filter shortcuts - derived from zoelzer, chamberlin and tarrabia

the simple one pole
this is typically used for parameter smoothing.
if c is near 1.0, the out slowly moves to the new value (lowpass)
if c is near 0.0, the out follows the new value closely (passthru)

```
out = in * (1 - c) + out * c;
out = in + (out - in)* c;
```

c can be calculated for a given frequency by

```
c = exp(-2.0 * PI * frequency/samplingRate);
```

get a highpass by subtracting in from out

```
out = in + (out - in)* c;
hpout = in - out;
```

coefficient calculation and filtering calculation

most filters have a section of code for setting the frequency, resonance and gain (calculating coefficients) and another section for filtering the audio (using those coefficients and saving internal state/delays)
it's typical for the frequency/resonance to coefficient calculation to be more cpu intensive than the filtering of sound

first and second order filter code derived from udo zölzer's chapters in the DAFX book

an allpass filter keeps the amplitude the same at all frequencies. the first order allpass filter shifts the phase 180 degrees at the nyquist frequency and 0 degrees at 0Hz, with a smooth variation of phase shift between 0Hz and nyquist. adding the original signal to the output of this allpass filter results in a lowpass filter.

one can chain multiple first order sections to create a sharper filter

```
// first order allpass using one coefficient and 2 state variables
float c, in1, out1;

void foAllpass::setFrequency(float f)
{
    tf = tan(pi * f/sampleRate);
    c = (tf - 1.0f)/(tf + 1.0f);
}

float foAllpass::process(float in, float c)
{
    out = c * in + in1 - c * out1;
    in1 = in;
    out1 = out;
    return(out);
}

// lowpass requires a slight change to the 4th line
float foLowpass::process(float in, float c)
{
    out = c * in + in1 - c * out1;
    in1 = in;
    out1 = out;
    return((in1 + out1) * 0.5f);
}

// highpass requires inverting the phaseshifted out
float foHighpass::process(float in, float c)
{
    out = c * in + in1 - c * out1;
    in1 = in;
    out1 = out;
    return((in1 - out1) * 0.5f);
}
```

the second order allpass filter shifts the phase 360 degrees at the nyquist frequency and 0 degrees at 0 Hz, so it is 180 degrees (inverted) somewhere in between. this 180 degree point is the center frequency of the filter. you can create a band reject filter by adding the input and allpass output and a band pass filter by subtracting the allpass output from the input. in zolzer's implementation, filter bandwidth can be set as well as frequency.

```
float c, d, in1, in2, out1, out2;

void soAllpass::setFrequency(float f, float bw)
{
    tf = tan(pi * bw/sampleRate);
    c = (tf -1.0f)/(tf + 1.0f);
    d = -cos(twopi * f/sampleRate);
}

float soAllpass::process(float in, float c)
{
    out = -c * in + d * (1.0f - c) * in1 + in2 - d * (1.0f - c) * out1 + c * out2;
    out2 = out1;
    out1 = out;
    in2 = in1;
    in1 = in;
    return(out);
}
// like the first order example - bandpass and reject derived from allpass
float soBandpass::process(float in, float c)
{
    out = -c * in + d * (1.0f - c) * in1 + in2 - d * (1.0f - c) * out1 + c * out2;
    out2 = out1;
    out1 = out;
    in2 = in1;
    in1 = in;
    return(in - out);
}
float soBandreject::process(float in, float c)
{
    out = -c * in + d * (1.0f - c) * in1 + in2 - d * (1.0f - c) * out1 + c * out2;
    out2 = out1;
    out1 = out;
    in2 = in1;
    in1 = in;
    return(in + out);
}
```

the most used digital filter - the biquad filter

the second order filters are using 4 state variables and 2 coefficients. we can now look at a similar and slightly more complex filter, the biquad filter. there are many examples of how to create biquad coefficients: the robert bristow-johnson article: http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt, some filters from zoelzer: http://www.musicdsp.org/archive.php?classid=3#37, and the following butterwort lowpass from patrice tarrabia: http://www.musicdsp.org/archive.php?classid=3#38

note that the the intermediate variables C and resonance are not dependent on each other. these could easily be put into tables.

```
BQLowpass::BQLowpass()
{
   A1 = A2 = A3 = 0.0;   // coefficients for inputs
   B1 = B2 = 0.0;    // coefficients for outputs
   X1 = X2 = 0.0;   // previous inputs
   Y1 = Y2 = 0.0;   // previous outputs
   Pi = 4.0 * atan(1.0);   // one way to calculate PI
}

BQLowpass::SetFilter(float frequency, float resonance)
{
   float C, f0;

   if(resonance > sqrt(2.0)) resonance = sqrt(2.0); // limit the resonance
   if(resonance < 0.1) resonance = 0.1;
   if(frequency > sampleRate * 0.5) frequency = sampleRate * 0.5; // limit freq

   f0 = frequency/sampleRate;
   if(f0 < 0.1)
         C = 1.0 / (f0 * Pi);  // quick approximation for low frequencies
   else
         C = tan((0.5 - f0) * Pi);
   A3 = A1 = 1.0 / ( 1.0 + resonance * C + C * C);
   A2 = 2.0 * A1;
   B1 = 2.0 * ( 1.0 - C * C) * A1;
   B2 = ( 1.0 - resonance * C + C * C) * A1;
}
```

```
BQLowpass::FilterBlock(float *input, float *output, long samplePerBlock)
{
   long sample;
   for(sample = 0; sample < samplesPerBlock; sample++)
   {
      *(output + sample) = A1 * *(input + sample) + A2 * X1 + A3 * X2
        - B1 * Y1 - B2 * Y2;
      X2 = X1;
      X1 = *(input + sample);
      Y2 = Y1;
      Y1 = *(output + sample);
   }
}
```

the state variable filter

a state variable filter has two integrators feeding back into each other, and a network that derives all 4 filter types simultaneously this is derived from the BASIC code in hal chamberlain's musical applications of microprocessors. damping and freqCoef are limited to keep the filter more stable.

```
SVFilter::ProcessBlock(float *input, float *output, long samplesPerBlock,
                                    float frequency, float resonance, float drive)
{
   long sample;
   float in, out;
   float freqCoef, damping;
   freqCoef = 2.0*sin(pi * (frequency/sampleRate)));
   if(freqCoef > 0.5)
      freqCoef = 0.5;
   damping = 2.0*(1.0 - pow(resonance, 0.25));
   if(damping > 2.0)
      damping = 2.0;
   if(damping > (2.0/freqCoef - (freqCoef * 0.5)))
      damping = 2.0/freqCoef - (freqCoef * 0.5);
   for (sample=0; i<samplesPerBlock; sample++)
   {
      in    = *(input+i);
      notchOut = in - (damp * bandOut);
      lowOut   = lowOut + (freqCoef * bandOut);
      highOut  = notchOut - lowOut;
      bandOut  = bandOut + (freqCoef * highOut);
      output[i] = lowOut;
   }
}
```