2 PHASORS
3 ANGULAR FREQUENCY

4 DELAYS
5 INTERPOLATION
DECAY

6 MASS-SPRINGS
7 FAST SINE OSCILLATOR
PHASE CORRECTION (DAMPING)
UNISON VOICING
HARD SYNC
REVERSING SOFT SYNC

8 ALLPASS FILTERS

9 PANNING
DECIBELS
SOFTLIMITING
MID-SIDE PROCESS

e 2.71828182845904523536028747135266249775724709369995
1 + 1/1 + 1/(1*2) + 1/(1*2*3) + 1/(1*2*3*4) . . .

for 0 to pi/2:
sin(x) x - (x^3 / 3!) + (x^5 / 5!) - (x^7 / 7!) + (x^9 / 9!) - ... (Taylor series)
cos(x) 1 - (x^2 / 2!) + (x^4 / 4!) - (x^6 / 6!) + (x^8 / 8!) - ...
tan(x) x + (x^3/3) + (2*x^5/15) + (17*x^7/315) + (62*x^9/2835) ....
1, 3, 15, 315, 2835, 155925, 6081075, 638512875, 10854718875, 1856156927625, 194896477400625,
2900518163668125, 3698160658676859375..

in Horner form:
fastsin(x) x* (x2 * (x2 * (x2 * (x2 * (1.0/362880.0)
- (1.0/5040.0))
+ (1.0/120.0))
- (1.0/6.0))
+ 1.0);

or: o= x*x;
x * (o * (o * (o * (o * .0000027557319 - .0001984126984) + .0083333333333) - .1666666666666) + 1);

pi / 2 1.57079632679489661923132169163975
pi 3.14159265358979323846264338327950
tau 6.28318530717958647692528676655901

plastic number 1.32471795724474602596090885447809 n^3 - 1 = n
phi 1.61803398874989484820458683436564 (1 + sqrt(5)) / 2
sqrt(10) 3.16227766016837933199889354443271

two useful gaussian transforms:
y = pow(2, -2 * x * x); x = .5, y = .7071
y = pow(2, -4 * x * x); x = .5, y = .5

PHASORS

The phasor, for my purposes, is the variable assigned to the phase of an oscillator. This is an elementary concept anyone should be able to implement without reference. The wavecycle itself may run from 0 to 1, from 0 to 2*pi, -pi to +pi, 0 to the length of a lookup array, depending on how the oscillator is efficiently rendered. The basic form is:

```
wavelength = 1;                                          //    initialisation
increment = wavelength * frequency_in_Hertz / samplerate;

phase += increment;                                      //    loop
if (phase >= wavelength) phase -= wavelength;
```

INT phasors are commonly used for efficiency. The use of unsigned INTs makes the second loop argument unnecessary if the wavelength equals the range of the variable. Otherwise, the IF statement often proves more efficient for wrapping than modulus, ANDing, or other techniques in my experience.

This simple subject is addressed to include the computation of phasors using the SynthEdit SDK. In Synthedit, pitch is ergonomically scaled with unit octaves, 5 "volts" equal to 440Hz and yielding the range of 0 to 10 as 11.25Hz to 14080Hz.

Development in SynthEdit is complicated by the 0 to 10 "volt" range in the modular environment converting to 0 to 1 when put to the pin of a SEM module, eg. 7 "volts" in the modular environment will be passed to the SEM code as 0.7. The scaling is conveniently applied in reverse upon exiting the module.

If we wish to create a phasor for SynthEdit which wraps from 0 to 1, we could use:

```
increment = 440.f * pow(2.f, pitch * 10.f - 5.f) / samplerate;    //    frequency in Hertz / samplerate
```

This statement can be economised by declaring a samplerate variable for computing oscillators:

```
oscrate = samplerate / 440.f;
increment = pow(2.f, pitch * 10.f - 5.f) / oscrate;
```

It can be economised further:

```
oscrate = 32.f * samplerate / 440.f;
```
OR:
```
oscrate = samplerate / 13.75f;
increment = pow(2.f, pitch * 10.f) / oscrate;
```

It can be economised even further since division is generally slower than multiplication. Instead of dividing something by n, the same result can be produced by multiplying by the inverse of n. If oscrate is calculated as an inverse of the above:

```
oscrate = 13.75f / samplerate;
increment = pow(2.f, pitch * 10.f) * oscrate;
```

ANGULAR FREQUENCY

Denoted by lower case omega, which looks like a 'w', angular frequency is a DSP fundamental.

```
w = 2 * pi * frequency_in_Hertz / samplerate;
```

We can again economise this for the SynthEdit SDK by declaring a variable which factors in the 2 * pi:

```
wrate = samplerate / (13.75 * 2 * pi);
w = pow(2, pitch * 10) / wrate;
```

OR:

```
wrate = (13.75 * 2 * pi) / samplerate;
w = pow(2, pitch * 10)  * wrate;
```

It can be necessary to limit angular frequency to nyquist depending on the application.

What can we do with angular frequency? A 1-pole lowpass filter can be produced as follows:

```
output = w * input + (1 - w) * y1;
y1 = output;
```

DELAYS

A delay is easily implemented as a circular buffer. A write pointer wraps through an array advancing once per sample, and a read pointer chases it, at the delay length in samples, making the longest possible delay one less sample than the length of the array.

The fastest code I know of for accomplishing this is to make the array 2^16 samples long, and use unsigned short INTs for the pointers (or 2^8 samples, and use unsigned CHARs, if your delay is extremely short). This avoids having to wrap the variables. If you use interpolation, you will have at least three variables that need to be wrapped.

```
float b[65536];                         //    initialisation
unsigned short int p, r;                //    pointer and read variables
int len;

if (len > 65535) len = 65535;           //    check at delay length calculation
else if (len < 1) len = 1;              //    otherwise, a length of 0 will output the input from 65536 samples ago

r = p - len;                            //    loop
output = b[r];
b[p] = input + output * feedback;
p++;
```

Uninterpolated delays can be used wherever the pitch of the delay loop isn't critical because the signal is lossless, samples retain their original values. Linear interpolation is simple to implement, though improved interpolation formulas are preferred for fidelity. The basis of linear interpolation is as follows:

```
        decimal = 0.4;                  //    0.4 between x0 and x1
        output = decimal * x0 + (1 - decimal) * x1;
OR:     output = x0 + decimal * (x1 - x0);
```

Here is the delay algorithm again using linear interpolation, and using LONG INTs instead of short, which will need to be wrapped.

```
float b[65536];                         //    initialisation
int p, r0, r1;
float len, d;                           //    d = decimal part of length
int leni;                               //    integer value of length

if (len > 65534) len = 65534;           //    must be 1 unit shorter due to additional read variable
else if (len < 1) len = 1;
leni = (int) len;                       //    float to INT conversions are cpu intensive. Use the typecast :)
d = len - leni;

r0 = p - leni;                          //    loop
r1 = r0 - 1;
if (r0 < 0) r0 += 65536;
if (r1 < 0) r1 += 65536;
output = b[r0] + d * (b[r1] - b[r0]);
b[p] = input + output * feedback;
p++;
if (p > 65535) p = 0;
```

As we've seen, the three IF statements in the loop can be removed by using unsigned INTs if the delay length fits your application.

INTERPOLATION

The Karplus-Strong algorithm models a plucked string with a delay line set to the wavelength of the pitch. Using linear interpolation will elicit that high frequencies are quickly attenuated by this interpolation technique. If you are modeling piano strings or other systems where the retention of high frequencies is critical, it is necessary to use another interpolation method. The two methods described here are similarly effective for this task.

I have not yet investigated higher order interpolation and cannot accurately attribute the source of these methods or that they are accurately named. An app available on my webpage graphically illustrates the differences between them.

These interpolations require four values and are conducted at point d between the central pair b[i] and b[i+1]. Note that these arrays are arranged "forwards" whereas reading a delay array is performed "backwards".

```
a = (b[i+2] - b[i+1]) - (b[i-1] - b[i]);                    //    bicubic interpolation
b = (b[i-1] - b[i]) - a;
c = b[i+1] - b[i-1];
d = decimal value
output = ((a * d + b) * d + c) * d + b[i];

a = (3 * (b[i] - b[i+1]) - b[i-1] + b[i+2]) * 0.5;          //    hermite interpolation
b = b[i+1] + b[i+1] + b[i-1] - (5 * b[i] + b[i+2]) * 0.5;
c = (b[i+1] - b[i-1]) * 0.5;
d = decimal value
output = ((a * d + b) * d + c) * d + b[i];
```

This is a nice arrangement if you want to discern several points between two values, because you only have to perform the last argument.

DECAY

The gain of a recursive system at recursion n is derivable using the pow function:

```
gain_at_n_recursions = pow(gain, n);
```

Eg. a variable multiplied by 0.5 on every sample is obviously 0.0625 of its original value on the 4th iteration:

```
0.0625 = pow(0.5, 4);
```

If we intend to have a specific gain at a specific number of recursions, we can calculate a gain coefficient:

```
sought coefficient = pow(intended_gain_at_n_recursions, inverse_of_n);
```

```
eg:     0.5 = pow(0.0625, 1/4);
```

Given any two of these variables we can derive the third using the identities:

```
x = pow(b, y);
b = pow(x, 1 / y);
y = log(x) / log(b);
```

```
eg.     4 = log(0.0625) / log(0.5);
```

One can then calculate the RT60 time for a reverb or Karplus-Strong model from the delay length of the algorithm.

MASS-SPRINGS

My mass-spring algorithm seems to fulfill all of the technical qualifications (sinusoidal contour, exponential decay) while exhibiting a spartan form. It also makes an efficient sine oscillator, if the pitch isn't modulated while running.

This algorithm is an expression of Hooke's law of elasticity, F = -kx, where x is the amount of displacement, F is the restoring force, and k is known as the spring constant. My digital implementation is as such:

```
k = -w * w;                         //    spring constant is negative square of angular frequency
if (k < -4) k = -4;                 //    stability seems about here, I've been using 3.9
mp = 1;                             //    initial position
mv = 0;                             //    initial velocity

mv = k * mp + gain * mv;            //    loop
mp += mv;
output = mp;
```

This is a reactive system that does not need to be initialised if coupled with other signals. I believe it can be termed a 2nd order IIR filter. If initialisation at mp = 0 is desired ('sine' instead of 'cosine'), unity gain can be achieved by scaling velocity by angular frequency:

```
mp = 0;
mv = w;
```

Arbitrary phase can accordingly be produced using a sine and cosine function. It has been observed that arbitrary phase varies slightly from unity gain. A compatriot's analysis was that that this error is reduced by oversampling, so apparently this is the correct derivation:

```
mp = cos(phase);
mv = sin(phase) * w;
```

The decay time does not adhere to the formula provided in the previous section. I have yet to discern anything more concrete than the following approximations, I'm confident a fairly simple derivation would be obviated with some exploration:

```
gain = pow(0.01, 1/samples);        //    ~0.1 amplitude, about -20dB
gain = pow(0.01, 4/samples);        //    ~0.01 amplitude, about -40dB
gain = pow(0.01, 6.91/samples);     //    ~0.001 amplitude, about -60dB
```

Distance from another element can be implemented using an offset coefficient:

```
mv = k * (mp - offset) + gain * mv;
mp += mv;
if (mp < 0) mp = 0;
```

Variant with a collision normal:

```
if (mp < boundary) {
      mp = boundary;
      mv = collision_normal * k * mp - gain * collision_normal * collision_normal * mv;
}
else mv = k * mp + gain * mv;
mp += mv;
```

FAST SINE OSCILLATOR

Widely referenced, I haven't used it myself. See also mass-spring algorithm above.

```
a = 2 * sin(pi * frequency_in_Hertz / samplerate);          //    initialisation
s0 = 1;
s1 = 0;

s0 -= a * s1;                                               //    loop
s1 += a * s0;
sine_output = s0;
cosine_output = s1;
```

I have used this.. (thanks here to mystran)

```
cosw = cos(w);                                              //    initialisaton
sinw = sin(w);
c = 1;
s = 0;

float temp = cosw * c - sinw * s;                           //    loop
sine_output = s = sinw * c + cosw * s;
cosine_output = c = temp;

float leg = c * c + s * s;                                  //    once every block
if (leg) {leg = sqrt(leg); c /= leg;     s /= leg;}
```

PHASE CORRECTION (DAMPING)

The length of a tuned delay can be adjusted for the phase offset of a nested damping filter (or 6dB/octave lowpass as described in Angular Frequency section). The coefficients for the filter and oscillator frequencies of course only need to reflect the correct ratio and can be angular frequency, Hertz, et c.

```
length *= atan(filter / pitch) / halfpi;
```

UNISON VOICING

Correct gain for unison voicing is:

```
1 / sqrt(number_of_voices);
```

HARD SYNC

Correct adjustment of phase during hard sync is:

```
phase2 = phase1 * increment2 / increment1;
```

REVERSING SOFT SYNC

Correct adjustment of phase during reversing soft sync is:

XXXXXXXXXXXXXXXXha!ha!ha!XXXXXXXXXXXXXXXX

Also remember to apply the sign of increment2 when recalculating it at pitch modulation.

ALLPASS FILTERS

The allpass filters here have a greater effect on the phase of lower frequencies and can be used to model dispersion in rigid materials (the biquad allpass in the rbj cookbook alternatively sets the greatest delay at the filter frequency). Adding this allpass filter to a Karplus-Strong algorithm will 'stretch' the partials away from the fundamental. In noncritical implementations, a filter coefficient ranging from 0 to 1 (or a fraction below 1) can suffice, making this form viable:

```
y1 = x1 + gain * (y1 - in);
x1 = in;
out = y1;
```

If you wish to combine the allpass with a delay line for dispersion as part of a reverb unit, then the conventional form must be used:

```
temp = y1;
y1 = in + y1 * gain;
out = temp - y1 * gain;
```

Note that both of these forms have a one sample delay on the entire bandwidth.

The gain coefficient can be computed to correspond to the frequency that is phase shifted by 90°:

```
wh = pi * frequency_in_Hertz / samplerate;
gain = tan(wh);
gain = (1 - gain) / (1 + gain);
if (gain > 1) gain = 1;
else if (gain < 0) gain = 0;
```

This can be approximated by:

```
gain = 1 - 4 * (frequency_in_Hertz / samplerate);
```

When using dispersion in a waveguide, it is necessary to adjust the length of the delay to accomodate for the phase response.

```
len = length_of_delay_in_samples;
order = integer_number_of_allpasses;      //      enumerated from 1, not 0..

af = (2 * pi) / len;
ac = cos(af);
aa = -gain * ac + 1;
ac -= gain;
ad = sin(af);
ab = -gain * ad;
theta = atan2((ab * ac - aa * ad),(aa * ac + ab * ad));
len += order * len * theta * (1 / pi) * 0.5;
```

PANNING

Using cosine and sine is often adequate for panning. Here's how to calculate panning in decibels:

```
pan = ranged from 0 to 1
pan_l = 20 * log10(cos(pih * pan));
pan_r = 20 * log10(cos(pih * (1 - pan)));
```

In use:

```
pan_l = pow(10, log10(cos(pih * pan)));
pan_r = pow(10, log10(cos(pih * (1 - pan)));
```

DECIBELS

The definition of decibels seems to be derived from the zodiac. I believe the most common implementation in dsp is pow(10, decibels/20) which produces the following gain coefficients:

```
0     = 1.0           = pow(10, 0/20);
-3    = 0.70794578    = pow(10, -3/20);
-6    = 0.50118723    = pow(10, -6/20);
-10   = 0.31622776    = pow(10, -10/20);      //    the reciprocal is worth calculating
-20   = 0.1           = pow(10, -20/20);      //    Perry Cook rssfia uses 20dB = .1
-40   = 0.01          = pow(10, -40/20);      //    so someone agrees with this definition
-60   = 0.001         = pow(10, -60/20);
```

Some people seem to use pow(10, n/10) if the above doesn't correlate with other information.

SOFTLIMITING

This method does not affect the signal until its amplitude passes 1, and stops it from ever passing 2. With minor adjustment, the limit and crush range can be customised.

```
if (o > 1.f) o = 1.f + (o - 1.f) * (1.f / o);
else if (o < -1.f) o = -1.f - (o + 1.f) * (1.f / o);
```

Softer static form, processes at 1, limits at 4:

```
if (o > 1.f) o = (1.f - 4.f / (o + 3.f)) * 4.f + 1.f;
else if (o < -1.f) o = (1.f + 4.f / (o - 3.f)) * -4.f - 1.f;
```

..or not quite as soft..

```
if (o > 1.f) o = (1.f - 2.f / (o + 1.f)) * 2.f + 1.f;
else if (o < -1.f) o = (1.f + 2.f / (o - 1.f)) * -2.f - 1.f;
```

MID-SIDE PROCESS

```
Mid = L + R                    //    encode L-R > M-S
Side = L - R

Left = (M + S) * .5            //    decode M-S > L-R
Right = (M - S) * .5
```