A Rotary Speaker Modeling Plug-In

Ross Penniman

Introduction

In this app note we will explore the topic of modeling a physical device using DSP techniques.

One of the most distinctive sounds of popular music in the last 50-plus years has been the sound of the Hammond B3 organ. From lush, atmospheric textures, to screaming roars, the instrument has a wide range of expressive possibilities. Part of what makes the sound so unique, and so expressive, is that it is usually played through a rotating speaker, the most famous being those made by the Leslie company. For convenience we will refer to this speaker simply as “the Leslie speaker.”

Rotary Speaker Sim is our plug-in that we will develop in this app note that simulates the sound of the Leslie speaker using digital signal processing. While this plug-in is by no means a perfect re-creation of the original, we will try to capture the most prominent features of the sound while retaining a relatively low complexity.

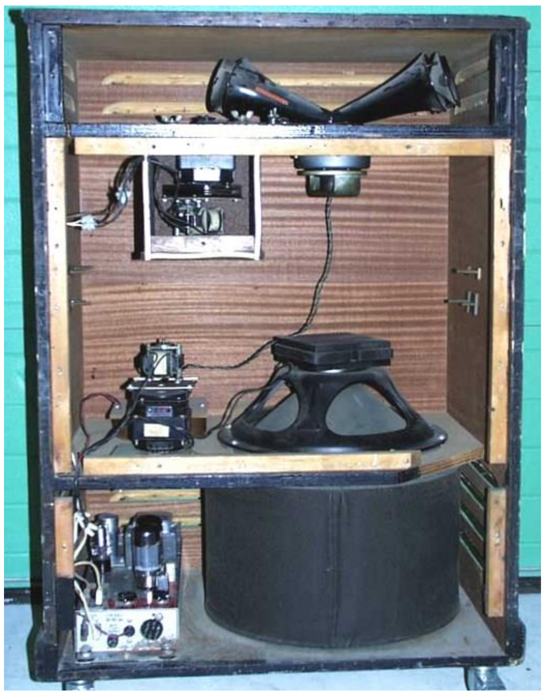
Fig. 1: The back side of a Leslie Model 147 speaker.

The great majority of the technical information needed to create this plug-in can be conveniently found in an article by Clifford Henricksen [1]. The information in said article pertains to Leslie Models 122, 145, and 147. These models all share similar features: a single-channel 40-watt tube amplifier, an 800 Hz passive crossover, a rotating treble horn, and a bass speaker that fires into a rotating drum-shaped baffle (see Fig. 1).

Modeling Techniques

As we begin a plan of how to create a model of the Leslie speaker, it is best to look at the major features of the “Leslie sound” and how they can be emulated using DSP techniques.

The first feature to simulate is the tube amplifier which powers the speakers. The saturation of this amplifier is an important part of the sound, so it is simulated using a waveshaper based on the equation:

$$y(n) = \left(\frac{1}{\arctan(k)}\right) * arctan(k \cdot x(n))$$

In this equation (from [2], p. 497), the amount of distortion can easily be altered by changing the value of k.

The rotating treble horn is by-and-large the essence of the Leslie sound. The basic idea is that the sound is reproduced from a compression driver connected to a horn that spins in a circle. As the horn moves closer or further away from the microphone (in this case a simulated microphone), there are three things that happen.

1) Due to the motion of the horn, there is a small Doppler shift in the pitch. As the horn is moving towards the microphone, the pitch rises, and as the horn moves away, the pitch falls. In musical terms, this creates frequency modulation, or vibrato.
2) Due to the narrow directivity of the horn, and the changing distance between the horn and the microphone, the loudness of the sound changes with time. This creates an amplitude modulation, or tremolo effect.
3) Also due to the directivity of the horn, the tonal quality, or timbre, changes as the horn alternately points towards or away from the microphone. When the horn is pointing away, the high frequencies will be muffled, as they are much more directional than the low frequencies.

It should be mentioned that while the treble rotor has two horns facing in opposite directions, only one of them produces sound. The other is simply a counter balance to keep the whole assembly from wobbling as it spins.

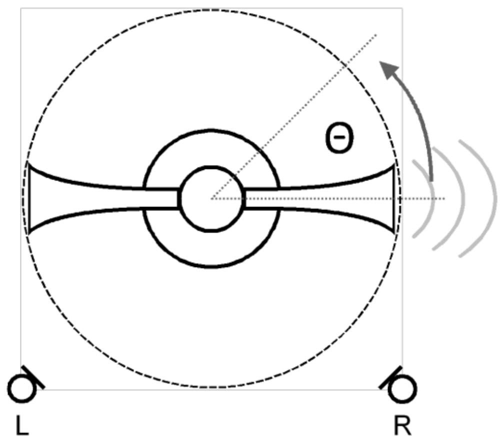

Fig. 2: Geometry of simulated rotating horn and microphones. Zero-degree axis points to the right.

A complete physical model of the speaker and all possible microphone positions could be difficult to design, and computationally expensive to implement, so we can make a few simplifications. The pick-up of sound is assumed to be two virtual microphones placed at the corners of a square which inscribes the path of the rotating horn (as shown in Fig. 2). These microphones are assumed to be omnidirectional with flat frequency response and are placed in anechoic surroundings (not very realistic, but it saves quite a bit of additional DSP). The three features described above are simulated, respectively, by the following methods:

1) A circular delay buffer is used to simulate the propagation time from the mouth of the horn to the left and right microphones. As the delay time shortens and lengthens, the Doppler effect will happen automatically. Only one delay line is needed since there is only one source. The left and right signals each have their own read index which moves independently through the buffer.
2) The relative distance between the horn and microphones is used as the basis for the amplitude (tremolo) effect. This will be applied as a simple modulated gain.
3) A second order Butterworth low-pass filter with changing cutoff frequency is used to simulate the tonal modulation effect.

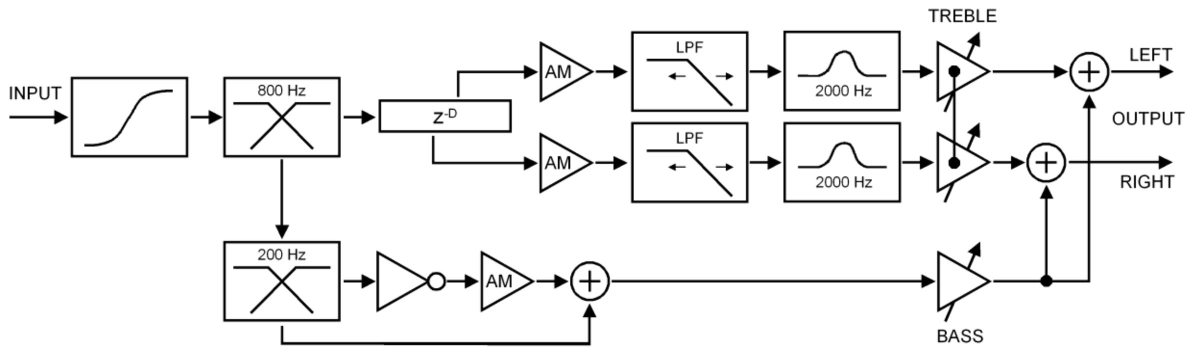

Fig. 3: Block diagram of the *Rotary Speaker Sim* algorithm

In addition to these dynamic elements, it is also important to emulate the static frequency response of the horn. The real Leslie speaker exhibits a strong band-pass characteristic centered at 2 kHz [1]. We will simulate this using a peaking filter with a center frequency of 2 kHz and gain of 10 dB.

The crossover between the treble and bass speakers is a passive circuit with a crossover frequency of 800 Hz. We will implement this using a second order Linkwitz-Riley crossover.

In a real Leslie speaker, the bass driver fires downward into a scoop-shaped baffle that is mounted inside a rotating drum. When the scoop is facing out toward the open side of the cabinet, the sound is un-inhibited, however when the scoop is facing in toward the closed side the cabinet, the sound is muffled, and reduced in volume. The modulation effect of the rotating baffle is simulated using a tremolo effect with an envelope that has a sinusoidal shape in decibels. However, not all frequencies are affected equally by the baffle. Frequencies below 200 Hz are likely unaffected because of their long wavelength and acoustic energy. Thus, there is an additional Linkwitz-Riley crossover to separate the modulated bass from the un-modulated bass. Because of the phase response of the crossover, the higher, modulated, signal is inverted to assure the correct frequency response when the signals are re-combined.

One of the advantages of having a modeled device is that you can adjust parameters that are fixed on the real thing. A convenience feature we can put in is the ability to balance the overall levels of the bass and treble signals before they are combined into the final stereo output. Refer to the block diagram in Fig. 3 to see the algorithm as a whole.

Geometry Calculation

At any instance in time, there are two values that must be computed for each virtual microphone: the distance between the mouth of the horn and the microphone, and the angle between the same. Fig. 2 shows the geometry used to solve for the distance and angle, the 0 degree axis points to the right. This distance can be calculated by the following equations:

$$x_L = r_h + r_h \cdot \cos\theta$$

$$x_R = r_h - r_h \cdot \cos\theta$$

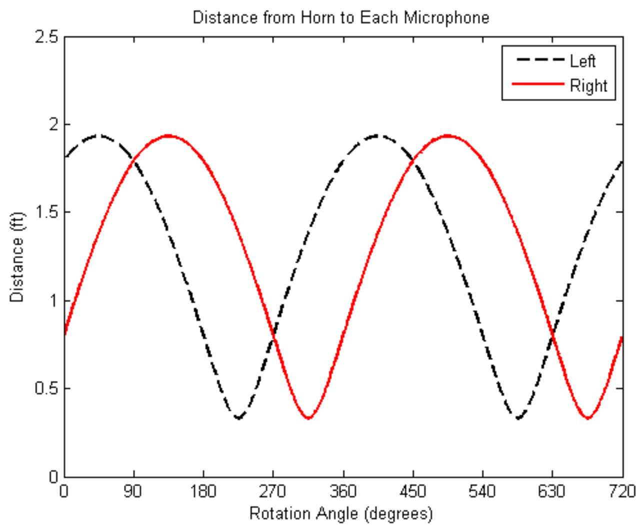

Fig. 4: Distance between the treble horn and each microphone, with respect to rotation angle.

$$y = r_h + r_h \cdot \sin\theta$$

$$d_L = \sqrt{x_L^2 + y^2}$$

$$d_R = \sqrt{x_R^2 + y^2}$$

Where θ is the rotation angle, x_L and x_R represent the x distance to each microphone, r_h is the radius of the horn, and d_L and d_R represent the total distance to each microphone. The distance is the basis of the frequency and amplitude modulation (see Fig. 4).

When the horn is pointed directly at the microphone the angle between them is 0 degrees, and when pointed directly away, this angle is 180 degrees. The left microphone has a "rotation angle" of -135 degrees, and the right microphone has a "rotation angle" of -45 degrees. The angle between horn and microphone is the basis of the tonal modulation (see Fig. 5).

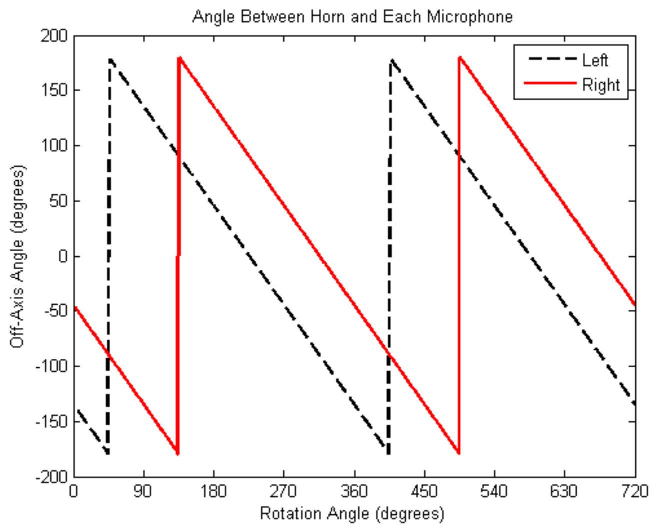

Fig. 5. Angle between treble horn and each microphone, with respect to rotation angle.

Motor Control

In addition, one of the distinctive characteristics of the Leslie speaker is that it has both a fast and a slow speed. The change in sound as it accelerates or decelerates is a much-loved expressive feature. The plug-in has an auto speed control mode that allows the user to choose between "Off", "Slow", and "Fast" for the desired speed. The plug-in will accelerate and decelerate the rotation according to parameters set by the user. Once the target rotation speed is reached, it will remain constant until another speed is selected. The treble and bass rotor are controlled separately so that they can have separate acceleration/deceleration speeds and are not synchronized.

Controls

The parameters for the plug-in are as follows:

- Input Drive (distortion level)
- Treble Rotational Speed (in RPM)
- Treble Horn Radius (in feet)
- Treble Amplitude Modulation Depth
- LPF Modulation Depth (in octaves)
- LPF Center-Cutoff Frequency
- Treble Horn Resonance Center Frequency
- Treble Horn Resonance Gain
- Bass Rotational Speed (in RPM)
- Bass Amplitude Modulation Depth
- Bass to Treble Crossover Frequency
- Overall Treble Volume
- Overall Bass Volume

In addition, if the user selects automatic speed control, the manual rotational speed controls for bass and treble are disabled, and instead the user can adjust the acceleration and deceleration time (in seconds) for the bass and treble rotation.

Key Functions

Following through the block diagram we will discuss the basic techniques of coding this plug-in. The complete code is given in the project files. The first item, the waveshaper, is a single equation and does not require further explanation.

The two sets of Linkwitz-Riley crossovers are handled by means of four CBiQuad objects called mHornHPF, mBassLPF, mBassHPF, and mBypassLPF. The coefficients for these filters are calculated in the “crossCoeffs” function. The design equations for the filters can be found on p. 186 of [2].

The modulated delay buffer has a single write index mdWriteIdx, and two read indices, mfReadIdxL, and mfReadIdxR. The write index simply increments one step each time processAudioFrame is called, and wraps when it reaches the end of the delay buffer. The read function is where the work is done, it uses linear interpolation to obtain fractional delay outputs from the delay buffer:

```
void CRotarySpeakerSim::getDelOutputs()
{
	// This function uses linear interpolation to obtain fractional delay
	// outputs from the delay buffer.

	int mdReadIdxL0, mdReadIdxL1, mdReadIdxR0, mdReadIdxR1;
	float fracDel;

	// Left Channel
	// calc indicies
	mfReadIdxL = (float)mdWriteIdx - mfDelayL;
	mdReadIdxL0 = (int)floorf(mfReadIdxL);
	mdReadIdxL1 = mdReadIdxL0 + 1;
	fracDel = mfReadIdxL - (float)mdReadIdxL0;
	// wrap indicies
	if (mfReadIdxL < 0) mfReadIdxL += mdBufferSize;
	if (mdReadIdxL0 < 0) mdReadIdxL0 += mdBufferSize;
	if (mdReadIdxL1 < 0) mdReadIdxL1 += mdBufferSize;

	// interpolate output
	mfDelOutL = ((1.0 - fracDel) * mpDelayBuf[mdReadIdxL0]) + (fracDel * 
mpDelayBuf[mdReadIdxL1]);

	// Right Channel
	// calc indicies
	mfReadIdxR = (float)mdWriteIdx - mfDelayR;
	mdReadIdxR0 = (int)floorf(mfReadIdxR);
	mdReadIdxR1 = mdReadIdxR0 + 1;
	fracDel = mfReadIdxR - (float)mdReadIdxR0;

	// wrap indicies
	if (mfReadIdxR < 0) mfReadIdxR += mdBufferSize;
	if (mdReadIdxR0 < 0) mdReadIdxR0 += mdBufferSize;
	if (mdReadIdxR1 < 0) mdReadIdxR1 += mdBufferSize;

	// interpolate output
	mfDelOutR = ((1.0 - fracDel) * mpDelayBuf[mdReadIdxR0]) + (fracDel * 
mpDelayBuf[mdReadIdxR1]);

	return;
}
```

The coefficients for the low-pass filters (LPF's) are calculated using the design equations given on p. 183 of [2] in the function "calcCoeffs." This function is called from the "updateTheta" function (discussed soon) which is also called on every iteration of processAudioFrame. The coefficients for the horn frequency response are calculated using the constant Q peaking filter design equations given on pp. 192-193 of [2]. The function for calculating these values for the horn filter is called "calcHornCoeffs."

As mentioned, the "updateTheta" function handles the calculation of geometry as well as relating the horn position to delay, cutoff frequency, and amplitude modulation. The first half of the function is given here:

```
void CRotarySpeakerSim::updateTheta()
{
	float xBase = mfHornRad1 * cos(mfTheta);
	float xTL = mfHornRad1 + xBase;
	float xTR = mfHornRad1 - xBase;
	float yT = mfHornRad1 + (mfHornRad1 * sin(mfTheta));

	// Distance to source
	mfDelayL = sqrtf((xTL*xTL) + (yT*yT));
	mfDelayR = sqrtf((xTR*xTR) + (yT*yT));

	// Compute amplitude modulation
	mfAmplL = mfAmplMod*(mfHornRad1 - mfDelayL)/mfHornRad1;
	mfAmplR = mfAmplMod*(mfHornRad1 - mfDelayR)/mfHornRad1;

	float fcLeft = 0.5 + (mfHornRad1 - mfDelayL)/mfHornRad1;  // ~ +/-1
	float fcRight = 0.5 + (mfHornRad1 - mfDelayR)/mfHornRad1;

	// Convert amplitude modulation to gain factors
	mfGainL = powf(10.0, (mfAmplL/20.0));
	mfGainR = powf(10.0, (mfAmplR/20.0));

	// Convert delay to samples
	mfDelayL *= (float)m_nSampleRate/mfSpdSound;
	mfDelayR *= (float)m_nSampleRate/mfSpdSound;

	// Update Fc values and filter coefficients
	fcLeft = mfTrebLPFfc * powf(2.0, mfTrebLPFModDepth*fcLeft);
	fcRight = mfTrebLPFfc * powf(2.0, mfTrebLPFModDepth*fcRight);

	calcCoeffs(fcLeft, fcRight);
```

The second half of the "updateTheta" function has to do with the automatic speed control. It increments the current speed if the horn is accelerating, and decrements the speed if it is braking. You may also notice a few commands in there that deal with the horn radius. This is because I found that a smaller horn radius sounds better at high speed, which a larger horn radius sounds better at low speed. The final step is to increment mfTheta, which is the actual variable that keeps track of the horn position.

```
// Auto Speed Control
	if (bUseAutoSpeed && bSpeedChange)
	{
		if (bSpeedAccel)
		{
			mfCurSpeed += mfSpeedInc;
			mfHornRad1 = mfSlowHornRad - (mfHornRPMtoRadConv * (mfCurSpeed - 
mfSlowSpeed));
			if (mfCurSpeed >= mfSpeedTarget)
			{
				bSpeedChange = FALSE;
				sendStatusWndText("Speed Accel Finished");
			}
		}
		else	// Decelerate
		{
			mfCurSpeed += mfSpeedDec;
			mfHornRad1 = mfSlowHornRad - (mfHornRPMtoRadConv * (mfCurSpeed - 
mfSlowSpeed));
			if (mfCurSpeed <= mfSpeedTarget)
			{
				bSpeedChange = FALSE;
				sendStatusWndText("Speed Brake Finished");
			}
		}

		// Convert from RPM to rad/sample
		mfThetaInc = (2.0*pi*mfCurSpeed)/(60.0*(float)m_nSampleRate);
	}

	// increment and wrap
	mfTheta += mfThetaInc;
	if (mfTheta > (2.0*pi)) mfTheta -= (2.0*pi);
```

```
    return;
}
```

It is worth noting that there is also a function to handle the rotation of the bass element, called updateBassTheta. This function is similar to updateTheta, but is much simpler, since the bass element only uses amplitude modulation.

Another key part of the automatic speed control is how the logic is handled for selecting different speeds. This is part of the userInterfaceChange function. The radio buttons in question are control number 42.

```
case 42:
{
    if (bUseAutoSpeed) // ignore new data if speed is manual
    {
        if (muAutoSpeed == OFF)
        {
            mfSpeedTarget = mfOffSpeed;
            mfSpeedTargetB = mfOffSpeedB;
        }
        else if (muAutoSpeed == SLOW)
        {
            mfSpeedTarget = mfSlowSpeed;
            mfSpeedTargetB = mfSlowSpeedB;
        }
        else if (muAutoSpeed == FAST)
        {
            mfSpeedTarget = mfFastSpeed;
            mfSpeedTargetB = mfFastSpeedB;
        }

        // Determine if treble rotor is accelerating or decelerating
        if (mfCurSpeed > mfSpeedTarget)
        {
            bSpeedAccel = FALSE;
            bSpeedChange = TRUE;
        }
        else if (mfCurSpeed < mfSpeedTarget)
        {
            bSpeedAccel = TRUE;
            bSpeedChange = TRUE;
        }
        else
        {
            bSpeedChange = FALSE;
        }

        // Determine if bass rotor is accelerating or decelerating
        if (mfCurSpeedB > mfSpeedTargetB)
        {
            bSpeedAccelB = FALSE;
            bSpeedChangeB = TRUE;
        }
        else if (mfCurSpeedB < mfSpeedTargetB)
        {
            bSpeedAccelB = TRUE;
            bSpeedChangeB = TRUE;
        }
        else
        {
            bSpeedChangeB = FALSE;
        }
    }

    break;
}
```

Final Thoughts

The only way to really understand something is to spend some time working through the code, which is why I have not tried to explain every detail. The source code is provided so that you can understand the remaining details for yourself.

The distortion produced by the wave shaper is an important part of the sound, but it can also be problematic. Since it is a non-linear process, it adds additional (higher) frequencies to the signal coming in. This has the potential to cause aliasing, which is a very unpleasant sound. Ideally, oversampling should be used to eliminate this possibility. Fortunately, in the intended usage of this plug-in, the problem is not readily noticeable as tone-wheel organ sounds often lack strong high frequency components and distortion levels are often kept to a moderate level.

References:

[1] C. Henricksen, "Unearthing the Mysteries of the Leslie Cabinet" *Recording Engineer/Producer*, April 1981.
(also available on-line at: http://www.theatreorgans.com/hammond/faq/mystery/mystery.html)

[2] W. Pirkle, *Designing Audio Effect Plug-Ins in C++*, Focal Press, 2013