Songlib: built-in filters

Song Li Buser

Revision Date: November 7, 2011

The following filters come with the **songlib** system. For information on how to use them, see filters .

Funtion:

```
void lowPass(int *data,int length,double frequency,double resonance);
```

This filter removes the higher frequencies from the audio data. Frequencies above the given *frequency* are severely attenuated. According to the http://www.musicdsp.org website, the *resonance* parameter should be between sqrt(2) (lowest resonance) to 0.1 (highest resonance), assuming *frequency* is greater than three or four kilohertz. For more info on acoustic resonance, see http://en.wikipedia.org/wiki/Acoustic_resonance.

4000 : the 1kHz - 4kHz mid frequency band is where the human ear is most sensitive

1.414 : the square root of 2 (1.414) is the ratio between the average and peak values of a sine wave

Function:

```
   void highPass(int *data,int length,double frequency,double resonance);
```

Like *lowPass* but attenuates frequencies below the given *frequency*.

Function:

```
    void amplify(int *data,int length,double amp);
```

This filters scales all the values in *data* by *amp*. If *amp* is greater than one, the effect will be to increase the volume. Conversely, a value less than one will decrease the volume.

Function:

```
    void attackLinear(int *data,int length,double amp,double delta)
\color{black}

There are two important cases for this filter. The first case is:

\begin{verbatim}
    amp < 1 and delta > 0
```

This softens the first part of the note. Sample i is scaled thusly:

```
    if (amp + delta * i < 1)
i = i * (amp + delta * i);
```

The first sample is scaled the most, the second a little less, the third, a little less yet, and so on.

The other important case is:

```
amp > 1 and delta < 0
```

This increases the loundess of the first part of the note. Sample i is scaled thusly:

```
    if (amp + delta * i > 1)
i = i * (amp + delta * i);
```

Here are two typical calls:

```
attackLinear(data,length,0.5,0.0002);

attackLinear(data,length,1.5,-0.0002);
```

Function:

```
void attackExponential(int *data,int length,double amp,double delta);
```

Like *attackLinear* only the ramp is exponential. Sample i is scaled as follows:

```
i = i * amp * pow(delta,i);
```

Here are two typical calls:

```
attackExponential(data,length,0.5,0.100075);

attackExponential(data,length,1.5,0.99995);
```

Function:

```
void diminishLinear(int *data,int length,int offset,double delta);

void diminishExponential(int *data,int length,int offset,double factor);
```

Like the attack filters, but works on the end of the data rather than the beginning. Sample i is updated thusly:

```
i = i * (1 + delta * i);  //linear

i = i * pow(factor,i);    //exponential
```

Function:

```
distort1(int *data,int length,int cutoff);

distort2(int *data,int length,int cutoff);

distort3(int *data,int length,int cutoff,double level);
```

Three filters for adding distortion.