Mathematical Derivation of the C++ Routine Used to Initialize Arrays with Sinusoidal Waveforms

Scott Chesnut
Verigy

1. Introduction

Since the early 1970s, digital signal processing has become common place. Yet digital creation of waveforms is not often described. What follows is a description of the mathematical process one uses to create single tones, multi-tones, complex multi-tones and IF signals.

2. Single Tone Creation

In general, a sinusoidal varying quantity can be expressed as

$$F(t) = A\cos(\omega t + \theta) \quad \text{(Equation 1)}$$

Where:

$F(t)$ = Instantaneous value of the cosine function
A = Amplitude or maximum value
ω = Frequency in radians per second
t = Time in seconds
θ = Phase angle in radians (theta)

Since an angle of 2π radians corresponds to one complete cycle, a tone frequency in cycles per second or hertz (Hz) can be described as:

$$f_t = \frac{\omega}{2\pi} \quad \text{(Equation 2)}$$

This can be rearranged as follows:

$$\omega = 2\pi f_t \quad \text{(Equation 3)}$$

By substituting Equation 3 into equation 1 we get:

$$F(t) = A\cos(2\pi f_t t + \theta)$$

Because computers can not emulate continuous time, the variable "t" will be renamed: Δt thus also making the function F a discrete time function. So now the equation becomes:

$$F(\Delta t) = A\cos(2\pi f_t \Delta t + \theta) \quad \text{(Equation 4)}$$

We would like to digitally synthesize f_t. Again, computers can not emulate continuous frequency, so we will have to mathematically represent f_t in terms that are "computer friendly". To do so, assume that the waveform to be created is to be coherent.

If so, it can be described by the following equation:

$$\frac{M}{N} = \frac{f_t}{f_s}$$ (Equation 5, Coherent Sampling Equation)

Where:

M = Number cycles in a waveform; an integer

N = Number of samples used to describe that waveform; an integer.

f_t = The frequency of the tone to be produced; a real number.

f_s = The sample rate; the rate (frequency) at which the waveform samples are sourced or sampled; a real number.

Solving for f_t

$$f_t = f_s \frac{M}{N}$$ (Equation 6)

Substituting this into equation 4:

$$F(\Delta t) = A\cos(2\pi f_s \frac{M}{N}\Delta t + \theta)$$ (Equation 7)

This function which will generate a discrete value $F(\Delta t)$, for every discrete unit of time Δt.

We can initialize an array with a set of these discrete values if this equation is put into a C++ "for" loop – as is done below:

DOUBLE TimeOffset, Δt *;*
ARRAY_DOUBLE F; //Declare an array of double floating point values.

for (Δt *= 0;* Δt < One Waveform Period; Δt + *TimeOffset*)
{
$F[\Delta t] = A\cos(2\pi f_s \frac{M}{N}\Delta t + \theta)$**;** *//Fill array with the function values for each* Δt.
}

Loop control maintenance of the above routine becomes messy, though. The routine has the following problems:

1) If the coherent waveform has more that one waveform period, the loop control ("for") statement breaks down.

2) If the "TimeOffset" variable is not integrally related to the "One Waveform Period" variable, coherency of the waveform is lost.

The solution to problem number one is to replace the variable "One Waveform Period" with another called "Unit Test Period" (UTP) or some form of it. Doing so allows us to describe waveforms of any number of cycles. The equation for UTP is given as:

$$UTP = N * T_s\Big|_{T_s=\frac{1}{f_s}=\text{sec}onds} \qquad \text{(Equation 8)}$$

The UTP is the amount of time required to collect N samples of a waveform. That is, N times the period T_s is the number of seconds required to collect a waveform. With this in mind, we can use the concept of the UTP to help provide a solution to problem number two.

The best way to solve problem number two is to replace Δt with a quantity which is related to the total number of samples to be sourced or measured, or better yet, to the UTP. Doing so will also help represent equation components in "computer friendly" terms.

By substituting $\frac{1}{f_s}$ for T_s in equation 8 above it becomes

$$UTP = N * \frac{1}{f_s}\Bigg|_{\substack{N=TotalNumberOfSamplesInWaveform \\ f_s=frequencyOfSamples}}$$

So one unit of time, Δt, can be described as: $\Delta t = \frac{1}{f_s}$

Two units of time become: $2\Delta t = \frac{2}{f_s}$ and so on. If we replace the numerator of this equation with an integer variable, say "i", then $\Delta t = i * \frac{1}{f_s}$. By doing so we have produced a solution to problem number two and our "for" loop becomes: (we have set θ to zero for simplicity).

```
int i;
ARRAY_DOUBLE F;   //Declare an array of double floating point values.
for (i= 0; i < N; i++)
{
        F[i] = A cos(2πf_s (M/N)(i/f_s)) ; //Fill array with the function values for each Δt = i*(1/f_s)
}
```

This equation can be simplified because the terms f_s in the numerator and denominator cancel out. It then becomes:

```
int i;
ARRAY_DOUBLE F;   //Declare an array of double floating point values.
for (i= 0; i < N; i++)
{
```

$F[i] = A\cos(2\pi \frac{M}{N} i)$ **;** *//Fill array with the function values for each* $\Delta t = i * \frac{1}{f_s}$

}

The resulting array will contain "N" elements of a sine wave which describes a waveform containing "M" cycles. Preservation of coherent waveform representation has been accomplished.

3. Multi Tone Creation

Now that the process of initializing an array with the discrete values of a single tone has been described, extending these concepts to development of multi-tone waveform generation will be visited.

In general, a multi-tone sinusoidally varying quantity can be expressed as the sum of multiple separate tones.

$$F(t) = A_1 \cos(\omega_1 t + \theta_1) + A_2 \cos(\omega_2 t + \theta_2) + ... + A_n \cos(\omega_n t + \theta_n)$$

By the mathematical process we used to derive the single tone discrete time function, the discrete time function for an "n" toned waveform becomes:

$$F[i] = A_1 \cos(2\pi \frac{M_1}{N} i + \theta_1) + A_2 \cos(2\pi \frac{M_2}{N} i + \theta_2) + ... + A_n \cos(2\pi \frac{M_n}{N} i + \theta_n)$$

Where:

M_n = Number cycles in a waveform "n"; an integer. Keep in mind that the number of cycles of each tone must fit integrally into coherent sampling equation and satisfy equation 9 below:

$$\frac{M_1}{N} = \frac{M_2}{N} = \frac{M_3}{N} ... = \frac{M_n}{N} = \frac{f_t}{f_s}$$ (Equation 9, Multi-tone Coherent Sampling Equation)

N = Number of samples used to describe the total UTP of all tones in waveform; an integer. Note that this number is the same for all tones.

θ_n = The absolute phase of the tone (relative to zero radians). This quantity may be used to change the phases of the individual tones relative to another. To convert degrees to radians multiply the desired phase offset by $\frac{2\pi(radians)}{360(\deg rees)} = 0.175 \frac{radians}{\deg ree}$.

The C++ routine to initialize an array with the values of this multi-tone would become:

int i;
ARRAY_DOUBLE F; //Declare an array of double floating point values.
DOUBLE PH;

for (i= 0; i < N; i++)
{

$$F[i] = A_1 \cos(2\pi \frac{M_1}{N} i + \frac{2\pi}{360} PH_1) + A_2 \cos(2\pi \frac{M_2}{N} i + \frac{2\pi}{360} PH_2) + ... + A_n \cos(2\pi \frac{M_n}{N} i + \frac{2\pi}{360} PH_n)$$

}

4. Complex Multi Tone Creation

Now that the process of initializing an array with the discrete values of a multi tone has been described, extending these concepts to development of multi-tone complex waveform generation will be visited.

In General, a multi-tone sinusoidally varying complex quantity can be expressed as the real and imaginary components of the sum of multiple separate tones.

$$F.real(t) = A_1 \cos(\omega_1 t + \theta_1) + A_2 \cos(\omega_2 t + \theta_2) + ... + A_n \cos(\omega_n t + \theta_n)$$

And

$$F.imag(t) = A_1 \sin(\omega_1 t + \theta_1) + A_2 \sin(\omega_2 t + \theta_2) + ... + A_n \sin(\omega_n t + \theta_n)$$

Where:

$F.real(t)$ = The real portion of the complex waveform.

$F.imag(t)$ = The imaginary portion of the complex waveform.

The C++ routine to initialize an array with the values of this multi-tone would become:

int i;
ARRAY_COMPLEX F; //Declare an array of complex double floating point values.
for (i= 0; i < N; i++)
{

$$F.real[i] = A_1 \cos(2\pi \frac{M_1}{N} i + \frac{2\pi}{360} PH_1) + A_2 \cos(2\pi \frac{M_2}{N} i + \frac{2\pi}{360} PH_2) + ... + A_n \cos(2\pi \frac{M_n}{N} i + \frac{2\pi}{360} PH_n)$$

$$F.imag[i] = A_1 \sin(2\pi \frac{M_1}{N} i + \frac{2\pi}{360} PH_1) + A_2 \sin(2\pi \frac{M_2}{N} i + \frac{2\pi}{360} PH_2) + ... + A_n \sin(2\pi \frac{M_n}{N} i + \frac{2\pi}{360} PH_n)$$

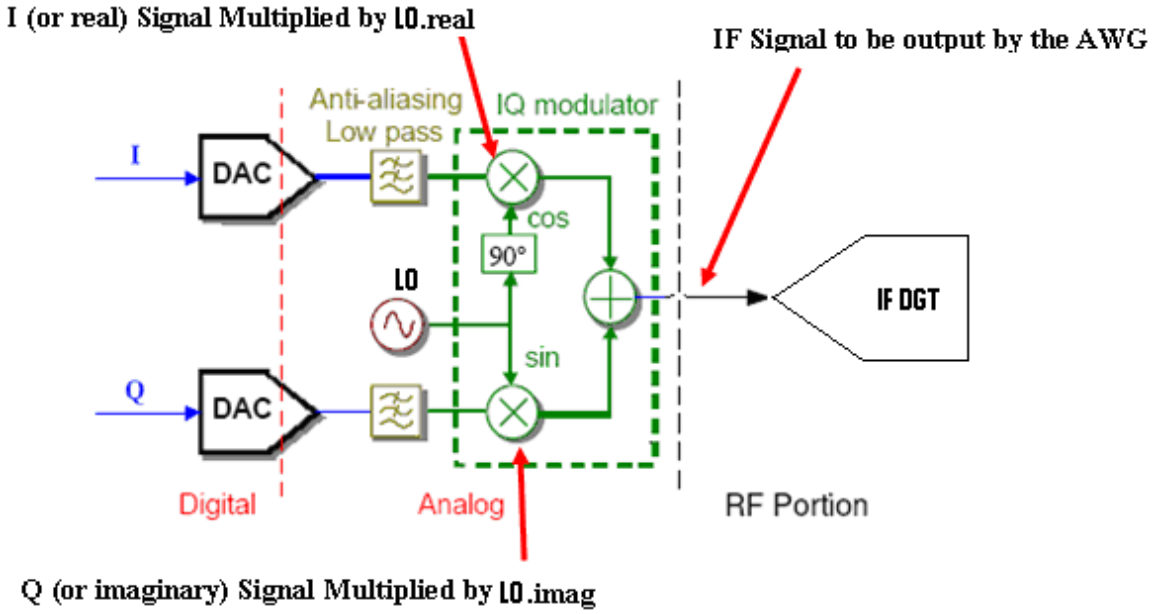
}

5. Up Converted Complex Multi Tone Creation

Now that the process of initializing an array with complex multitones has been described, extending these concepts to development of an IF complex waveform generation will be visited.

IF signals are useful when encoding/decoding data which is to be sent or received using a single ADC (or DAC) – as is the case when information is sent via radio waves to/from an antenna.

Below is a description of how an IF signal could be created and received in hardware (ADCs, mixers, DACs, Local Oscillators). We will want to replicate this process in software and write the IF waveform to an array.



The IF waveform referenced above can be created within a C++ algorithm.

An LO (local oscillator) waveform will be of a very high frequency whose cycle count; Mlo will be very large.

We create an IF signal by multiplying the complex multi-tone by the LO tone in the following approximate manner:

Transmitted IF Waveform = LO*(Multi-Tone)

In reality the multi-tone's real and imaginary components are multiplied by the real and imaginary components of the LO to obtain the IF (real) waveform as follows:

$$IF\_Waveform[i] = Tone1.real[i] * LO.real[i] - Tone1.imag[i] * LO.imag[i]$$

or

$$IF\_Waveform[i] = Tone1.real[i] * LO.real[i] + Tone1.imag[i] * LO.imag[i]$$

Recalling the C++ routine to initialize an array with the values of this multi-tone from the previous section, we have:

```
int i;
ARRAY_COMPLEX F;    //Declare the multi-tone array.
ARRAY_COMPLEX LO;    //Declare the local oscillator array.
```

ARRAY_COMPLEX IF; //Declare the intermediate frequency array.
Double maxval(0);

for (i= 0; i < N; i++)
{

$$F.real[i] = A_1 \cos(2\pi \frac{M_1}{N} i + \frac{2\pi}{360} PH_1) + A_2 \cos(2\pi \frac{M_2}{N} i + \frac{2\pi}{360} PH_2) + ... + A_n \cos(2\pi \frac{M_n}{N} i + \frac{2\pi}{360} PH_n)$$

$$F.imag[i] = A_1 \sin(2\pi \frac{M_1}{N} i + \frac{2\pi}{360} PH_1) + A_2 \sin(2\pi \frac{M_2}{N} i + \frac{2\pi}{360} PH_2) + ... + A_n \sin(2\pi \frac{M_n}{N} i + \frac{2\pi}{360} PH_n)$$

$$LO.real[i] = A_{lo} \cos(2\pi \frac{M_{lo}}{N} i)$$

$$LO.imag[i] = A_{lo} \sin(2\pi \frac{M_{lo}}{N} i)$$

$$IF[i] = F.real[i] * LO.real[i] - F.imag[i] * LO.imag[i]$$

//find the max value of the IF array so that it can be used later to keep from
//clipping the waveform as it comes out of an AWG.
if (fabs(IF[i]) > maxval) maxval = fabs(if[i]);
}

//Scale the real waveform to keep from clipping the waveform as it comes out of the AWG
for (i= 0; i < N; i++)
{

$$IF[i] = IF[i] / \max val;$$

}

The resulting IF waveform is not complex and can be sent or received from a single DAC or ADC just as it could be sent out a single antenna.