The Stereo Position Simulator

Ross Penniman
AES Student Design Competition
University of Michigan Student Section

Summary:

The Stereo Position Simulator (SPS) is a set of three VST plug-ins which provide a more sophisticated algorithm for panning than what is provided in standard audio software. SPS Basic provides intuitive control of inter-channel level and time differences, while SPS Orch and SPS Piano model the pickup of a stereo pair of microphones. The development was done in Matlab and the final implementation in C++.

Parameter Descriptions:

SPS Basic:

Pan:	Left (-100 %) to right (+100 %) position.
Level Dif:	Maximum inter-channel level difference applied. (interpolated using linear function)
Time Dif:	Maximum inter-channel time difference applied. (interpolated using sinusoid function)

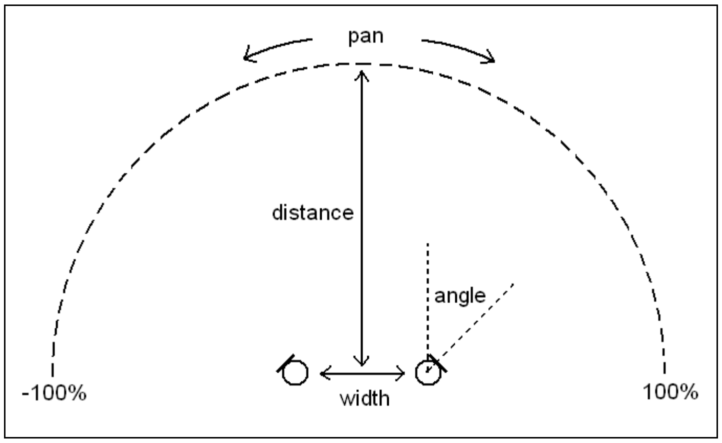
SPS Orch:

Pan:	Left to right position, modeled on semi-circle.
Distance:	Radius of the source arc from the center of the microphone pair.
Width:	Spacing between the microphones.
Pattern:	Polar pattern of the microphones, choose from: Omni, Wide-Cardioid, Cardioid, Hyper-Cardioid or Figure-of-8.
Angle:	Angle which the microphone's axis forms with forward.

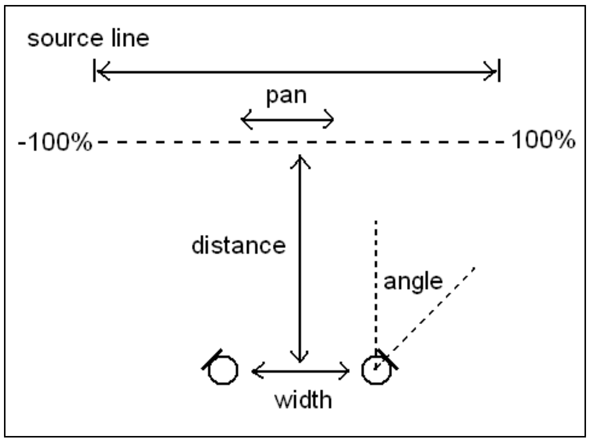
SPS Piano:

Pan:	Left to right position, modeled on straight line.
Source Line:	Length of the line on which the source is located.
Distance:	Distance between the microphone pair and the source line.
Width, Pattern, Angle:	See above.

SPS Orch



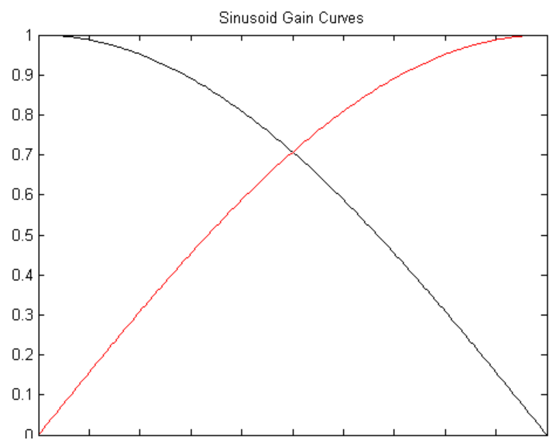
SPS Piano

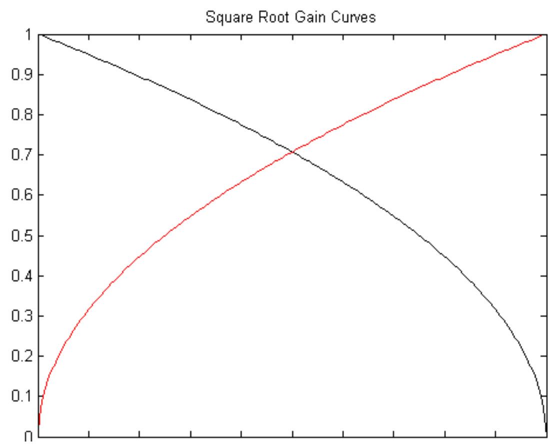


The Creation of The Stereo Position Simulator

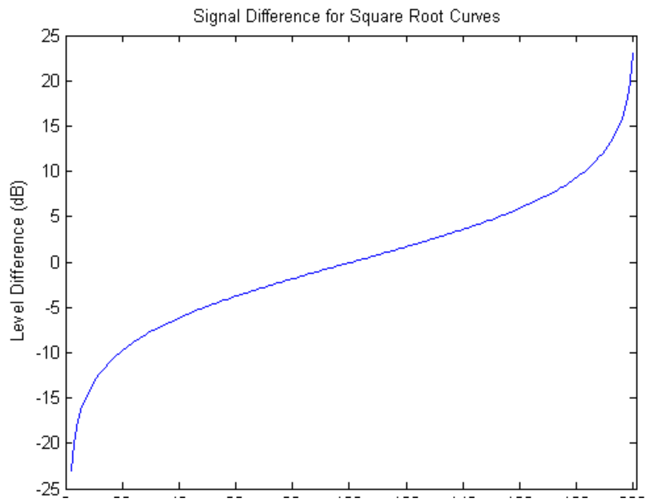
The Stereo Position Simulator (SPS) is a set of three VST plug-ins. The original intent was to provide a more sophisticated method of creating a stereo image beyond the simple amplitude differences used in most software. The inspiration for these plug-ins came from a lecture on binaural sound by Prof. Gregory Wakefield at the University of Michigan and from the application *Image Assistant 2.0* by Theile and Wittek (http://www.hauptmikrofon.de/).

The majority of pan position algorithms use either a sinusoid or square-root function to derive the gain curves applied to the left and right channels.





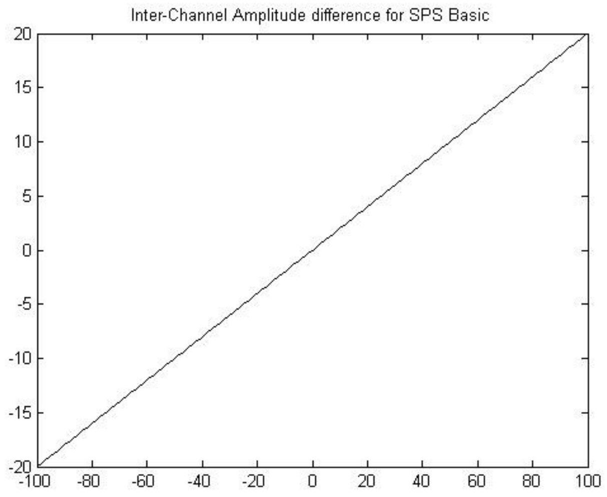
For general purpose use, both algorithms work quite well. They have the benefit that the power sum is flat from left to right, meaning that equal loudness will be maintained regardless of the sound position. They also have the benefit that a signal panned hard left or right will only be present in one speaker, useful for when a sound source is stereo and needs to be kept that way. This same property is not very aesthetically pleasing for monaural sources, however. Near the edges, the image will suddenly jump into one speaker only. This can be seen in the level differences between the left and right channels shown below (square root gain curves).

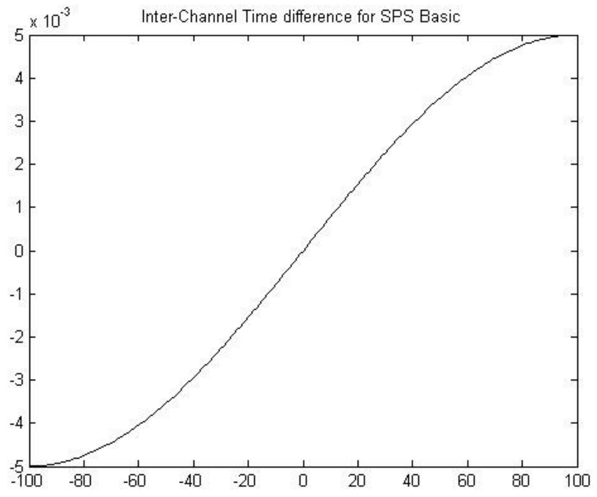


The other downfall of the traditional panning algorithms is that they do not using any timing differences to create the stereo image. Timing differences are key for precise sound localization as well as providing the aesthetic quality of spaced microphones. A sound which arrives from the left side slightly ahead of the right will be heard as originating from the left, even if the levels are equal.

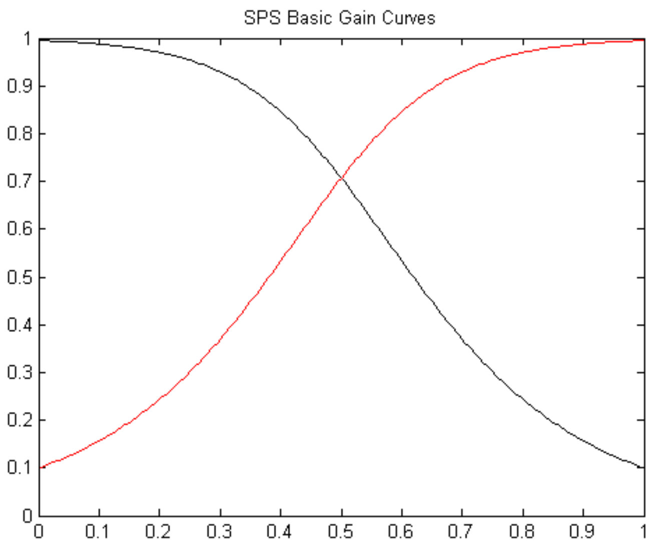
What I noticed in some of the charts in *Image Assistant 2.0* was that for many stereo microphone pairs, the amplitude relationships tend to be linear (in dB) or close to it. I also noticed that for spaced microphones, the timing differences between the microphones seem to follow a sinusoid function. I decided that creating a panning algorithm modeled after stereo microphones would probably provide a more aesthetically enjoyable stereo image.

The first plug-in to be developed, "SPS Basic," uses linear amplitude differences between the channels and a sinusoidal delay curve. The user specifies the ranges that these amplitude and time differences can take.





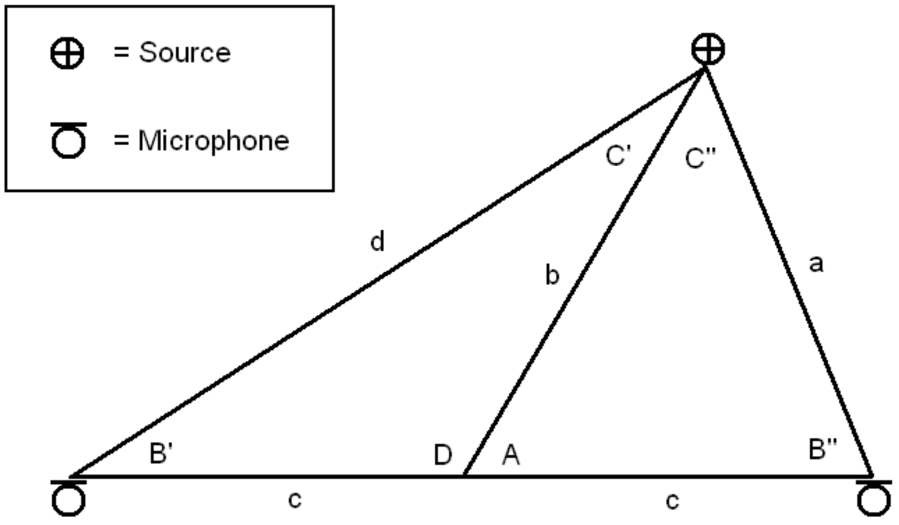
The gain curves are compensated so that the power sum will still be flat from left to right.



After figuring out the algorithms needed to calculate the gain and time curves, I wanted to check how closely they corresponded with the imaging characteristics of the stereo microphone pairs I was trying to imitate. In the process of writing Matlab scripts for testing purposes, I came on the idea that I could generate a custom set of gain and timing curves based on a user defined pair of microphones.

Time for some trigonometry!

I figured out that for a given source position, the gain and timing characteristics of a pair of microphones could be determined if I could calculate the relative distances between each microphone and the source as well as the angle of incidence that the incoming sound made with the pickup axis of the microphone. The problem looks something like this:



Given angle **A** and side lengths **b** and **c**, we have to solve for side lengths **a** and **d**, as well as the angles **B'** and **B"**. This calls for some Law of Cosines action!

The Law of Cosines tells us that: $\mathbf{a^2 = b^2 + c^2 - 2bc \cos A}$. Which means that we can solve for **a** and **d** and get:

$$a = \sqrt{b^2 + c^2 - 2bc\cos(A)} \qquad d = \sqrt{b^2 + c^2 + 2bc\cos(180 - A)}$$

Likewise we can also solve for the angles:

$$B' = \arccos\left(\frac{-b^2 + d^2 + c^2}{2dc}\right) \qquad B'' = \arccos\left(\frac{-b^2 + a^2 + c^2}{2ac}\right)$$

We account for the various polar patterns of the microphones using polar math. Every polar pattern is some combination of an omni-directional pickup and a figure-of-eight pickup. In polar coordinates this is:

Omni:	Gain = 1
Figure-of-Eight:	Gain = $\cos(\theta)$

(θ = angle of incidence)

A cardioid pattern can be obtained by taking the average of these two:

Cardioid:	Gain = $0.5 + 0.5*\cos(\theta)$

Likewise, other patterns including wide-cardioid or hyper-cardioid can be obtained by some weighting of the omni and figure-of-eight responses:

$$\text{Wide Cardioid:} \quad \text{Gain} = 0.7 + 0.3*\cos(\theta)$$
$$\text{Hyper-Cardioid:} \quad \text{Gain} = 0.3 + 0.7*\cos(\theta)$$

Finally, we need to account for the fact that a sound which is much closer to one microphone than the other will also be louder in that microphone. The inverse square law tells us that the intensity (power per unit area) of a sound wave is proportional to the inverse of the distance squared (from the source). We also know that acoustic intensity is proportional to the power of the microphone's audio signal and that power in an audio signal is proportional to the square of its amplitude. From this we can see that the amplitude of the resulting audio signal is proportional to the inverse of the distance from the source to the microphone.

$$\text{Intensity} \sim \frac{1}{(\text{distance})^2} \qquad \text{Power} \sim (\text{amplitude})^2$$

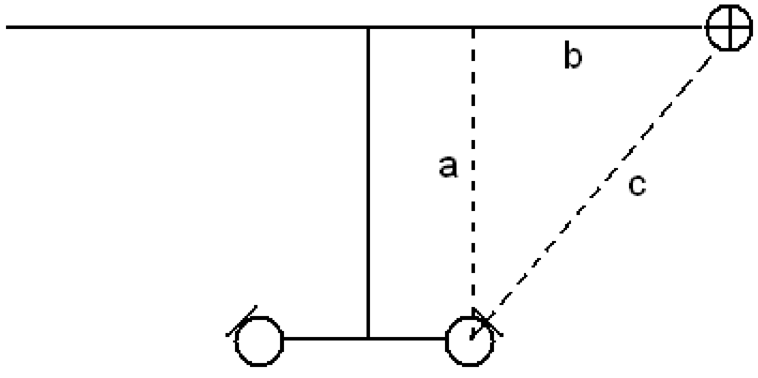
$$\text{Intensity} \sim \text{Power} \qquad \text{amplitude} \sim \frac{1}{\text{distance}}$$

What this boils down to is that we can use the distance to the microphone to find a gain factor that will appropriately compensate for the inverse square law. Specifically:

$$\text{Gain} = \frac{\text{Reference Distance}}{\text{Distance to Microphone}}$$

Since the geometry described above models the source as being at some point along a semi-circle, and orchestras generally sit in semi-circles, I decided to call this version of the plug-in "SPS Orchestra" or "SPS Orch" for short.

I also had the idea that it would also be cool to be able to model the source as being at some point along a straight line. The geometry for this setup is much simpler.



The relationship of the source and microphone will always form a right triangle. If **a** and **b** are known then all we need is the Pythagorean theorem and an arctangent to figure out the distance and angle.

Because this geometry is particularly applicable to the way a piano is recorded, this plug-in is called "SPS Piano."

Development

- Used Matlab to develop and test algorithms.

- Created scripts for each plug-in to understand the structure of the code and do as much de-bugging as possible. (38 separate scripts by the time I was done!)

- Matlab makes it easy to write code in a short amount of time because it is an interpreted language (it doesn't need to be compiled before it runs).

- Matlab can handle large sets of data as single variables, which greatly simplifies the code.

- Matlab has very useful functions for plotting and for exporting data as an audio file.

Implementation

- Used the VST plug-in Software Development Kit (SDK) available for free download from Steinberg.

- Programming done in C++ (Microsoft Visual C++)

- VST plug-in exists as a class to which you add member functions and member variables.

- VST host will call member functions as appropriate.

- Actual audio processing is done in the "process( )" function, done in blocks of samples.

- Other member functions are used to store, recall, and process parameters.

- Gain and time information is stored in a 201 point look-up table. This table is re-computed every time a parameter (other than "Pan") is changed. This minimizes the processing needed to sweep the pan position. Linear interpolation is used to generate values for pan locations that fall in-between table values.

- Precise timings are achieved by using a variable length circular delay buffer with linear interpolation.

Additional Thoughts

Are there any existing plug-ins which do the same thing?

There are plug-ins which have similar functionality, but none that are exactly the same. Some plug-ins allow you to set level and time differences for a signal, but they do not provide an interface which puts these values under intuitive control. Other plug-ins have sophisticated binaural imaging algorithms which are based off of Head Related Transfer Functions (HRTF's) but these are generally only effective while using headphones. There is also a guy by the name of Aristotel Digenis who has published several plug-ins dealing with ambisonics. While very sophisticated, they only work with level differences, not timing differences.

What sort of improvements are planned for the future?

The biggest one is a proper user interface. Right now the plug-ins simply use the default VST interface which is functional but nothing more. An attractive and intuitive layout would definitely help a lot. Ultimately, it would be great to have click-and-drag capability for changing things like source position and microphone properties. It would also help to implement more sophisticated interpolation in the delay buffer and look-up table so that when the plug-in parameters are changing there are no artifacts in the sound.

What are the recommended uses for each of the plug-ins?

As with anything creative, there are no rules. Whatever sounds the best for what you are trying to do is the right choice to make. That said, SPS Basic is probably the most useful for production work as, in many ways, it represents the "ideal" stereo microphone pair. The other two plug-ins are more for educational purposes. They allow you to hear the different imaging characteristics of an infinite variety of microphone pairs. Overall, these plug-ins do not present any technological breakthroughs, but simply provide a very useful implementation of existing technology.

What are the sonic differences between the plug-ins and a real stereo mic pair?

There are two key factors which prevent SPS from being a realistic simulation of a stereo microphone pair. First, in any real situation, not only would the direct sound be picked up by the microphones but also early reflections as well. These early reflections contribute quite a bit to the imaging of a stereo recording. The math in SPS basically assumes that you are recording in an anechoic chamber, not something that happens very often! Second, the polar patterns used in SPS are mathematically perfect. The same cannot be said of the directivity of real microphones. Most omni-directional microphones are actually directional in the higher frequencies. Likewise, most cardioid microphones are only truly cardioid around 1 kHz.