

*** Improving the simple bandpass filter ***

We could compensate the effect of the conjugate pole by adding a zero onto the "-" axis, between the poles. For example, if we had poles at coordinates (0.6, 0.5) and (0.6,-0.5), we'd put a zero at (0.6, 0):

```
             __---|---__
           _/     |     \_
          /       |    x  \
         |        |        |
        -|--------+----o---|-
         |        |        |
          \_      |    x _/
            \__   |   __/
               ---|---
```

The transfer function for this is:

```
               a0 (z-z1)
        H(z) = -------------
             (z-p1) (z-p2)
```

The output side coefficients are exactly the same as before. Input side coefficients can be solved like this:

```
        z1 = zx + i*0 = zx

        a0 (z-z1) = a0 (z-zx)  =  a0z - a0zx  =  a0z^1 + (-a0zx)z^0

        -> a0 * input(t+1) + (-a0zx) * input(t)
```

In case you want to use this filter, you should be able to do the normalization yourself. I won't do it here.

*** Words of wisdom ***

It is easy to make a filter more efficient: Double all poles and zeros. The frequency response of the new filter is the square of the old. There are better ways, but this is the easiest.

If you put a zero on a pole, you neutralize both.

A pole outside the unit circle causes the filter to become unstable. A pole on the unit circle may turn the filter into an oscillator.

Large number of poles and zeros means large number of taps.

Zeros affect the input coefficients, poles output.

Poles and zeros must have conjugate pairs, because otherwise you'd get complex filter coefficients and, consequently, complex output signal.

With low r values, the most amplified frequency is not always at the same angle with the pole, because of the effect of the conjugate pole. Try differentiating the magnitude response if you want exact precision.

An IIR filter with no poles is a FIR filter.

```
--------------------------------------- Some pole-zero IIR filters ---

0 =< r < 1 always applies.

*** Bandpass with r ***

Read chapter "IIR filter design using pole-zero method".

*** Notch with r ***

     __---|---__        A zero with a conjugate pair at:
   _/     |     o_      (cos(2pi f/SR), sin(2pi f/SR))
  /       |   x   \
 |        |        |    A pole with a conjugate pair at:
-|--------+--------|-   r * (cos(2pi f/SR), +- sin(2pi f/SR))
 |        |        |
  \_      |   x  _/     Use the frequencies at (1,0) and (-1,0) in
    \__   |   __o       normalization. Depending on f, either of them
       ---|---          has higher amplification.

The higher the r, the narrower the stopband.

*** Lowpass with r ***

This can be done in several ways:

     __---|---__        One or two zeros at: (-1,0)
   _/     |     \_
  /       |    x  \     A pole with a conjugate pair at:
 |        |        |    r * (cos(2pi f/SR), sin(2pi f/SR))
-o--------+--------|-
 |        |        |    Use 0 Hz in normalization.
  \_      |    x _/
    \__   |   __/
       ---|---

     __---|---__        A pole with a conjugate pair at:
   _/     |     \_      r * (cos(2pi f/SR), sin(2pi f/SR))
  /       |       \
 |        |  x     |    Use 0 Hz in normalization.
-|--------+--------|-
 |        |  x     |    Only works with f values below SR/4.
  \_      |      _/
    \__   |   __/       Same as the simple bandpass filter.
       ---|---

The higher the r, the stronger the resonation. Resonant lowpass filter
is surely the most used filter type in synthesizers.
```

```
*** Allpass with r ***

 o   __---|---__        A pole with a conjugate pair at:
   _/     |     \_      r * (cos(2pi f/SR), sin(2pi f/SR))
  /  x    |       \
 |        |        |    A zero with a conjugate pair at:
-|--------+--------|-   (1/r) * (cos(2pi f/SR), sin(2pi f/SR))
 |        |        |
  \_ x    |      _/     Allpass filter has a flat magnitude frequency
    \__   |   __/       response, but a wavy phase response. A zero
 o     ---|---          that has the same angle as a pole, but inverse
                        radius, neutralizes the effect of both in the
magnitude frequency response, but not in the phase frequency response.
The higher the r, the stronger the effect. One use for allpass filters
is to make the phase response of an IIR filter linear, by using a
correctly parameterized allpass filter in serial with the IIR filter.

*** Highpass with r ***

     __---|---__           __---|---__      Almost the same as lowpass.
   _/     | x   \_       _/     |     \_    Normalization frequency
  /       |       \     /       |       \   (-1,0), zeros, if any, at
 |        |        |   |     x  |        |  (1,0).
-|--------+--------o- -|--------+--------|-
 |        |        |   |     x  |        |
  \_      |      _/     \_      |      _/
    \__   | x __/         \__   |   __/
       ---|---               ---|---
```

*** Impulse, sinc ***

If you read about sinc interpolation in the chapter "Interpolation of sampled sound", you know that you can replace a single sample peak (= impulse) in the sampledata with a correctly stretched sinc function. Correctly stretched means amplitude*sinc(t).

When you run a spectrum analysis on an impulse, you get a flat spectrum with upper limit at samplerate/2, the Nyquist frequency. Because impulse = sinc, this is also the spectrum of sinc:

```
       |___________
       |           |
       |           |
       |           |                     "|" axis: amplitude
      0|___________|____________         "-" axis: frequency
       0          SR/2
```

You could draw the conclusion that you get the sinc function if you sum together all the frequencies from 0 to SR/2, and divide the sum by the number of frequencies, to fulfil the equation sinc(0) = 1. And you'd be right.

From the spectrum analysis, you know that all the frequencies summed together have equal amplitudes. But what's their phase at the center of the impulse? Sinc function is symmetrical around x=0, so is cosine - so sinc must be made of cosines. If you test this with about 100 cosines, you get a pretty close approximation of sinc.

The sum of all frequencies from 0 to 1 (comparable to SR/2), divided with their number, can be written as: (Here "oo" means infinite)

$$\lim_{n \to \infty} \frac{\displaystyle\sum_{i=0}^{n} \cos\left(x \frac{i}{n}\right)}{n+1} = \frac{\sin(x)}{x} \to \frac{\sin(\pi t)}{\pi t} = \operatorname{sinc}(t) \qquad (x \to \pi t)$$

As done above, x must be replaced with "pi t", because the cycle length of sin is 2 pi, which must be stretched to 2 (which is the "wavelength" of the Nyquist frequency in the sampledata).

*** Phase shift ***

What if we replaced the cosines with sines? Lets try it! There's a universal formula (which, btw, i invented myself) we can use:

$$\lim_{n\to oo} \frac{\sum_{i=0}^{n} f\left(x\frac{i}{n}\right)}{n+1} = \frac{\int_{0}^{x} f(x)\,dx}{x}$$

Therefore:

$$\lim_{n\to oo} \frac{\sum_{i=0}^{n} \sin\left(x\frac{i}{n}\right)}{n+1} = \frac{\int_{0}^{x} \sin(x)\,dx}{x} = \frac{1-\cos(x)}{x} \xrightarrow{(x \to \pi t)} \frac{1-\cos(\pi t)}{\pi t}$$

Now, if we replace all the impulses in the sound with this new function, we actually perform a -90 degree phase shift! This can be done by creating a FIR filter, where the coefficients are taken from this new function: (1-cos(pi t))/(pi t), but in reverse order, by replacing t with -t, so it becomes: (cos(pi t)-1)/(pi t).

Here's an example that explains why it is necessary to use -t instead of t: Say you want to replace all the impulses in the signal with the sequence 1,2,3. If the input signal is 0,1,0, common sense says it should become 1,2,3. If you just use 1,2,3 as filter coefficients in that order, the filtered signal becomes:

```
in(0-1)*1+in(0)*2+in(0+1)*3,
in(1-1)*1+in(1)*2+in(1+1)*3,
in(2-1)*1+in(2)*2+in(2+1)*3  = 0+0+3,0+2+0,1+0+0 = 3,2,1
```

Which is not what you asked for! But if you use coefficients 3,2,1, you get the right result, Ok, back to the -90 degree phase shift filter...

When you are picking the filter coefficients from (cos(pi t)-1)/(pi t), at t=0 you unluckily get a division by zero. Avoid this by calculating the limit t->0, on paper, or with a math proggy. If you use your brains a little, you notice it is 0, because the filter formula is a sum of sines, and sin(0)=0, so at t=0 it is a sum of zeros.

Like sinc, our new function has no ending, so a compromise must be made in the number of taps. This causes waves in the magnitude response, and attenuation of the very lowest and highest frequencies. By applying a windowing function to the coefficients, you can get rid of the waves, but i don't know anything that would help with the attenuation, except more taps. The windowing functions used with FFT work here also. The center of the windowing function must be at t=0, and it must be stretched so that the edges are lay on the first and the last tap.

You can also get a phase shift of any angle "a":

$$\lim_{n\to oo} \frac{\sum_{i=0}^{n} \cos(x\frac{i}{n}+a)}{n+1} = \frac{\int_{0}^{x} \cos(x+a)\,dx}{x}$$

$$(x \to -\pi t)$$

$$= \frac{\sin(x+a) - \sin(a)}{x} \to \frac{\sin(-\pi t + a) - \sin(a)}{-\pi t}$$

Note that reversing t has already been done here, so we can take the coefficients directly from this formula. The limit t->0 is naturally cos(a), because all the cosines added together had phase "a" at x=0.

In case you didn't yet realize it, the main idea in FIR filter making is to create a function that contains the frequencies you want to pass the filtering. The amplitudes of the frequencies in the function directly define the magnitude frequency response of the filter.
The phases of the frequencies define the phase response.

Reversing the coefficients is only necessary with phase shifting filters, because filters that do not introduce a phase shift of any kind are symmetrical around t=0.

*** Defining the frequency range included ***

If you use sinc as your filter coefficient formula, you actually do no filtering, because all the frequencies from 0 to Nyquist are equally presented in sinc. Here you'll see how you can select which frequencies will be present in your filter coefficient formula. Remember where we originally got sinc from:

$$\frac{\int_{0}^{x} \cos(x)\,dx}{x} = \frac{\sin(x)}{x} \rightarrow \frac{\sin(\pi t)}{\pi t} \qquad (x \rightarrow \pi t)$$

In the integral, the upper limit (1x) actually represents the highest frequency included (1), and the lower limit (0x) the lowest (0). So if you want a formula for a bandpass filter, you can write:

$$\frac{\int_{\text{bottom } x}^{\text{top } x} \cos(x)\,dx}{x}$$

where top and bottom are the cutoff frequencies in such way that 1 means the Nyquist frequency, and 0 means 0Hz. Now just put there whatever frequencies you want, calculate, and replace x with (pi t). There's your filter coefficient formula ready! For example, if you want to make a halfband lowpass filter (which naturally has cutoff frequency at samplerate/4, same as Nyquist frequency / 2):

$$\frac{\int_{0\,x}^{0.5\,x} \cos(x)\,dx}{x} = \frac{\int_{0}^{0.5\,x} \cos(x)\,dx}{x} = \frac{\sin(x/2)}{x} \rightarrow \frac{\sin(\pi t/2)}{\pi t} \qquad (x \rightarrow \pi t)$$

To create multi-band filters, you can combine several bandpass filter formulas by adding them together.

*** The equalizer example ***

If you want to make an equalizer (a filter that allows you to define the magnitudes for certain frequencies), you probably sum together a lot of bandpass filter formulas, scaled by the magnitudes you want for the frequency segments. This gives you a magnitude response that looks very much like as if it was made of bricks:

(*=defined frequency)

```
             ___*___                              ____*____
            |       |                    ____*___|         |
        *___|       |___*____           |                  |___*
                             |          |
                             |_____*____|
```

Maybe you'd want it to look more like this instead:

```
             __-*-__                            ____--*-__
          __-       -__                    __*--          --__
        *-             -*-___           __-                   -*
                             --___   __-
                                  -*-
```

There are three ways. The first way is to "use smaller bricks", meaning that you divide the frequency into narrower-than-before segments and use interpolation to get the magnitude values for the new narrow bandpass filters you then combine.

The second way is to define a polynomial (like ax^3+bx^2+cx+d) that has the wanted characteristics (and where x=1 represents freq=SR/2), and to make the magnitude response of your filter to follow it. This is possible.

The third way is to add together several bandpass "ramp" filter formulas. In the magnitude response this solution looks like straight lines drawn between the adjacent defined frequencies. This is also possible, and, in my opinion, the best solution.

*** Polynomial shaped magnitude frequency response ***

In sinc, all the cosine waves added together have equal amplitudes, as you see here - all the frequencies are treated equally:

$$\sum_{i=0}^{n} \cos\left(x\frac{i}{n}\right) \qquad (x \to \pi t)$$

$$\lim_{n\to\infty} \frac{\sum_{i=0}^{n} \cos\left(x\frac{i}{n}\right)}{n+1} = \frac{\sin(x)}{x} \to \frac{\sin(\pi t)}{\pi t} = \mathrm{sinc}(t)$$

You can change this by putting there a function g() that defines the amplitudes of the cosine waves of different frequencies:

$$\lim_{n\to oo} \frac{\sum_{i=0}^{n} \cos(x\frac{i}{n}) * g(\frac{i}{n})}{n+1}$$

If the function g(x) is form a*x^b, the calculations go like this:

$$\lim_{n\to oo} \frac{\sum_{i=0}^{n} \cos(x\frac{i}{n}) * a*(\frac{i}{n})\hat{}b}{n+1} = \frac{\int_0^x \cos(x)*ax\hat{}b\ dx}{x\hat{}(1+b)}$$

For a simple example, if we want the magnitude frequency response to be a straight line, starting from 0 at 0Hz, and ending at 1 at SR/2, we define g(x) = x:

```
                  y

                /|      _*
                 |    _/  (1,1)
                 |  _/
                 |_/
        --------=*--------> x
              _/ |(0,0)
            _/   |
          _/     |
         /       |
```

And the filter coefficient formula calculations for this:

$$\frac{\int_0^x \cos(x)*1*x\hat{}1\ dx}{x\hat{}(1+1)} = \frac{\cos(x)+x*\sin(x)-1}{x\hat{}2} \xrightarrow{(x\ \to\ \text{pi } t)} \frac{\cos(\text{pi } t)+(\text{pi } t)*\sin(\text{pi } t)-1}{(\text{pi } t)\hat{}2}$$

That's it!

In other cases, to get the formula for a full polynomial, do the calculations for each of its terms (a*x^b) separately and sum the results.

*** Bandpass magnitude-ramp ***

Here's an example of the magnitude frequency response of a ramp filter:

```
                          __-|
                      __--   |
                   _--       |
                  |          |
        Mag:      |          |
         0 ___________|          |__________________ frequency ->
```

To make a bandpass ramp, you must first define the polynomial g(x) that describes how the magnitude behaves inside the bandpass limits. The magnitude is linear inside the limits, so the polynomial g(x) must be form c*x+d. C and d can be solved from the equations:

$$\begin{cases} g(x1) = y1 \\ g(x2) = y2 \end{cases}$$

where x1 is the lower frequency limit, and x2 the higher. Y1 and y2 are the magnitudes of the limit frequencies. Remember that here x=1 equals frequency=SR/2. OK, here are c and d solved:

$$c = \frac{y1 - y2}{x1 - x2} \qquad d = \frac{x1 * y2 - x2 * y1}{x1 - x2}$$

G(x)=c*x+d is a polynomial, and you already know how to make the magnitude frequency response have the same shape (Section "Polynomial shaped magnitude frequency response") as a polynomial. You also already know how to include only limited range of frequencies (Section "Defining the frequency range included") in your coefficient formula. Combine this knowledge, and you can write the coefficient formula for the ramp bandpass filter:

$$\frac{\int_{x1\,x}^{x2\,x} \cos(x)*c*x^1\,dx}{x^{(1+1)}} + \frac{\int_{x1\,x}^{x2\,x} \cos(x)*d*x^0\,dx}{x^{(1+0)}} =$$

$$c\,\frac{\cos(x2*x)+x2*x*\sin(x2*x)-\cos(x1*x)-x1*x*\sin(x1*x)}{x^2} +$$

$$d\,\frac{\sin(x2*x)-\sin(x1*x)}{x} \qquad \text{(And then, as always, } x \to \pi\, t\text{)}$$

A note about implementing the equalizer... If the equalizer is to be adjustable realtime, recalculating the whole equalizer filter formula with all the trigonometric functions may turn out too heavy. It may be better to precalculate coefficients for several overlapping filters, for example these for a three-channel equalizer:

```
              __                                      __
                \                 /\                 /
                 \               /  \               /
                  \             /    \             /
                   \______   __/      \__   ______/
```

When calculating the coefficients for the whole equalizer, just pick the corresponding coefficients from these, scale according to the "equalizer sliders", and sum.

*** Windowing ***

If you take your FIR filter coefficients directly from your "filter formula", you get a very wavy magnitude response. The reason is simple: The number of coefficients is limited, but the filter formula is not, it continues having nonzero values outside the range you are using for the coefficients.

A windowing function helps. Not using a windowing function is the same thing as using a rectangular (= flat inside its limits) windowing function. Using a windowing function means that you multiply the values taken from your infinitely long filter formula by the corresponding values taken from your finitely long windowing function, and use the results as filter coefficients.

Here are some windowing functions, and the produced magnitude responses of a FIR lowpass filter with a low number of taps, illustrated:

```
          rectangular              cos^2                 cos^4
 1_| _____________________          ___                   ___
   |                             _--   --_               /   \
   | time->                    _/         \_           _/     \_
 0_|                       __--             --__ ____--         --____
     _      __
 1_|  \    /  \            ______--_             _______
   |   \__/    |                    \                   --
   |           |                     |                    \
   |           |                     |                     |
   |           | __  __  _           |                      \
 0_| freq->    |/  \/  \/             \_--______             --_______
```

As you see, the steeper the cutoff, the more waves you get. Also, if we'd look at the magnitude responses in dB scale, we'd notice that from the three, cos^4 gives the best stopband (= the frequency range that should have 0 magnitude) attenuation.

Mathematically, multiplication in the time domain is convolution in the frequency domain, and windowing is exactly that. (Also, multiplication in the frequency domain is convolution in the time domain.)

I hope i didn't slam too many new words to your face. Time domain means the familiar time-amplitude world, where we do all the FIR and IIR filtering. The frequency domain means the frequency-amplitude&phase world that you get into through Fourier transformation. And convolution; In the time domain, FIR filtering is convolution of the input signal with the filter coefficients. Say you convolute 0,1,0,0,2,0,1,0 with 1,2,3 (where 2 is at the center): You'll get 1,2,3,2,4,7,2,3. If you understand this example, you surely understand convolution too.

Ideally (impossible), there would be no windowing, just the constant value 1 infinitely in time. And a steady constant value in the time domain is same as 0Hz in the frequency domain, and if you (in the frequency domain) convolute with 0Hz, it is the same as no convolution.

Convolution in the frequency domain equals to multiplication in the time domain, and convolution in the time domain equals to multiplication in the frequency domain. Sounds simple, eh? But note that in this "frequency domain", there are positive AND NEGATIVE frequencies. You'll learn about those in chapter "Positive and negative frequencies".

*** Words of wisdom ***

You get flat (but not necessarily continuous) phase response if your filter (=filter coefficients) is symmetrical around t=0 or around (0,0), even if you limit the number of coefficients and/or window them.

Sometimes you can optimize your filter code a lot. Some coefficients may turn zero, so you can skip their multiplications. If your filter is symmetrical around t=0, you can instead of "input(t)*a+input(-t)*a" write "(input(t)+input(-t))*a)". If your filter is symmetrical around (0,0), replace "input(t)*a-input(-t)*a)" with "(input(t)-input(-t))*a".

Sinc(t) is 1 at t=0, and 0 at other integer t values.

Calculating the limit t->0 is very simple. If your filter formula was originally a sum cosines (meaning it's not a phase shift filter), the limit t->0 is simply the area of the magnitude frequency response, in such way that the area of "no filtering" is 1.

-- Filter implementation ---

The actual filter implementation (after possible coefficient calculations) depends much on how the input data is fed to the filter. I can see three cases:

*** Case 1 ***

You have the whole input data in front of you right when you start. A sample editor is a good example on this.

This is the easiest case. With FIR filters, just take values from the input data, multiply with coefficients and sum, like this: output(t) = a0*input(t-2) + a1*input(t-1) + a2*input(t) + a3*input(t+1) + a4*input(t+2). The only problem is what to do at the start and at the end of the input table, because reading data from outside it would only cause problems and mispredictability. A lazy but well working solution is to pad the input data with zeroes, like this:

```
    [-----Filter------]    (Situation: t = 0)
    00000000[------------------Input-data------------------]00000000

                          (Situation: t = max)   [-----Filter------]
    00000000[------------------Input-data------------------]00000000
```

This is how it's mostly done with FFT filtering. With FIR filters, it isn't that hard to write a version of the routine that only uses a limited range of it's taps, like this:

```
            [er------]     (Situation: t = 0)
            [------------------Input-data------------------]

            [lter------]   (Situation: t = 2)
            [------------------Input-data------------------]

                          (Situation: t = max)    [-----Fil]
            [------------------Input-data------------------]
```

and to use that version at the start and at the end. For this, it is easiest if you have a table of coefficients instead of hard-coding them into the routine.

*** Case 2 ***

Data is fed to the filter in small chunks, but it is continuous over the chunk borders. This is the most common situation in programs handling realtime audio.

```
*** Case 3 ***
```

One sample at a time. Case 2 can be treated as this, because the chunks can always be chopped into single samples.

It is a fact that you cannot use future inputs in this case, so a FIR filter would have to be of form such as: output(t) = a0*input(t-4) + a1*input(t-3) + a2*input(t-2) + a3*input(t-1) + a4*input(t). Clearly this kind of a filter creates a delay, but that's just a thing you have to learn to live with. Also, you only get in one sample at a time, which is not enough for filtering, so you have to store the old input values somehow. This is done using a circular buffer. The buffer is circular, because otherwise you'd soon run out of memory. Here's a set of pictures to explain the scheme:

```
                         W (= Write-to position)
               time=0    [-------Circular-buffer--------]
                         ]         [--------Filter-------

                              W
               time=5    [-------Circular-buffer--------]
                         -----]         [--------Filter--

                                                   W
              time=26    [-------Circular-buffer--------]
                             [--------Filter-------]

                         W
   time=buffer length    [-------Circular-buffer--------]
                         ]         [--------Filter-------
```

The buffer must be at least as long as the filter, but it is practical to set the length to an integer power of 2 (In the above example: 2^5=32), because then you can use the binary AND operation to handle pointer wrapping always after increasing or decreasing one (In the above example, AND with 31). Even better, use byte or word instructions, and wrapping will be "automatically" handled in over/underflows caused by the natural limits of byte or word.

Note that the buffer should be filled with zeroes before starting.

A similar circular buffer scheme is also often the best solution for implementing the output part of an IIR filter, no matter how the input part was realized.

-------------------------------- Positive and negative frequencies ---

There are both positive and negative frequencies. Until now we haven't had to know this, because we have been able to do all the calculations by using sines as frequencies. Don't be fooled that positive frequencies would be sines, and negative ones something else, because that is not the case.

In all real (meaning, not complex) signals, positive and negative frequencies are equal, whereas in a complex signal the positive and negative frequencies don't depend on each other. A single sinewave (real) consists of a positive and a negative frequency. So any sine frequency could be expressed as a sum of its positive and negative component. A single, positive or negative, frequency is:

$$\text{amplitude} * e^{i\,(\text{frequency } x + \text{phase})}$$

and could also be written as:

$$\text{amp} * (\ \cos(\text{freq } x + \text{phase}) + i \sin(\text{freq } x + \text{phase})\ )$$

As stated, a sinewave consists of a positive and a negative frequency component. Here's the proof: (The phase of the negative frequency must also be inverted, because it "rotates" to the other direction)

$$e^{i\,(+\text{freq } x + \text{phase})} + e^{i\,(-\text{freq } x - \text{phase})}$$

$$= \cos(\text{freq } x + \text{phase}) + i \sin(\text{freq } x + \text{phase})) + \cos(\text{freq } x + \text{phase}) - i \sin(\text{freq } x + \text{phase}))$$

$$= 2 \cos(\text{freq } x + \text{phase})$$

As you see, the imaginary parts nullify each others, and all that remains is the real part, the sine wave. Amplitude of the sine wave is the sum of the amplitudes of the positive and the negative frequency component (which are the same). This also proves that in any real signal, positive and negative frequencies are equal, because a real signal can be constructed of sine waves.

The complex Z-plane is a good place to look at positive and negative frequencies:

```
                              +pi/2
                           __---|---__
                         _/     |     \_
                        /       |       \
                   +pi  |       |       |
          (Nyquist)  or -|--------+--------|- 0
                   -pi  |       |       |
                        \_      |      _/
                          \__   |   __/
                             ---|---
                              -pi/2
```

Positive frequencies are on the upper half of the circle and negative frequencies on the lower half. They meet at angles 0 and the Nyquist frequency.

*** Aliasing ***

Aliasing usually means that when you try to create a sine wave of a frequency greater than the Nyquist frequency, you get another frequency below the Nyquist frequency as result. The new frequency looks like as if the original frequency would have reflected around the Nyquist frequency. Here's an example:

```
            What you want: |       | f       (f = sine wave frequency)
             What you get: |     f |
                           0      SR/2
```

The cause of aliasing can be easily explained with positive and negative frequencies. The positive component of the sine wave actually gets over the Nyquist frequency, but as it follows the unit circle, it ends up on the side of negative frequencies! And, for the same reasons, the negative component arrives on the side of positive frequencies:

```
                             __---|---__
                           -f     |     \_
                          /       |       \
                         |        |        |
                        -|--------+--------|-
                         |        |        |
                          \       |      _/
                           +f__   |   __/
                               ---|---
```

The result is a sine wave, of frequency SR-f.

*** Analytic signal ***

It is sometimes needed to first create a version of the original signal that only contains the positive frequencies. A signal like that is called an analytic signal, and it is complex.

How does one get rid of the negative frequencies? Through filtering! It is possible to do the job with an IIR filter that doesn't follow the conjugate-pair-poles-and-zeros rule, but a FIR filter is significantly easier to create. We'll use the old formula that we first used to create sinc:

```
                        n
                       ___               x
                       \       i          /
                        >  f(x---)       |
                       /__     n         | f(x) dx
                                        /
                      i = 0              0
                 lim  ------------  =  ----------
                n->oo     n+1             x
```

but this time, instead of cosines, only including the positive frequencies:

$$\qquad\qquad\qquad\qquad\qquad\qquad (x \to -\pi t)$$

$$\frac{\int_0^x e^{ix}\,dx}{x} = \frac{\sin(x)}{x} + i\,\frac{1-\cos(x)}{x} \to \frac{\sin(\pi t)}{\pi t} + i\,\frac{\cos(\pi t)-1}{\pi t}$$

As you see, the filter coefficients are complex. We should also halve the amplitude of the positive frequency (it should be half of the amplitude of the cosine, because the negative component is gone) but that's not necessary, because it'd only scale the magnitude.

To convert the complex analytic signal back to real, just throw away the imaginary parts and all the frequencies get a conjugate (on the z-plane) pair frequency. Here the amplitudes drop to half, but as we skipped the halving in the filtering phase, it is only welcome.

The real to analytic signal conversion could also be a good spot for filtering the signal in other ways, because you can combine other filters with the negative frequency removal filter.

-- Frequency shifting ---

*** Amplitude modulation ***

Amplitude modulation means multiplying two signals. All samplepoints in the modulated signal are multiplied by the corresponding samplepoints in the modulator signal. Here's an example:

```
                             ___     ___     ___
                   Original:    |   |   |   |   |
                                |___|   |___|   |___
                             _________
                  Modulator:          |
                                      |_____________
                             ___     _   ___     ___
                     Result:    |   | | |   |   |
                                |___| |_|   |___|
```

What happens if we modulate a signal with a sinewave? The original signal is (as we have learned) a sum of frequecy components, sinewaves of various frequencies, amplitudes and phases. Note that the signal we are talking about here is real, not complex. Say "sNUMBER" is one of the frequency components. So, we can write the original signal as:

```
                  s0 + s1 + s2 + s3   (and so on...)
```

Now, if we multiply this signal with the modulator signal "m", we get:

```
                    (s0  + s1   + s2   + s3) * m
                  = s0*m + s1*m + s2*m + s3*m
```

This is good, because as you see, it's the same as if the frequency components were processed separately, so we can also look at what happens to each frequency component separately. A frequency component can be written as:

```
                        amp * cos(f x + a)
```

where amp is the amplitude, f the frequency and a the phase. The modulator sine can be written the same way (Only added the letter m):

```
                       mamp * cos(mf x + ma)
```

Multiply those and you get:

```
        amp * cos(f x + a) * mamp * cos(mf x + ma)

        amp mamp                       amp mamp
      = -------- cos((f+mf)x + a+ma) + -------- cos((f-mf)x + a-ma)
           2                              2
```

If we discard the phase and amplitude information, we get:

```
                    cos((f+mf)x) + cos((f-mf)x),
```

which is two frequencies instead of the origial one.

Here's a graph that shows how the frequencies get shifted and copied. The original frequency is on the "-" axis and the resulting frequency/frequencies on the "|" axis:

```
              No modulation:         Modulated:

               |         _/            |     _/
               |       _/              |   _/      _/
               |     _/                | _/      _/
               |   _/             ____|/_______/________modulator____
               | _/                    |\_   _/          frequency
               |/___________          |__\_/________
              0,0                    0,0
```

In the graph "Modulated", the frequencies that would seem to go below zero, get aliased and therefore reflect back to above zero. In sampled signal, the Nyquist frequency also mirrors the frequencies.

*** Frequency shifting ***

With some tweaking and limitations, you could make a frequency shifter by using sinewave modulation, but there's a better way.

Let's try modulating the signal with e^(i mf x) instead of cos(mf x). Phases and amplitudes are irrelevant, so i've just ignored them. (I hope you don't mind!) Let's see what happens to a single positive/negative frequency when it is modulated:

```
                     e^(i f x) * e^(i mf x)
                   = e^(i f x + i mf x)
                   = e^(i (f+mf)x)
```

The answer is very simple. The original frequency got shifted by the modulator frequency. Notice how the rule "Multiplication in the time domain is convolution in the frequency domain." applies here also. Here's an example on the z-plane unit circle. p0, p1, p2 are the positive frequencies and n0, n1, n2 their negative conjugate frequencies. Say the modulator frequency rotates the frequencies 1/4 full cycle counterclockwise:

```
         Modulator:          No modulation:          Modulated:

        __---m---__           __---p1--__           _p0--|---n0
      _/     |     \_       _/     |     \_       _/     |     \_
     /       |       \     p2      |       p0    /       |       \
    |        |        |   |        |        |   |        |        |
   -|--------+--------|- -|--------+--------|- -p1-------+--------n1
    |        |        |   |        |        |   |        |        |
     \_      |      _/     n2      |      _n0    \_      |      _/
       \__   |   __/         \__   |   __/         \__   |   __/
          ---|---               ---n1--              p2--|---n2
```

In the modulated signal, the original pair frequencies (like p0 and n0) are no longer conjugate pairs. That's bad. Another bad thing is that negative frequencies get on the side of positive frequencies and vice versa.

But if we first filter all the negative, and those of the positive
frequencies that would arrive on the wrong side of the cirle, and then
modulate the filtered signal: (The filter formula is in the chapter
"A collection of FIR filters" in section "Combined negative frequency
removal and bandpass")

```
         Original:             Filtered:       Filtered & modulated:

        __---p1--__           __---p1--__           _p0--|---__
      _/     |     \_       _/     |     \_       _/     |     \_
     p2      |       p0    /       |       p0    /       |       \
    |        |        |   |        |        |   |        |        |
   -|--------+--------|- -|--------+--------|- -p1-------+--------|-
    |        |        |   |        |        |   |        |        |
     n2      |      _n0    \_      |      _/     \_      |      _/
       \__   |   __/         \__   |   __/         \__   |   __/
          ---n1--             ---|---               ---|---
```

Now it looks better! To make this filtered & modulated complex signal
back to real again, just discard the imaginary part and all the
frequencies get a conjugate pair:

```
               With imaginary:      Imaginary discarded:

                 _p0--|---__            _p0--|---__
               _/     |     \_        _/     |     \_
              /       |       \      /       |       \
             |        |        |    |        |        |
            -p1-------+--------|-  p1n1------+--------|-
             |        |        |    |        |        |
              \_      |      _/      \_      |      _/
                \__   |   __/          \__   |   __/
                   ---|---               n0--|---
```

*** Harmonics ***

For most sounds, frequency shifing doesn't do a very good job, because they consist of a fundamental frequency and its harmonics. Harmonic frequencies are integer multiples of the fundamental frequency. After you have shift all these frequencies by the same constant frequency, they no longer are harmonics of the fundamental frequency. There are ways to do scaling instead of shifting, but just scaling the frequencies would be same as resampling, and resampling also stretches the sound in time, so it has to be something smarter. The main idea is to divide the sound into narrow frequency bands and to shift/scale them separately.

OK, so frequencies usually come with harmonics - Why? Just think where sounds in nature originate from: vocal cords in our throat, quitar strings, air inside a flute... All vibrating "objects", and you have probably learned at school that objects have several frequencies in which they "like to" vibrate, and those frequencies are harmonics of some frequency.

What happens in those objects is that they get energy from somewhere (moving air, player's fingers, air turbulence), which starts all kinds of vibrations/frequencies to travel in them. When the frequencies get reflected, or say, go around a church bell, they meet other copies of themselves. If the copies are in the same phase when they meet, they amplify each other. In the opposite phases they attenuate each other. Soon, only few frequencies remain, and these frequencies are all harmonics of same frequency. Like so often in physics, this is just a simplified model.

A note about notation! :) The fundamental frequency itself is called the 1st harmonic, fundamental*2 the 2nd, fundamental*3 the 3rd, and so on.

*** Chromatic scale ***

In music, harmonics play a very important role. The "chromatic scale", used in most western music, is divided into octaves, and each octave is divided into 12 notes. The step between two adjanced notes is called a halftone. A halftone is divided into hundred cents.

An octave up (+12 halftones) means doubling the frequency, an octave down (-12 halftones) means halving it. If we look at all the notes defined in the chromatic scale on a logarithmic frequency scale, we note that they are evenly located. This means that the ratio between the frequencies of any two adjacent notes is a constant. The definition of octave causes that constant^12 = 2, so constant = 2^(1/12) = 1.059463.

If you know the frequency of a note and want the frequency of the note n halftones up (Use negative n to go downwards) from it, the new frequency is 2^(n/12) times the old frequency. If you want to go n octaves up, multiply by 2^n.

But why 12 notes per octave?

As said, harmonics are important, so it would be a good thing to have a scale where you can form harmonics. Let's see how well the chromatic scale can represent harmonics... The first harmonic is at the note itself: +0 halfnotes = 1. The second harmonic is at +1 octave = 2. The third harmonic is very close to +1 octave +7 halftones = +19 halftones = 2^(+19/12) = 2.996614. And so on... Here's a table that shows how and how well harmonics can be constructed:

Harmonic	Octaves+Halftones		Halftones	Error(Cents)	Acceptable
1	0	0	0	0	Yes
2	1	0	12	0	Yes
3	1	7	19	-1.955	Yes
4	2	0	24	0	Yes
5	2	4	28	+13.686	Yes
6	2	7	31	-1.955	Yes
7	2	10	34	+31.174	No
8	3	0	36	0	Yes
9	3	2	38	-3.910	Yes
10	3	4	40	+13.686	Yes
11	3	6	42	+48.682	No
12	3	7	43	-1.955	Yes
13	3	8	44	-40.528	No
14	3	10	46	+31.174	No
15	3	11	47	+11.731	Yes
16	4	0	48	0	Yes
17	4	1	49	-4.955	Yes
18	4	2	50	-3.910	Yes
19	4	3	51	+2.487	Yes
20	4	4	52	+13.686	Yes
21	4	5	53	+29.219	No
22	4	6	54	+48.682	No
23	4	6	54	-28.274	No
24	4	7	55	-1.955	Yes
25	4	8	56	+27.373	No

Not bad at all! The lowest harmonics are the most important, and as you see, the errors with them are tiny. I also tried this with other numbers than 12, but 12 was clearly the best of those below 30. So, the ancient Chinese did a very good choice! The above table could also be used as reference when tuning an instrument, for example a piano (bad example - no digital tuning in pianos), to play some keys and chords more beautifully, by forcing some notes to be exact harmonics of some other notes.

A common agreement is that one of the notes, "middle-a", is defined to be at 440Hz. This is just to ensure that different instruments are in tune.

```
------------------------------------------------------------ Flanger ---
```

Flanger is simply:

```
        output(t) = input(t) + input(t-d)
```

where d is the length of the variable delay. D values have a lower limit, and the variation comes from sine:

```
  d = something1 + something2 * sin(t something3)
```

Because d is not integer, we must interpolate. Most probably, annoying high frequency hissing still appears. It can be reduced by lowpass filtering the delayed signal.

-- Wavetable synthesis ---

Wavetable synthesis means that the instruments being played are constructed of sampled sound data. MOD music is a well-known example. Also most of the basic home synthesizers use wavetable synthesis.

*** Pitching ***

Say you have a sampled instrument, and want to play it at frequency f = 440Hz, which is middle A in the chromatic scale.

To be able to do this, you need to know A) the samplerate of the sample and the frequency of the sampled instrument, or B) the wavelength of the instrument expressed as number of samples (doesn't have to be integer). So you decide to precalculate the wavelength to speed up the realtime routines a little:

```
        ol = sample_SR/sample_f = say 256.
```

The samplerate of your mixing system, "SR", is 44100Hz. Now that you know this, you can calculate the new wavelength, the one you want (number of samples):

```
        nl = SR/f
           = 44100Hz / 440Hz = 100.22727
```

In the mixer innerloop, a "sample offset" variable is used in pointing to the sampledata. Every time a value is read from the sampledata and output for further mixing, sample offset is advanced by adding variable A to it. Now we must define A so that ol (256) is stretched (here shortened) to nl (100.22727), in other words, so that for ol samplepoints in the sampledata, you produce nl output values:

```
        A = ol/nl
          = 256 / 100.22727 = 2.55420
```

Everything on one line:

```
        A = ol/(SR/f) = ol*f/SR
```

That's it! By using A as the addvalue, you get the right tone.

*** Click removal ***

There are some situations when unwanted clicks appear in the output sound of a simple wavetable synthesizer:

* Abrupt volume (or panning/balance) changes.
* A sample starts to play and it doesn't start from zero amplitude.
* A sample is played to the end and it doesn't end at zero amplitude. (Biased sampledata or badly cut out sample!)
* A sample is killed abruptly, mostly happens when new notes kill the old ones.
* Poor loops in a sample.

And what does help? Here's some advice:

Volume changes must be smoothed, maybe "ramped", so that it'll always take a short time for the new volume to replace the old. Clicky sample starts can be muffled, meaning that the volume is first set to zero and then slided up. This could of course be done beforehand too, and some think muffling sample starts is wrong, because the click may be deliberate. Some drum sounds lose a lot of their power when the starts are muffled.

Another case is when the playing of a sample is not started from its beginning. That will most probably cause a click, but muffling is not the only aid - starting to play from the nearest zero crossing also helps. Abrupt sample ends should also be faded down. This may require some sort of prediction, if you want to fade down the sound before it's "ran over" by another sound. This prediction can be made by using a short information delay buffer. It may be easier to just use more channels, to allow the new sound to start while the other one is being faded out in the background, on another channel.

When the sampledata ends at a value other than zero, the cause may be that the sampledata is not centered around the zero level, or that the creator of the sample has just cut the end of the sample away. The easiest way to fix this is to fade out the end of the sample beforehand. However, this is not always possible.

--- Shuffling IIR equations ---

- Symmetric form
- Turning an IIR filter backwards
- Getting rid of output(t+n)
- Getting rid of input(t+n)
- FIR frequency response
- IIR frequency response.

Olli wrote he tried to make his text as down-to-earth as possible.
Well, here's a more mathematical approach.

But I've still tried to make this intuitive and FUN rather than boring
myself with lengthy proofs.

This also means that there may be errors, most probably in signs.

*** Symmetric form ***

Say you have this IIR filter:

```
output(t) =   a0*input(t) + a1*input(t-1) + a2*input(t-2)
            + b1*output(t-1) + b2*output(t-2) + b3*output(t-3)
```

You can put its equation to this symmetric form:

```
a0*input(t) + a1*input(t-1) + a2*input(t-2)
 =   output(t) + (-b1)*output(t-1)
   + (-b2)*output(t-2) + (-b3)*output(t-3)
```

Now define a new function, middle(t):

```
a0*input(t) + a1*input(t-1) + a2*input(t-2)
 = middle(t)
 =   output(t) + (-b1)*output(t-1)
   + (-b2)*output(t-2) + (-b3)*output(t-3)
```

You can rewrite this as:

```
middle(t) = a0*input(t) + a1*input(t-1) + a2*input(t-2)
middle(t) =   output(t) + (-b1)*output(t-1)
            + (-b2)*output(t-2) + (-b3)*output(t-3)
```

Notice how the transition from input(t) to middle(t) is a FIR filter
and the transition from output(t) to middle(t) is another. So the IIR
filter in fact consists of two FIR filters facing each other. This
gives a simple approach to frequency response calculations (see the
section "IIR frequency response").

```
*** Turning an IIR filter backwards ***

You can solve input(t) from the IIR equation:

output(t) =   a0*input(t) + a1*input(t-1) + a2*input(t-2)
            + b1*output(t-1) + b2*output(t-2) + b3*output(t-3)

a0*input(t) =   output(t) + (-b1)*output(t-1) + (-b2)*output(t-2)
              + (-b3)*output(t-3) + (-a1)*input(t-1) + (-a2)*input(t-2)

input(t) =   (1/a0)*output(t) + (-b1/a0)*output(t-1)
           + (-b2/a0)*output(t-2) + (-b3/a0)*output(t-3)
           + (-a1/a0)*input(t-1) + (-a2/a0)*input(t-2)

Now swap input and output and you have a filter that undoes what the
original did.

But if the frequency response of the original filter was ZERO for some
frequency, the inverted one will amplify that frequency INFINITELY.
This is just logical.

The inverted filter will also have an opposite phase shift, so that if
R(f) is the frequency response of the original filter as a complex
number and r(f) is the frequency response of the inverted filter,
R(f)*r(f)=1 for every f.

*** Getting rid of output(t+n) ***

Say you have somehow found that you need an IIR filter like this:

output(t) =   a0*input(t) + a1*input(t-1) + a2*input(t-2)
            + b1*output(t-1) + b2*output(t-2)
            + B1*output(t+1) + B2*output(t+2)

You need to know both output(t+2) and output(t-2) to be able to
compute output(t).  Doesn't seem very practical.  But you can
shuffle the equation a little:

a0*input(t) + a1*input(t-1) + a2*input(t-2)
 =   (-B2)*output(t+2) + (-B1)*output(t+1)
   + output(t) + (-b1)*output(t-1) + (-b2)*output(t-2)

Now define a new variable u=t+2 and use it instead of t:

a0*input(u-2) + a1*input(u-3) + a2*input(u-4)
 =   (-B2)*output(u) + (-B1)*output(u-1)
   + output(u-2) + (-b1)*output(u-3) + (-b2)*output(u-4)

Then solve output(u):

B2*output(u) =   (-a0)*input(u-2) + (-a1)*input(u-3)
               + (-a2)*input(u-4)
               + (-B1)*output(u-1) + output(u-2)
               + (-b1)*output(u-3) + (-b2)*output(u-4)

output(u) =   (-a0/B2)*input(u-2) + (-a1/B2)*input(u-3)
            + (-a2/B2)*input(u-4)
            + (-B1/B2)*output(u-1) + (1/B2)*output(u-2)
            + (-b1/B2)*output(u-3) + (-b2/B2)*output(u-4)
```

Now you can use the filter.

*** Getting rid of input(t+n) ***

Notice how in the previous example, input(t) became input(u-2).
Had there been input(t+1), it would have become input(u-1) which
can be used in real time filters.

Generally, you can get rid of input(t+n) this way if the equation
also uses output(t+m) where m>=n, because you can define u=t+m which
turns input(t+n) to input(u-(m-n)) which you get in time.

If m<n, this is not possible:

```
output(t) = a0*input(t) + A1*input(t+1)
```

Here m=0 and n=1, so you can't get rid of input(t+1) and keep the
filter mathematically equivalent to the original.

However, you can delay the output by one time unit:

```
delayed_output(t+1) = output(t) = a0*input(t) + A1*input(t+1)
delayed_output(u) = a0*input(u-1) + A1*input(u)
```

Usually, this small delay doesn't matter. But it changes the phase
frequency response of the filter and this DOES matter if you then mix
the filtered signal with the original one or others derived from it;
in that case, you'd better make sure that all of the signals have the
same delay. (Except if you happen to like the extra effect.)

(For example, if you have a filter output(t)=input(t-1), it doesn't do
much as such. But if you mix the "filtered" signal with the original
one, the mixing becomes a filter in itself and you can compute its
frequency response and all.)

If you try to force the original filter through the u=t+m trick by
introducing a dummy 0*output(t+1) term:

```
output(t) = a0*input(t) + A1*input(t+1) + 0*output(t+1)

a0*input(t) + A1*input(t+1) = -0*output(t+1) + output(t)

a0*input(u-1) + A1*input(u) = -0*output(u) + output(u-1)

-0*output(u) = a0*input(u-1) + A1*input(u) - output(u-1)

output(u) = (a0*input(u-1) + A1*input(u) - output(u-1)) / 0
```

you'll just get division by zero.

*** FIR frequency response ***

Treat a sine wave as a rotating phasor e^(i*t*2*pi*f/fs) where:
e = base of natural logarithms (~2.718)
i = imaginary unit (i*i=-1)
t = time (integer, as it's sampled)
pi = ratio of circle perimeter to radius (~3.141)
f = frequency
fs = sampling frequency = samplerate

The real component of this phasor is the regular sine wave.

The neat thing about this is that you can multiply it with various complex numbers to scale the magnitude and shift the phase at the same time.

By defining z=e^(i*2*pi*f/fs), the phasor can be written as z^t. This is the same z that is used in pole-zero calculations (see chapter "IIR filter design using pole-zero method").

Here's the general FIR equation:

$$\text{output}(t) = \sum_{k=0}^{n} m(k)*\text{input}(t-k)$$

where
k = counter running from 0 to n
n = number of taps - 1
m(k) = multiplier (a real number) indexed by k

Now, let's look what the filter does to an infinitely long sine wave with frequency f.

input(t) = sin(t*2*pi*f/fs)

$$\text{output}(t) = \sum_{k=0}^{n} m(k)*\sin((t-k)*2*pi*f/fs)$$

But this sine wave can be replaced with the rotating phasor if we then throw away the imaginary component of the output. m(k) is real so the real and imaginary components can't affect each other.

input(t) = z^t

$$\text{output}(t) = \sum_{k=0}^{n} m(k)*z^{(t-k)}$$

$$\text{output}(t) = \sum_{k=0}^{n} m(k)*z^{t}*z^{(-k)}$$

Here the z^t factor doesn't depend on k, so it can be moved outside the sum:

$$\text{output}(t) = z^{t} * \sum_{k=0}^{n} m(k)*z^{(-k)}$$

$$\text{output}(t) = \text{input}(t) * \sum_{k=0}^{n} m(k)*z^{(-k)}$$

z depends on f (z=e^(i*2*pi*f/fs), remember?) but the value of the sum doesn't depend on t. I'll call it R(f):

```
        n
R(f) = sum m(k)*z^(-k)
       k=0
```

output(t) = R(f) * input(t)

output(t) is a rotating phasor at the same frequency as input(t); it just has a different amplitude and phase as defined by R(f). This means that for an infinitely long sine wave of frequency f, R(f) shows how the filter affects its amplitude and phase.

In other words, R(f) is the frequency response of the filter. It's a complex function. If you don't remember what this means, see section "Complex math with filters" in chapter "What's a filter?" in this file.

*** IIR frequency response ***

When two filters are concatenated so that one filter's output is fed to the other filter's input, the responses are multiplied at each frequency:

Rconcat(f) = Rfirst(f) * Rsecond(f)

A filter that just connects its input to its output and doesn't change the signal at all has a frequency response of 1 at all frequencies:

Rnothing(f) = 1

Now assume that we have a filter with frequency response R(f) and we make another filter with frequency response Rinv(f) that UNDOES everything the first filter did to the signal when they are concatenated.

```
R(f) * Rinv(f) = 1    ==>    Rinv(f) = 1/R(f)
```

So the inverse filter also has an inverse frequency response.

Remember, an IIR filter consists of two FIR filters facing each other (see section "Symmetric form"). This setup can be treated as a normal FIR filter followed by an inverted FIR filter:

```
Riir(f) = Rfir1(f) * Rinvfir2(f)
        = Rfir1(f) * (1/Rfir2(f))
        = Rfir1(f) / Rfir2(f)
```

This means that if you can calculate the frequency responses of the two FIR filters, you can calculate the IIR frequency response by dividing one with the other.

An example. You have this IIR filter.

```
output(t) =   a0*input(t) + a1*input(t-1) + a2*input(t-2)
            + b1*output(t-1) + b2*output(t-2) + b3*output(t-3)
```

It becomes:

```
middle(t) = a0*input(t) + a1*input(t-1) + a2*input(t-2)
middle(t) =   output(t) + (-b1)*output(t-1)
            + (-b2)*output(t-2) + (-b3)*output(t-3)
```

Change the names of functions a little:

```
output1(t) = a0*input1(t) + a1*input1(t-1) + a2*input1(t-2)
output2(t) =   input2(t) + (-b1)*input2(t-1)
             + (-b2)*input2(t-2) + (-b3)*input2(t-3)
```

Compute the frequency response of filter input1->output1 (originally input->middle).

The general formulas:

```
z=e^(i*2*pi*f/fs)

        n
R(f) = sum m(k)*z^(-k)
       k=0
```

In this particular case:

```
n=2
m(0)=a0, m(1)=a1, m(2)=a2

R1(f) = m(0)*z^(-0) + m(1)*z^(-1) + m(2)*z^(-2)
      = a0*1 + a1*z^(-1) + a2*z^(-2)
      = a0 + a1*e^(-i*2*pi*f/fs) + a2*e^(-i*4*pi*f/fs)
```

The input2->output2 (originally output->middle) filter:

```
n=3
m(0)=1, m(1)=-b1, m(2)=-b2, m(3)=-b3

R2(f) = m(0)*z^(-0) + m(1)*z^(-1) + m(2)*z^(-2) + m(3)*z^(-3)
      = 1*1 - b1*z^(-1) - b2*z^(-2) - b3*z^(-3)
      = 1 - b1*e^(-i*2*pi*f/fs) - b2*e^(-i*4*pi*f/fs)
          - b3*e^(-i*6*pi*f/fs)
```

Now the whole IIR:

```
R(f) = R1(f) / R2(f)

            a0 + a1*e^(-i*2*pi*f/fs) + a2*e^(-i*4*pi*f/fs)
= ---------------------------------------------------------------------
  1 - b1*e^(-i*2*pi*f/fs) - b2*e^(-i*4*pi*f/fs) - b3*e^(-i*6*pi*f/fs)
```

To actually calculate the frequency response at some frequency, you'd apply Euler's formula and the usual complex number rules:

```
e^(i*x) = cos x + i*sin x

r * (a + b*i) = r*a + (r*b)*i

(a + b*i) + (c + d*i) = (a+c) + (b+d)*i

(a + b*i) * (c + d*i) = a*c + a*d*i + b*i*c + b*i*d*i
                      = a*c + (a*d+b*c)*i + b*d*(-1)
                      = (a*c-b*d) + (a*d+b*c)*i

(a + b*i) * (a - b*i) = a*a - a*b*i + b*i*a - b*i*b*i
                      = a*a + b*b

a + b*i   (a + b*i) * (c - d*i)   (a*c+b*d) + (b*c-a*d)*i
------- = --------------------- = -----------------------
c + d*i   (c + d*i) * (c - d*i)         c*c + d*d
```

```
------------------------------------- A collection of IIR filters ----

R in the filters means resonance, steepness and narrowness.

*** Fastest and simplest "lowpass" ever! ***

  c = 0..1  (1 = passes all, 0 = passes nothing)

  output(t) = output(t-1) + c*(input(t)-output(t-1))

*** Fast lowpass with resonance v1 ***

  Parameters:
  resofreq = resonation frequency  (must be < SR/4)
  r = 0..1, but not 1

  Init:
  c = 2-2*cos(2*pi*resofreq / samplerate)
  pos = 0
  speed = 0

  Loop:
  speed = speed + (input(t) - pos) * c
  pos = pos + speed
  speed = speed * r
  output(t) = pos

*** Fast lowpass with resonance v2 ***

  Parameters:
  resofreq = resonation frequency  (must be < SR/4)
  amp = magnitude at the resonation frequency

  Init:
  fx = cos(2*pi*resofreq / samplerate)
  c = 2-2*fx
  r = (sqrt(2)*sqrt(-(fx-1)^3)+amp*(fx-1))/(amp*(fx-1))
  pos = 0
  speed = 0

  Loop:
  speed = speed + (input(t) - pos) * c
  pos = pos + speed
  speed = speed * r
  output(t) = pos
```

```
*** Halfband lowpass ***

  b1  =  0.641339     b10 = -0.0227141
  b2  = -3.02936
  b3  =  1.65298      a0a12 = 0.008097
  b4  = -3.4186       a1a11 = 0.048141
  b5  =  1.50021      a2a10 = 0.159244
  b6  = -1.73656      a3a9  = 0.365604
  b7  =  0.554138     a4a8  = 0.636780
  b8  = -0.371742     a5a7  = 0.876793
  b9  =  0.0671787    a6    = 0.973529

  output(t) = a0a12*(input(t  ) + input(t-12))
            + a1a11*(input(t-1) + input(t-11))
            + a2a10*(input(t-2) + input(t-10))
            + a3a9* (input(t-3) + input(t-9 ))
            + a4a8* (input(t-4) + input(t-8 ))
            + a5a7* (input(t-5) + input(t-7 )) + a6*input(t-6)
            + b1*output(t-1) + b2*output(t-2) + b3*output(t-3)
            + b4*output(t-4) + b5*output(t-5) + b6*output(t-6)
            + b7*output(t-7) + b8*output(t-8) + b9*output(t-9)
            + b10*output(t-10)

*** Notch ***

  f = notch center frequency
  r = 0..1, but not 1

  z1x = cos(2*pi*f/samplerate)
  a0a2 = (1-r)^2/(2*(|z1x|+1)) + r
  a1 = -2*z1x*a0a2
  b1 = 2*z1x*r
  b2 = -r*r

  output(t) = a0a2*(input(t)+input(t-2)) + a1*input(t-1)
            + b1*ouput(t-1) + b2*output(t-2)

*** Fast bandpass ***

  freq = passband center frequency
  r = 0..1, but not 1

  fx = cos(2*pi*freq/samplerate)
  a0 = (1-r)*sqrt(r*(r-4*fx^2+2)+1)
  b1 = 2 * fx * r
  b2 = -r^2

  output(t) = a0*input(t)
            + b1*output(t-1) + b2*output(t-2)
```

```
*** Allpass ***

  freq = frequency of the steepest changes in phase
  r = 0..1, but not 1

  fx = cos(2*pi*freq/samplerate)
  fy = sin(2*pi*freq/samplerate)
  z1x = fx/r
  z1y = fy/r
  p1x = fx*r
  p1y = fy*r
  a0 = r^2
  a1 = -2*z1x*a0
  a2 = a0*(z1x^2+z1y^2)
  b1 = 2*p1x
  b2 = -p1y^2-p1x^2

  output(t) = a0*input(t) + a1*input(t-1) + a2*input(t-2)
            + b1*output(t-1) + b2*output(t-2)

*** DC removal ***

  Filters 0 Hz completely away, does not attenuate basses above 5Hz in
  a 44100Hz sampling rate system. This is a realtime routine. For
  example in sample editors, subtracting the average of all the
  samplepoints from the whole sampledata usually does better job.

  Init:
  pos = 0
  speed = 0

  Loop:
  speed = speed + (input(t) - pos) * 0.000004567
  pos = pos + speed
  speed = speed * 0.96
  output(t) = (input(t)-pos)
```

```
------------------------------------ A collection of FIR filters ----

*** 90 degree phase shift ***

  Coefficients from: (1-cos(t*pi))/(t*pi)
  Limit t->0 = 0

  Coefficients at even t values are zero.
  Symmetrical around (0,0).

*** Phase shift of any angle ***

  a = angle

  Coefficients from: (sin(t*pi-a)+sin(a))/(t*pi)
  Coefficients at even t values, not including t=0, are zero.
  Limit t->0 = cos(a)

*** Lowpass ***

  f = cutoff frequency / (samplerate/2)

  Coefficients from: sin(t*f*pi)/(t*pi)
  Limit t->0 = f

  Symmetrical around t=0.

*** Halfband lowpass ***

  Coefficients from: 0.5*sin(t*pi/2)/(t*pi/2)
  Limit t->0 = 0.5

  Coefficients at even t values, not including t=0, are zero.
  Symmetrical around t=0.

*** Highpass ***

  f = cutoff frequency / (samplerate/2)

  Coefficients from: (sin(t*pi)-sin(t*f*pi))/(t*pi)
  Limit t->0 = 1-f

  Symmetrical around t=0.

*** Bandpass ***

  f1 = lower frequency limit / (samplerate/2)
  f2 = higher frequency limit / (samplerate/2)

  Coefficients from: sin(t*f2*pi)/(t*pi) - sin(t*f1*pi)/(t*pi)
  Limit t->0 = f2-f1

  Symmetrical around t=0.
```

```
*** Bandpass magnitude-ramp ***

  x1 = lower limit frequency / (samplerate/2)
  x2 = higher limit frequency / (samplerate/2)
  y1 = magnitude at lower limit frequency
  y2 = magnitude at higher limit frequency

  c = (y1-y2)/(x1-x2)
  d = (x1*y2-x2*y1)/(x1-x2)

  Coefficients from:
  ((d+c*x2)*sin(x2*t*pi)+(-d-c*x1)*sin(x1*t*pi)+
  (c*cos(x2*t*pi)-c*cos(x1*t*pi))/(t*pi))/(t*pi)
  Limit t->0 = (x2-x1)*(y1+y2)/2

  Symmetrical around t=0.

*** Negative frequency removal ***

  Complex coefficients from: sin(t*pi)/(t*pi) + i*(cos(t*pi)-1)/(t*pi)
  Limit t->0 = 1

  Real part symmetrical around t=0.
  Imaginary part symmetrical around (0,0).

*** Combined negative frequency removal and bandpass ***

  f1 = lower frequency limit / (samplerate/2)
  f2 = higher frequency limit / (samplerate/2)

  Complex coefficients from:
  (sin(f2*t*pi)-sin(f1*t*pi))/(t*pi) +
  i*(cos(f2*t*pi)-cos(f1*t*pi))/(t*pi)
  Limit t->0 = f2-f1

  Real part symmetrical around t=0.
  Imaginary part symmetrical around (0,0).
```

-End-