CIRCULAR WAVE updated december 2015

This pdf shows how to use a rational parametrization of the circle in combination with the algebra of complex numbers to program what I call a circular wave - which corresponds to a sine wave. It relies almost entirely on concepts developed by mathematician NJ Wildberger.
But if there is something specific you want to ask then please write me at:
vilbjorg at antidelusionmechanism dot org. And I hope I can answer.

Rational parametrization of the unit-circle

$X = \dfrac{1-t^2}{1+t^2}$ $\quad Y = \dfrac{2t}{1+t^2}$ (from Pythagorean triples, idea via professor NJ Wildberger)

PointXY = e(t) = ((1 – t*t)/(1 + t*t) , 2*t/(1 + t*t))

notice that the values (x,y) calculated from a given t actually lie on the circle and are not an approximation like (cosx,sinx)

I will start with the problematic part: Programming a 'traditional' sine wave one constantly uses the perimeter of the unit circle : 2*pi. There is a linear relation between frequency and the perimeter how ever one can think of that really being the case.
When calculating the parameter t from a given frequency there is no linear relation and no algebraic way (not using transcendentals) to do this.
To calculate angular-speed (theta / sample) from frequency : theta = 2pi*frequency/sampleRate
calculating the parameter t from theta can be done in a few ways, I have usually done it like this:

p = cos(theta)
t = sqrt ((1 – p) / (1 + p)) for theta: 0 <= theta < pi, for negative values of theta t should be negative as well.

for very small values of theta this could be more precise:
p = sin(theta)
des = sqrt(4 - 4*p*p)
t = (2 - des)/ (2*p) if abs(theta) on the on the other hand is greater than pi/2 : (2 **+** des)/ (2*p)

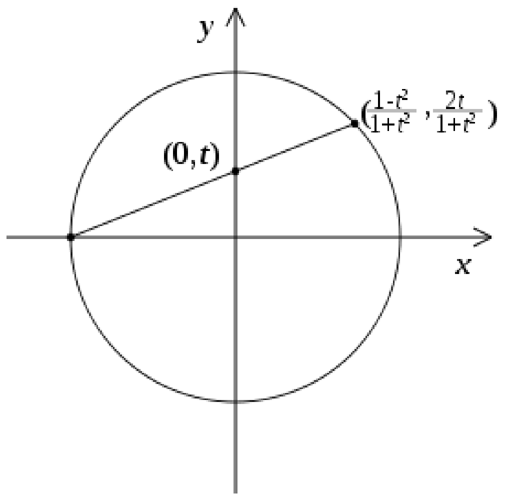
But once you have a certain rotation parameter - corresponding to an angular speed - you can generate the circular wave with only multiplications and divisions. Beneath this (constant) rotation parameter is called r.
One can make a complex number from the parameter.

$$Zr = \frac{1-r^2}{1+r^2} + i \, \frac{2r}{1+r^2}$$

multiplication with a complex number of unit length will create a rotation
with algebraic manipulations it can be shown that
calculating new parameter, t , from previous , t0 , and the rotation r :

$$Zt = Zr * Zt0 \;;\quad t = \frac{r+t0}{1-r*t0}$$

(refer to Wildberger's youtube video “WT15 – complex numbers and rotations”) special case r*t0 = 1 which means t = +/- infinity , [x,y] =[-1,0] (here t wraps from infinity to -infinity, therefore one can have a caution in the code, to avoid 0 in denominator)



CIRCULAR WAVE: in some pseudo code

```
 rotation(r, t)
{
        if( r*t ==1)    // caution for 0 in denominator
        {
           t =  -10000; // the greater the negative number, the smaller the irregularity in the wave,,
                        // and if your rotation parameter was negative this must be positive to
                        //continue  around the unit circle
           x = -1;              // if r*t is exactly 1
           y = 0;
         } else {
            t = (t + r)/(1- t*r) ; // it is here the 0 can occur
            x = (1 – t*t)/(1+ t*t); // some multiplications can be done just once
            y = 2*t/ (1+ t*t);
         }
         return t, x, y
}
```

In praxis I have been running this extensively without the caution and without ever running into exactly 0 in the denominator, but yes, just in case... If you for instance do choose a rotation parameter r of exactly 1 (angle pi/2) and start from 0 then you will get 0 in the denominator..

CONSIDERATION: possibility for type overflow extremely close to the point [-1, 0] , where t wraps from infinity to minus infinity, I have never encountered it, but in theory it is definitely possible.

And with homogeneous coordinates :

PointXYZ on the unit circle:
PointXYZ = e(t : u) = [u*u − t*t : 2*u*t : u*u + t*t]
    (put u=1 and you see it is the same as above e(t))

with homogeneous coordinates reaching the exact point [-1 : 0 : 1] is no longer a problem:

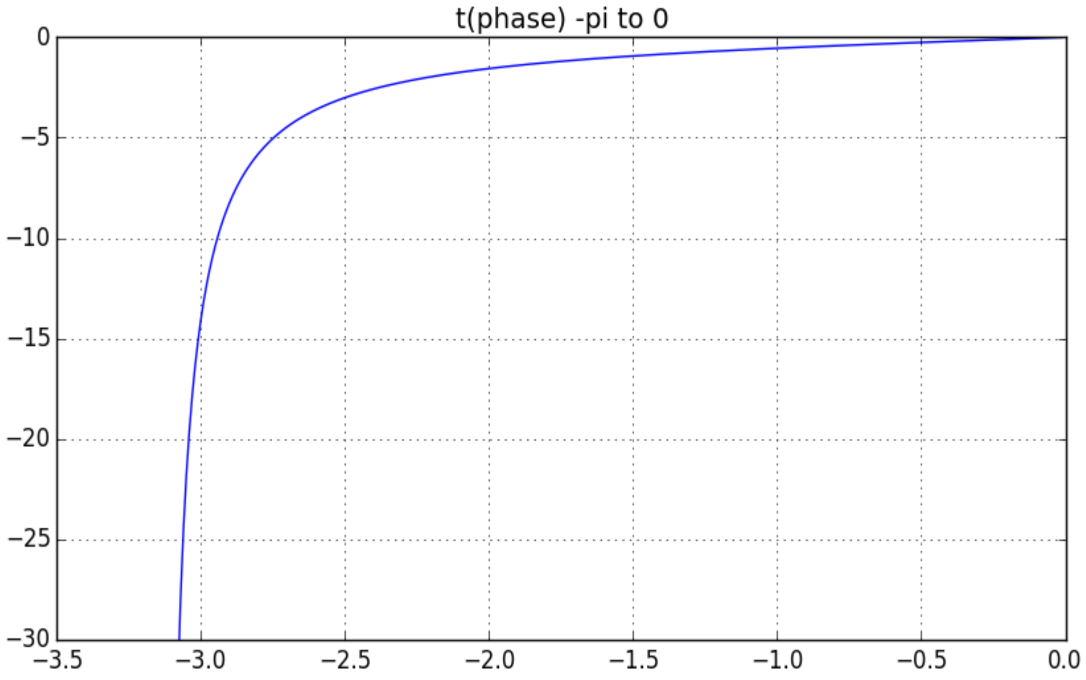
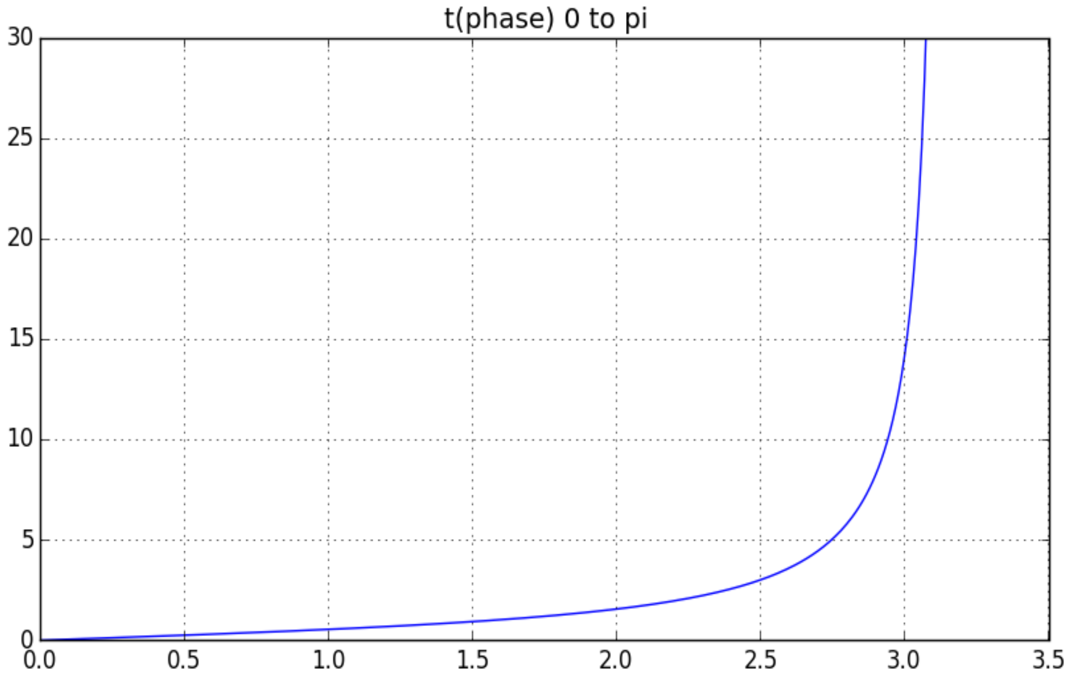
if 3 complex numbers (on the unit circle) are respectively e(t : u) , e(r : s) and e(v : w)
and   Ctu * Crs = Cvw
then there is the following relation of the parameters (can be shown with algebra) :
    v = s*t + r*u ;   w = s*u − r*t

CIRCULAR WAVE homogeneous coordinates pseudocode:
this can in principle be done with all integers , but they grow very large by iteration / repeated rotation, so for programming waveforms I have not yet found a good use for this. The idea to generate accurate points on the unit-circle with integers is thrilling though.

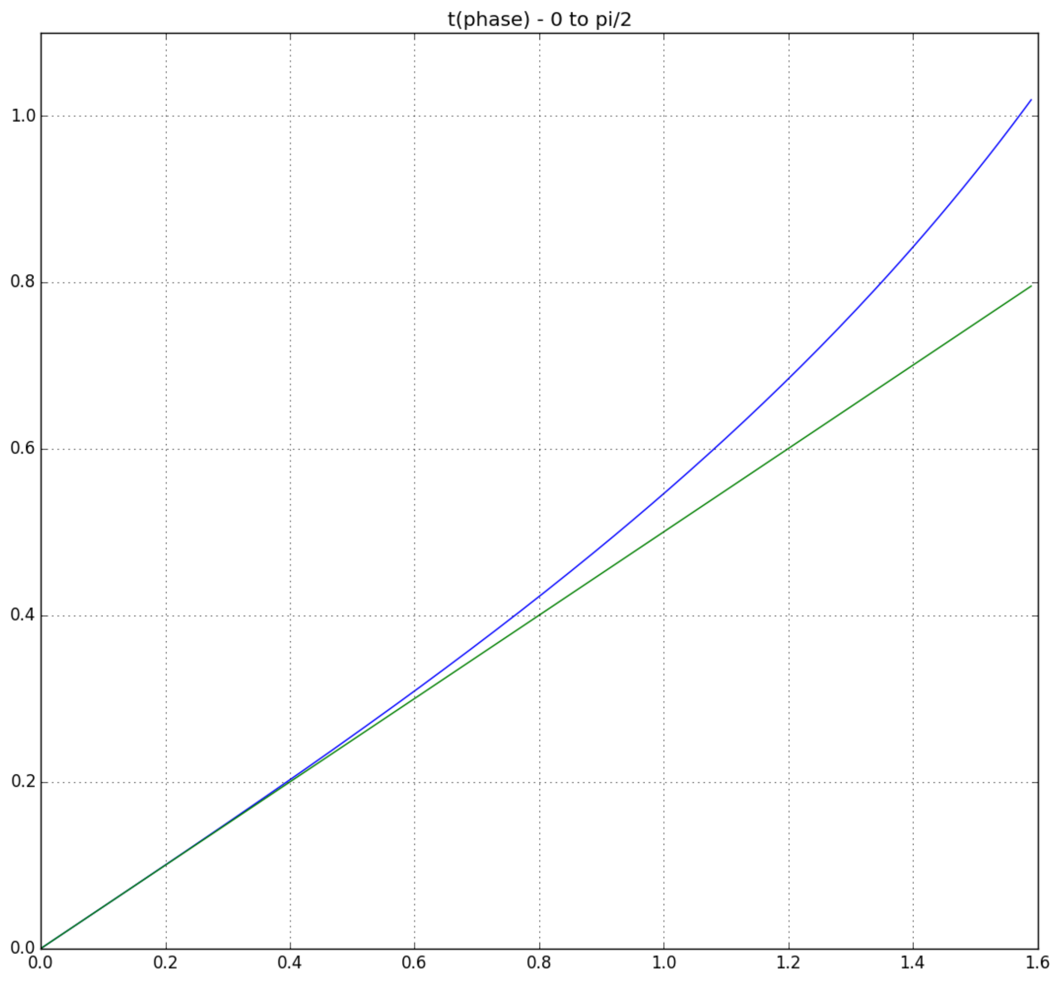
```
rotation( r, s, t, u)
{
        //first calculating new parameters
        long long long long long  int v;
         v = s*t  + r*u;
         u = s*u − r*t;
         t = v;
        //then calculating  the new point on the unit circle
        x = u*u − t*t;
        y = 2*u*t;
        z = u*u + t*t;

        return t, u, x, y, z
}
```

If you want to plot this in an X,Y plane: X = x/z and Y = y/z

Here comes some simple graphical research followed by short discussion.
The expressions 'phase' and 'angular speed' are used in between each other and that might be a bit confusing What one can keep in mind is that to create the rational rotation there are 2 parameters in use. One is constant and determines the 'size' of the rotation (called r in the code), the other is the parameter which is rotated (called t in the code). In a 'traditional' sine wave there is a linear relation between phase and frequency, To work with the rational rotation is to a large part to explore the relation between the parameter (called t or r) and the phase and thereby frequency..
   In the following I consider the angular-speed or phase (increment) can be calculated from frequency (fq) depending on chosen sample_rate. Phase = fq*2*pi/sample_rate. To go the other way from phase to frequency it is naturally: fq = phase*sample_rate/(2*pi).

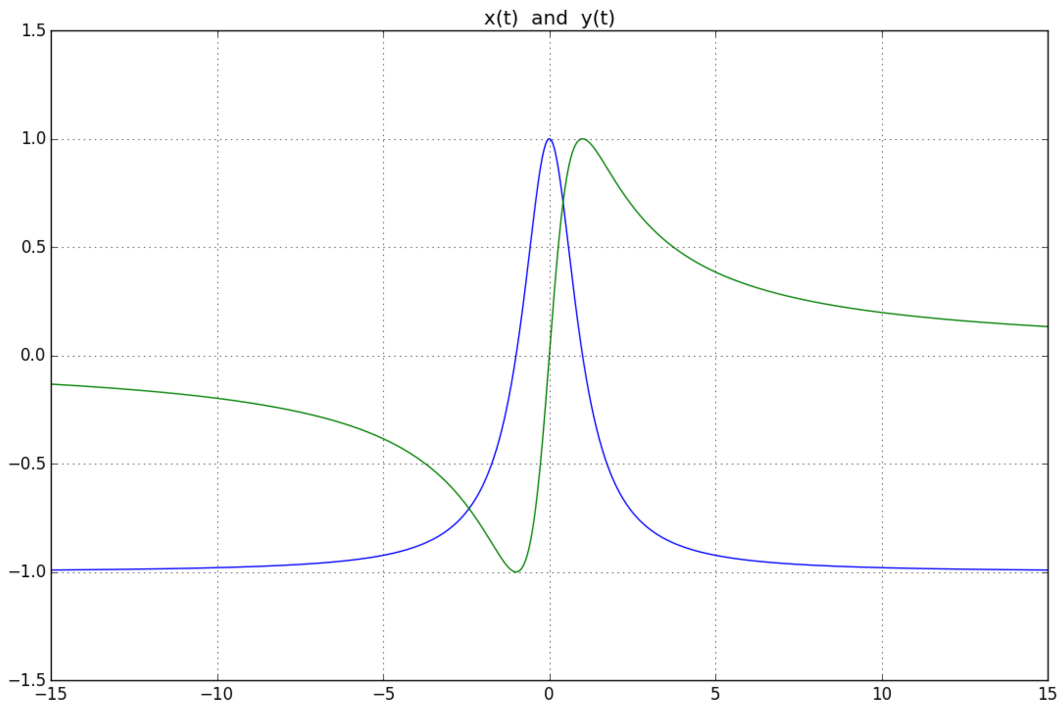

t(phase) -pi to 0
0
−5
−10
−15
−20
−25
−30
−3.5
−3.0
−2.5
−2.0
−1.5
−1.0
−0.5
0.0

t(phase) 0 to pi
30
25
20
15
10
5
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

Here is a plot of t as a function of phase concentrated between 0 and pi/2. The green line is linear for comparison. That t(phase) looks linear close to 0 , can be compared with that sin(theta) is close to theta for small values of theta.



The frequency distribution as a function of t in the final output becomes like the inverse function of the t(angular-speed) Starts out linear and squeezes more and more horizontal as t goes past 1.
-- t = 1 (exactly 1 ! !) corresponds to an angular speed of pi/2 which is 1/4 of your sample rate, by 44.1 kHz it is 10025 hz, it is around 1 octave beyond the piano. Nyquist rate is the point where t wraps from +infinity to -infinity.
To run a circular wave at constant frequency the rotation parameter needs only to be computed once, from there on there are only multiplications and additions.

I have experimented with approximating t(phase) with a polynomial (2nd degree), but have not yet found a result I would find acceptable and useful. Still I have included a python-script and some printouts at the end of this pdf, in case it could be of use to anyone. More interestingly looking at polynomial approximation raises the question if some polynomials perhaps could, not approximate, but rather define new tunings. This exploration I will still have to do.



x = (1 − t^2)/(1 + t^2) blue --- y =2*t/(1 + t^2) green

Discussion:

A sine wave with constant frequency is in the most sound programming environments most efficiently done by wave table lookup, but at the moment of writing in the environment Chuck it is actually done with a sin function. I believe c++ calls fsin in microcode and the result is interpolated. This is actually surprisingly efficient. Included in this pdf is chuck code for the circular wave and code for an external object for the environment Pure Data. I have done some estimates of the efficiency compared to doing something similar using a sine function. The 2 do not seem much different, though this could be tested thoroughly. . . And the precision is so high that it is not possible to hear a difference. But I could imagine applications where the precision of the rational parametrization could be interesting.
I

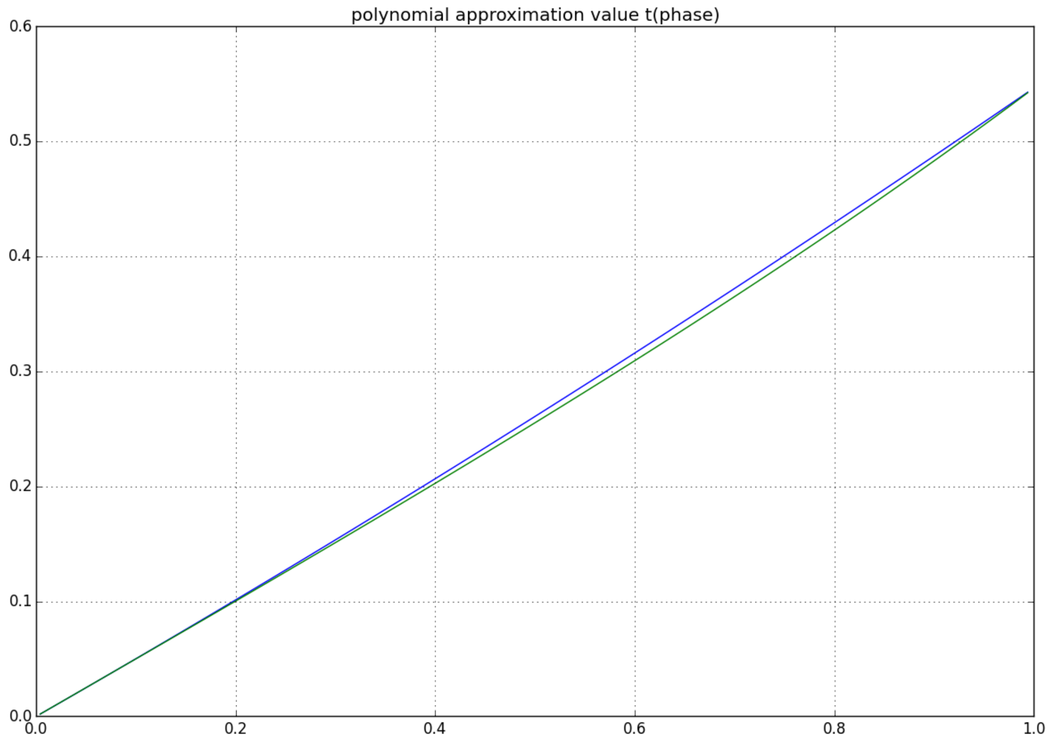
So I still think that the rational rotation is very elegant and also very precise indeed and it might have a future when implemented in some ways on lower levels.

// code for chuck

```
Impulse im1 => Gain g1 => dac.left;
Impulse im2 => Gain g2 => dac.right;
0.3 => g1.gain => g2.gain;

class CircularWave
{
    0.0 => float t;
    0.0 => float t2;
    0.0 => float r;

    0.0 => float x;
    0.0 => float y;




    fun float f(float fq)
    {
        Math.cos(fq*pi/22050) => r; // going from frequency to angular speed..
        //there is probably an object to get the sample rate

        Math.sqrt((1-r)/(1+r)) => r; //..to corresponding rotation parameter r

        while (true)
        {
            if ( (r*t) == 1)
            {
                -1.0 => x;
                0.0 =>  y;
                -1000000 => t;

            }
            else{
                (t+r)/(1- t*r) => t;
                t*t => t2;
                (1-t2)/(1+t2) => x;
                2*t/(1+t2) => y;

            }

            x => im1.next;
            y => im2.next;


            1::samp => now;
        }
    }
}

CircularWave cw;


cw.f(431.0);
//////////////////////////////////////////////////////////////////
```

Here follows the attempt at approximation done in python. As mentioned earlier it is at this point not precise enough to be useful.

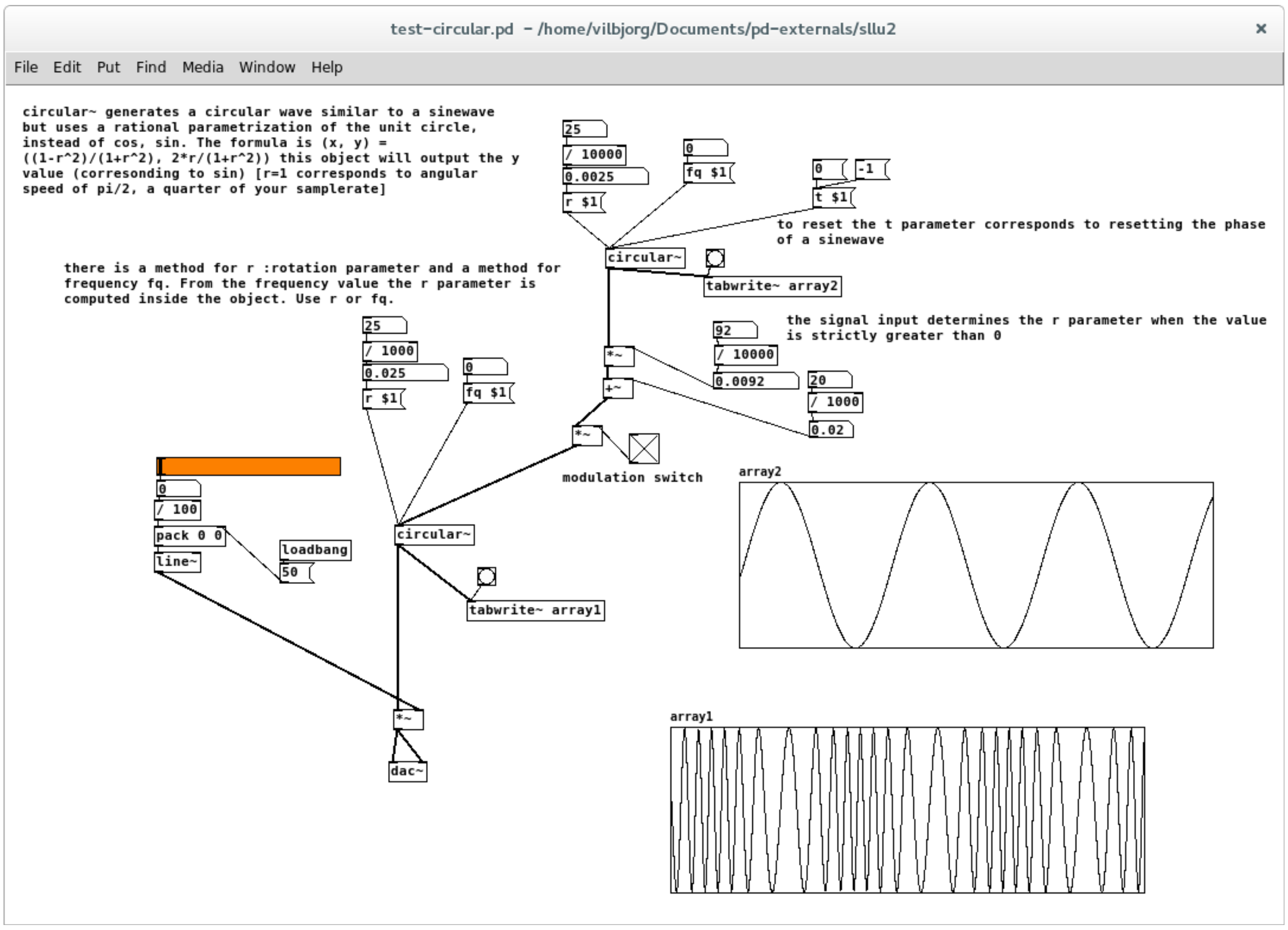
```
vilbjorg@vilbjorg:~/Documents/python/15decpy$ python approx1.py


octave range 8
base frequency 27.5 sample rate 44100
interpolation points in octaves above base: 0 4 8


polynomium:
0.0495811146132 *x^2 + 0.496872218556 *x +  1.14962836505e-05

frequency  ,   approximation  ,  difference in cents
27.5 27.5000000002 1.3154166184e-08
55.0 54.8597813112 -4.4192969851
110.0 109.641988598 -5.64374854913
220.0 219.451081314 -4.32497383412
440.0 440.0 1.57608583988e-11
880.0 884.425252203 8.68403276799
1760.0 1783.26004352 22.7300041731
3520.0 3592.64383842 35.3645880632
7040.0 7040.0 3.84411180458e-13
```



```
vilbjorg@vilbjorg:~/Documents/python/15decpy$ python approx1.py


octave range 4
base frequency 110.0 sample rate 44100
interpolation points in octaves above base: 0 2 4

polynomium:
0.0138068803582 *x^2 + 0.499133074419 *x +  1.03558763082e-05


frequency  ,   approximation  ,  difference in cents
110.0 110.0 -3.62307537582e-10
220.0 219.936344402 -0.50099402
440.0 440.0 1.57608583988e-11
880.0 880.509686484 1.00242177279
1760.0 1760.0 -3.65190621435e-12
```

```
import numpy as np
import matplotlib.pyplot as plt

def aspeed(fq, sRate):
      ph = 2*np.pi*fq / sRate
      return ph

def tpar(ph):
      t = np.sqrt( (1.0  - np.cos(ph))/ (1.0 + np.cos(ph)) )
      return t

def fq(t, sRate):
      x = (1.0 - t*t)/ (1.0 + t*t)
      ph = np.arccos(x)
      fq = ph*sRate/(2*np.pi)
      return fq

def fq2(ph, sRate):
      fq = ph*sRate/(2*np.pi)
      return fq

def cents(tone1, tone2):
      c = 1200* np.log2(tone1/tone2)
      return c

sRate = 44100
basefq = 110.000
numoctaves = 4

minx = aspeed(basefq, sRate)
maxx = np.power(2, numoctaves) * minx

intpoint1 = 0 #choosing interpolation points octaves above base frequency
intpoint2 = 2
intpoint3 = 4

print '\n'
print 'octave range' , numoctaves
print 'base frequency' , basefq, 'sample rate', sRate
print 'interpolation points in octaves above base:' , intpoint1, intpoint2, intpoint3,
x1 = np.power(2, intpoint1)*minx
y1 = tpar(x1)
x11 = x1*x1

x2 = np.power(2, intpoint2)*minx
y2 = tpar(x2)
x21 = x2*x2

x3 = np.power(2, intpoint3)*minx
y3 = tpar(x3)
x31 = x3*x3

a = [[x11, x1, 1.0], [x21, x2, 1.0], [x31, x3, 1.0]]
b = [y1, y2, y3]
pol = np.dot(np.linalg.inv(a), b)
print'\n'
print 'polynomium:'
print pol[0], '*x^2 +', pol[1], '*x + ', pol[2]
print'\n'

def polval(pol, x):
      y = pol[0]*x*x + pol[1]*x + pol[2]
      return y

xa = x1*2
ya = polval(pol , xa)
fqa = fq(ya, sRate)
print 'frequency  ,   approximation  ,  difference in cents '

for i in range (0, numoctaves+1):
      ph = np.power(2, i) * minx
      approx = polval(pol, ph)
      f1 = fq(approx, sRate)
      f2 = fq2(ph, sRate)
      cnts = cents(f1, f2)
      print f2, f1, cnts

print'\n\n'

ph = np.arange(minx, maxx, 0.01)

plt.subplot(111)
yval = polval(pol, ph)
line, = plt.plot(ph, yval, lw=1)
plt.ylim(0.0 , 0.6)
plt.grid(True)


ph = np.arange(minx, maxx, 0.01)

plt.subplot(111)
yval2 = tpar(ph)
line, = plt.plot(ph, yval2, lw=1)
plt.ylim(0.0 , 0.6)
plt.grid(True)

plt.title('polynomial approximation value t(phase)')
plt.show()
```

Pure Data external object for Pure Data by Miller Puckette, you can find Pure Data and a makefile template for the object online via https://puredata.info



```
circular~ generates a circular wave similar to a sinewave
but uses a rational parametrization of the unit circle,
instead of cos, sin. The formula is (x, y) =
((1-r^2)/(1+r^2), 2*r/(1+r^2)) this object will output the y
value (corresonding to sin) [r=1 corresponds to angular
speed of pi/2, a quarter of your samplerate]

to reset the t parameter corresponds to resetting the phase
of a sinewave

there is a method for r :rotation parameter and a method for
frequency fq. From the frequency value the r parameter is
computed inside the object. Use r or fq.

the signal input determines the r parameter when the value
is strictly greater than 0
```

```
/////code for the circular~ object:
#include "m_pd.h"
#include <math.h>


static t_class *circular_tilde_class;


typedef struct circular_tilde {
        t_object x_obj;
        t_sample r_par;
        t_sample t_par;
        t_sample f;

        t_outlet *x_out;

} t_circular_tilde;
```

```
t_int *circular_tilde_perform(t_int *w)
{
        t_circular_tilde *x = (t_circular_tilde *)(w[1]);
        t_sample *in = (t_sample *) (w[2]);
        t_sample *out = (t_sample *)(w[3]);
        int n = (int)(w[4]);
        double t = x->t_par;
        double r = x->r_par;
        double rx;


        while (n--)
        {
                rx = *in++;
                if(0 < rx)
                {
                        r = rx;
                }


                if(t*r == 1)
                {
                        t = -10000.0;
                        *out++ = 0.0;
                }else{
                        t = (t + r)/(1.0 - t*r);
                        *out++ = 2*t/(1 + (t*t));
                }
        }
        x->t_par = t;
        x->r_par = r;
        return (w+5);
}

void circular_tilde_dsp(t_circular_tilde *x, t_signal **sp)
{
        dsp_add(circular_tilde_perform, 4, x, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
}

void *circular_tilde_new(t_floatarg f)
{
        t_circular_tilde *x = (t_circular_tilde *)pd_new(circular_tilde_class);
        x->r_par = f;
        x->t_par = 0;
        x->x_out = outlet_new(&x->x_obj, gensym("signal"));

        return (void *)x;
}
```

```
void circular_set_r(t_circular_tilde *x, t_floatarg f)
{
        x->r_par = f;
}

void circular_set_t(t_circular_tilde *x, t_floatarg f)
{
        x->t_par = f;
}

void circular_set_fq(t_circular_tilde *x, t_floatarg f)
{
        t_float sr = sys_getsr();
        t_sample r = cos(f*M_PI*2.0/sr);
        x->r_par = sqrt((1-r)/(1+r));
}

void circular_tilde_free(t_circular_tilde *x)
{
        outlet_free(x->x_out);
}



void circular_tilde_setup(void)
{
        circular_tilde_class = class_new(gensym("circular~"),
                (t_newmethod)circular_tilde_new,
                (t_newmethod)circular_tilde_free, sizeof(t_circular_tilde),
                CLASS_DEFAULT,
                A_DEFFLOAT, 0);

        class_addmethod(circular_tilde_class,
                (t_method)circular_tilde_dsp, gensym("dsp"), 0);

        class_addmethod(circular_tilde_class,
                (t_method)circular_set_r, gensym("r"),
                A_DEFFLOAT, 0);

        class_addmethod(circular_tilde_class,
                (t_method)circular_set_t, gensym("t"),
                A_DEFFLOAT, 0);

        class_addmethod(circular_tilde_class,
                (t_method)circular_set_fq, gensym("fq"),
                A_DEFFLOAT, 0);
        CLASS_MAINSIGNALIN(circular_tilde_class, t_circular_tilde, f);
}


////////////////////////////////////////////////////////////////////////////
```