FeelFine

December 9, 2021

1 I Feel Fine, Digitally

This notebook is a little musing inspired by the "internal mechanics" of a very famous audio snippet, namely the guitar sound at the beginning of the song "I Feel Fine", written and recorded by the Beatles in 1964. The historical significance of the record lies in the following claim by John Lennon:

> *"I defy anybody to find a record... unless it is some old blues record from 1922... that uses feedback that way. So I claim it for the Beatles. Before Hendrix, before The Who, before anybody. The first feedback on record."*

In this notebook, we are going to look in detail at this famous first instance of recorded guitar feedback and we will try to set up a digital model of what went down in the recording studio on that fateful 18 October 1964. In doing so we will look at a guitar simulator, at an amp model and at the mechanics of feedback. But, before anything else, let's listen to what this is all about:

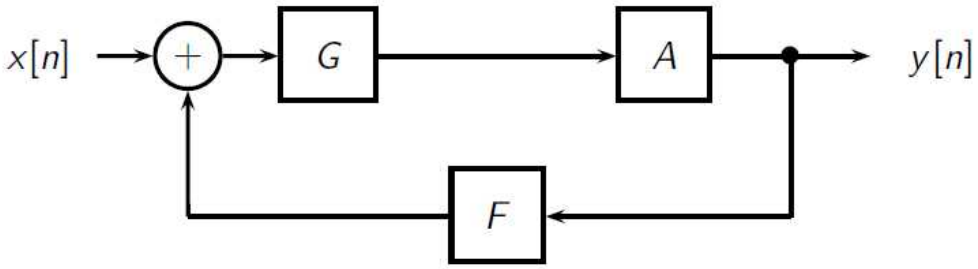
```
In [1]: %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt
        import numpy as np
        import IPython
        from scipy.io import wavfile
        from IPython.display import Image

In [2]: plt.rcParams['figure.figsize'] = 14, 4

In [5]: display(Image(filename='beatles.jpg', width=400))

        # fs will be the global "clock" of all discrete-time systems in this notebook
        fs, data = wavfile.read("iff.wav")
        # bring the 16bit wav samples into the [-1, 1] range
        data = data / 32767.0
        IPython.display.Audio(data=data, rate=fs, embed=True)
```

Out[5]: <IPython.lib.display.Audio object>

According to recording studio accounts, the sound was obtained by first playing the A string on Lennon's Gibson semiacoustic guitar and then by placing the guitar close to the amplifier. Indeed, the first two seconds of the clip sound like a standard decaying guitar tone; after that the feedback kicks in. The feedback, sometimes described as an "electric razor" buzz, is caused by two phenomena: the sound generated by the amplifier "hits" the A string and increases its vibration, and the resulting increased signal drives the amplifier into saturation.

Schematically, these are the systems involved in the generation of the opening of the song:

In [7]: display(Image(filename='bd.jpg', width=600))



In order to simulate this setup digitally, we need to come up with resonable models for:

- the guitar G, including the possibility of driving the string vibration during oscillation
- the amplifier A, including a saturating nonlinearity
- the feedback channel F, which will depend on the distance from the guitar to the amplifier

Let's examine each component in more detail.

1.1 1 - simulating a guitar

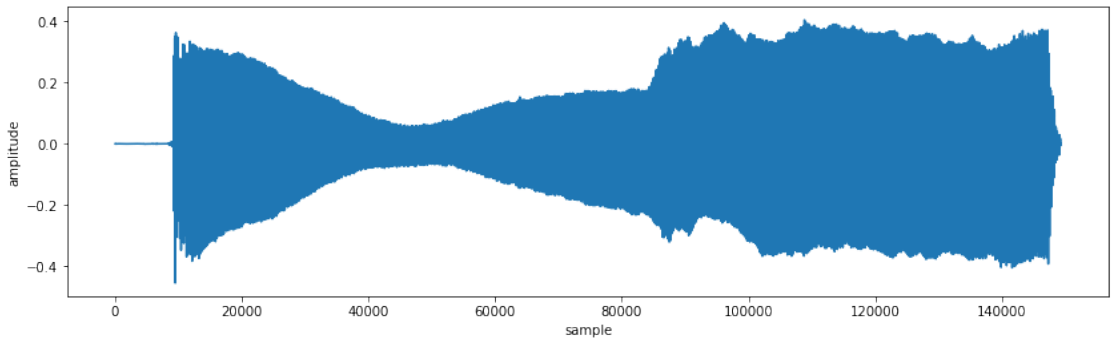
Although we have already studied the Karplus-Strong algorithm as an effective way to simulate a plucked sound, in this case we need a model that is closer to the actual physics of a guitar, since we'll need to drive the string oscillation in the feedback loop.

In a guitar, the sound is generated by the oscillation of strings that are both under tension and fixed at both ends. Under these conditions, a displacement of the string from its rest position (i.e. the initial "plucking") will result in an oscillatory behavior in which the energy imparted by the plucking travels back and forth between the ends of the string in the form of standing waves. The natural modes of oscillation of a string are all multiples of the string's fundamental frequency, which is determined by its length, its mass and its tension (see, for instance, here for a detailed explanation). This image (courtesy of Wikipedia) shows a few oscillation modes on a string:

These vibrations are propagated to the body of an acoustic guitar and converted into sound pressure waves or, for an electric guitar, they are converted into an electrical waveform by the guitar's pickups.

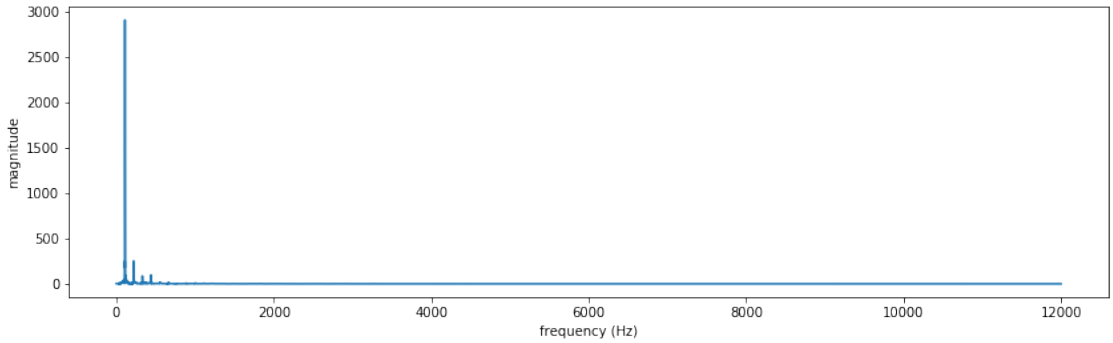
We can appreciate this behavior in the initial (non feedback) portion of the "I Feel Fine" sound snippet; let's first look at the waveform in the time domain:

```
In [8]: plt.plot(data);
        plt.xlabel("sample");
        plt.ylabel("amplitude");
```



The "pure guitar" part is approximately from sample 10000 to sample 40000. If we plot the spectrum of this portion:

```
In [9]: s = abs(np.fft.fftpack.fft(data[10000:40000]));
        s = s[0:int(len(s)/2)]
        plt.plot(np.linspace(0,1,len(s))*(fs/2), s);
        plt.xlabel("frequency (Hz)");
        plt.ylabel("magnitude");
```



Indeed we can see that the frequency content of the sound contains multiples of a fundamental frequency at 110Hz, which corresponds to the open A string on a standard-tuning guitar.

From a signal processing point of view, the guitar string acts as a resonator resonating at several multiples of a fundamental frequency; this fundamental frequency determines the *pitch* of the played note. In the digital domain, we know we can implement a resonator at a single frequency ω_0 with a second-order IIR of the form

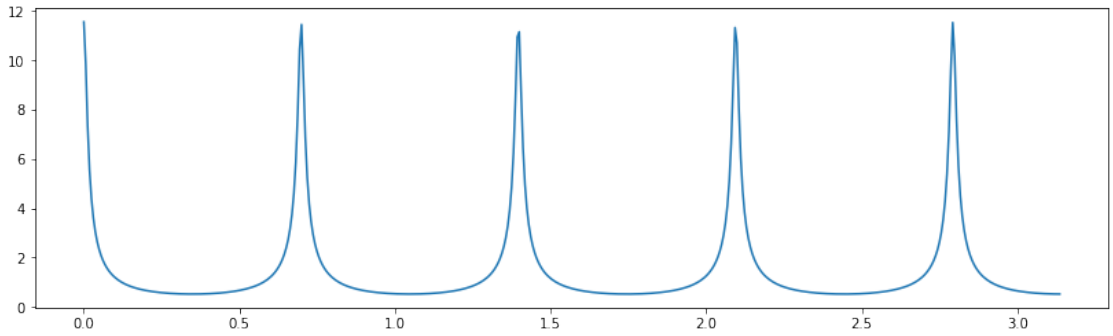
$$H(z) = \frac{1}{(1 - \rho e^{j\omega_0} z^{-1})(1 - \rho e^{-j\omega_0} z^{-1})}, \quad \rho \approx 1$$

i.e. by placing a pair of complex-conjugate poles close to the unit circle at an angle $\pm\omega_0$. A simple extension of this concept, which places poles at *all* multiples of a fundamental frequency, is the **comb filter**. A comb filter of order N has the transfer function

$$H(z) = \frac{1 - \rho z^{-1}}{1 - \rho^N z^{-N}}$$

It is easy to see that the poles of the filters are at $z_k = \rho e^{j\frac{2\pi}{N}k}$, except for $k = 0$ where the zero cancels the pole. For example, here is the frequency response of $H(z) = 1/(1 - (0.99)^N z^{-N})$ for $N = 9$:

```
In [10]: from scipy import signal

         w, h = signal.freqz(1, [1, 0, 0, 0, 0, 0, 0, 0, 0, -.99**9])
         plt.plot(w, abs(h));
```



An added advantage of the comb filter is that it is very easy to implement, since it requires only two multiplication per output sample *independently* of N:

$$y[n] = \rho^N y[n-N] + x[n] - \rho x[n-1]$$

With this, here's an idea for a guitar simulation: the string behavior is captured by a comb filter where N is given by the period (in samples) of the desired fundamental frequency. Let's try it out:

```
In [11]: class guitar:
             def __init__(self, pitch=110, fs=24000):
                 # init the class with desired pitch and underlying sampling frequency
                 self.M = int(np.round(fs / pitch)) # fundamental period in samples
                 self.R = 0.9999                    # decay factor
                 self.RM = self.R ** self.M
                 self.ybuf = np.zeros(self.M)  # output buffer (circular)
                 self.iy = 0                   # index into out buf
                 self.xbuf = 0                 # input buffer (just one sample)

             def play(self, x):
                 y = np.zeros(len(x))
                 for n in range(len(x)):
                     t = x[n] - self.R * self.xbuf + self.RM * self.ybuf[self.iy]
                     self.ybuf[self.iy] = t
                     self.iy = (self.iy + 1) % self.M
                     self.xbuf = x[n]
                     y[n] = t
                 return y
```

Now we model the string plucking as a simple impulse signal in zero and we input that to the guitar model:

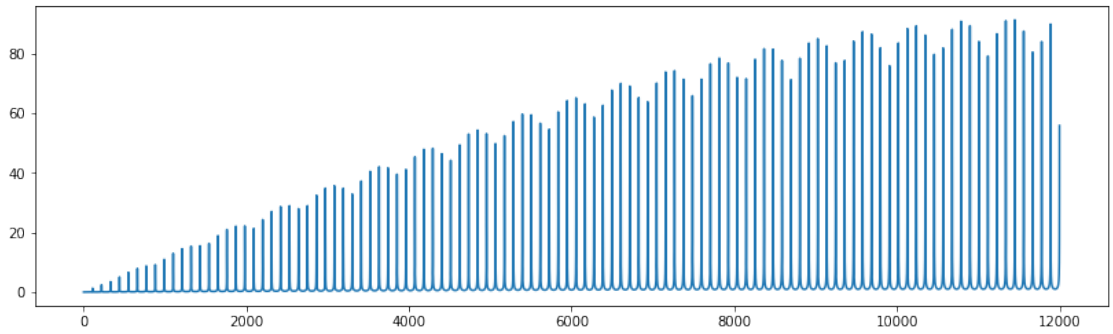
```
In [12]: # create a 2-second signal
         d = np.zeros(fs*2)
         # impulse in zero (string plucked)
         d[0] = 1
```

```
        # create the A string
        y = guitar(110, fs).play(d)
        IPython.display.Audio(data=y, rate=fs, embed=True)
```

```
Out[12]: <IPython.lib.display.Audio object>
```

Ouch! The pitch may be right but the timbre is grotesque! The reason becomes self-evident if we look at the frequency content:

```
In [13]: s = abs(np.fft.fftpack.fft(y));
         s = s[0:int(len(s)/2)]
         plt.plot(np.linspace(0,1,len(s))*(fs/2), s);
```



Although we have multiples of the fundamental, we actually have *too many* spectral lines and, because of the zero in the filter, a highpass characteristic. In a real-world guitar both the stiffness of the string and the response of the guitar's body would limit the number of harmonics to just a few, as we saw in the figure above where we analyzed the snippet from the song.

Well, it's not too hard to get rid of unwanted spectral content: just add a lowpass filter. In this case we use a simple Butterworth that keeps only the first five harmonics:

```
In [14]: from scipy import signal

         class guitar:
             def __init__(self, pitch=110, fs=24000):
                 # init the class with desired pitch and underlying sampling frequency
                 self.M = int(np.round(fs / pitch) )# fundamental period in samples
                 self.R = 0.9999                # decay factor
                 self.RM = self.R ** self.M
                 self.ybuf = np.zeros(self.M)  # output buffer (circular)
                 self.iy = 0                    # index into out buf
                 self.xbuf = 0                  # input buffer (just one sample)
                 # 6th-order Butterworth, keep 5 harmonics:
                 self.bfb, self.bfa = signal.butter(6, min(0.5, 5.0 * pitch / fs))
                 self.bfb *= 1000               # set a little gain
                 # initial conditions for the filter. We need this because we need to
```

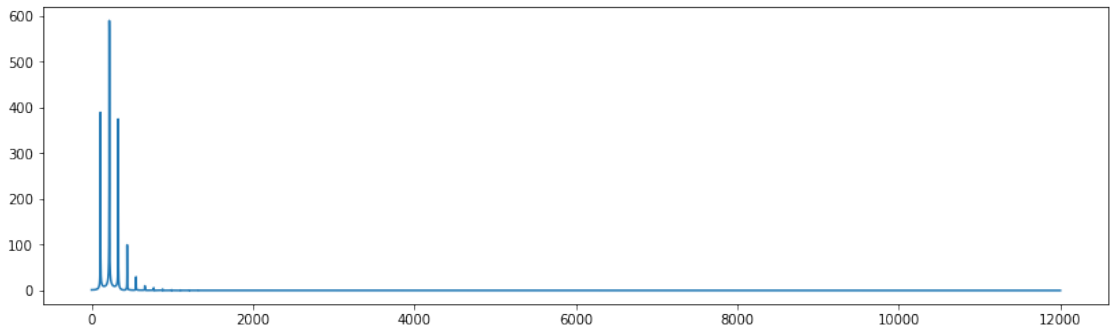
```
            # filter on a sample-by-sample basis later on
            self.bfs = signal.lfiltic(self.bfb, self.bfa, [0])

        def play(self, x):
            y = np.zeros(len(x))
            for n in range(len(x)):
                # comb filter
                t = x[n] - self.R * self.xbuf + self.RM * self.ybuf[self.iy]
                self.ybuf[self.iy] = t
                self.iy = (self.iy + 1) % self.M
                self.xbuf = x[n]
                # lowpass filter, keep filter status for next sample
                y[n], self.bfs = signal.lfilter(self.bfb, self.bfa, [t], zi=self.bfs)
            return y
```

OK, let's give it a spin:

```
In [15]: y = guitar(110, fs).play(d)
         IPython.display.Audio(data=y, rate=fs, embed=True)

Out[15]: <IPython.lib.display.Audio object>
```

Ah, so much better, no? Almost like the real thing. We can check the spectrum and indeed we're close to what we wanted; the guitar is in the bag.

```
In [16]: s = abs(np.fft.fftpack.fft(y[10000:30000]));
         s = s[0:int(len(s)/2)]
         plt.plot(np.linspace(0,1,len(s))*(fs/2), s);
```



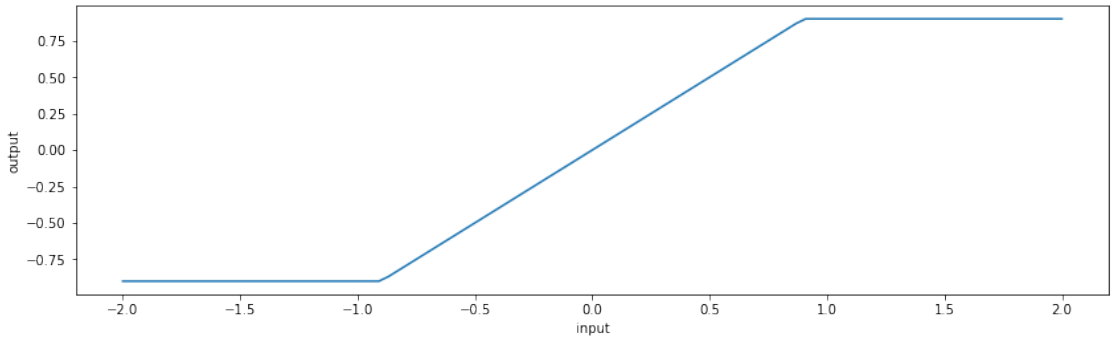
1.2 2 - the amplifier

In the "I Feel Fine" setup, the volume of the amplifier remains constant; however, because of the feedback, the input will keep increasing and, at one point or another, any real-world amplifier will be driven into saturation. When that happens, the output is no longer a scaled version of the input but gets "clipped" to the maximum output level allowed by the amp. We can easily simulate this behavior with a simple memoryless clipping operator:

```
In [17]: def amplify(x):
             TH = 0.9              # threshold
             y = np.copy(x)
             y[y >  TH] =  TH
             y[y < -TH] = -TH
             return y
```

We can easily check the characteristic of the amplifier simulator:

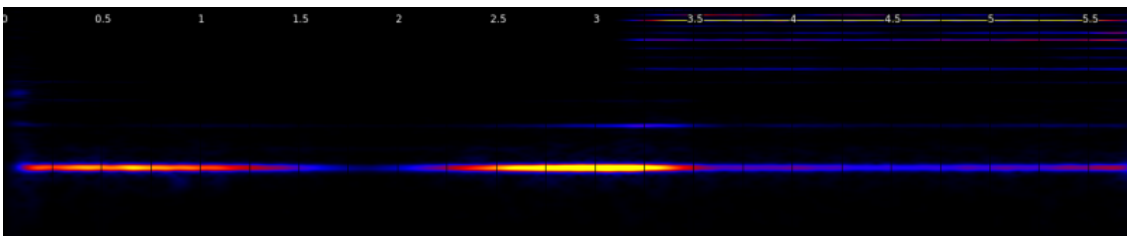
```
In [18]: x = np.linspace(-2, 2, 100)
         plt.plot(x, amplify(x));
         plt.xlabel("input");
         plt.ylabel("output");
```



While the response is linear between -TH and TH, it is important to remark that the clipping introduces a nonlinearity in the processing chain. In the case of linear systems, sinusoids are eigenfunctions and therefore a linear system can only alter a sinusoid by modifying its amplitude and phase. This is not the case with nonlinear systems, which can profoundly alter the spectrum of a signal by creating new frequencies. While these effects are very difficult to analyze mathematically, from the acoustic point of view nonlinear distortion can be very interesting, and "I Feel Fine" is just one example amongst countless others.

It is instructive at this point to look at the spectrogram (i.e. the STFT) of the sound sample (figure obtained with a commercial audio spectrum analyzer); note how, indeed, the spectral content shows many more spectral lines after the nonlinearity of the amplifier comes into play.

```
In [21]: display(Image(filename='specgram.png', width=800))
```



1.3 3 - the acoustic feedback

The last piece of the processing chain is the acoustic channel that closes the feedback loop. The sound pressure waves generated by the loudspeaker of the amplifier travel through the air and eventually reach the vibrating string. For feedback to kick in, two things must happen:

- the energy transfer from the pressure wave to the vibrating string should be non-negligible
- the phase of the vibrating string must be sufficiently aligned with the phase of the sound wave in order for the sound wave to "feed" the vibration.

Sound travels in the air at about 340 meters per second and sound pressure decays with the reciprocal of the traveled distance. We can build an elementary acoustic channel simulation by neglecting everything except delay and attenuation. The output of the acoustic channel for a guitar-amplifier distance of d meters will be therefore

$$y[n] = \alpha x[n - M]$$

where $\alpha = 1/d$ and M is the propagation delay in samples; with an internal clock of F_s Hz we have $M = \lfloor d/(cF_s) \rfloor$ where c is the speed of sound.

```
In [22]: class feedback:
             SPEED_OF_SOUND = 343.0 # m/s
             def __init__(self, max_distance_m = 5, fs=24000):
                 # init class with maximum distance
                 self.L = int(np.ceil(max_distance_m / self.SPEED_OF_SOUND * fs));
                 self.xbuf = np.zeros(self.L)      # circular buffer
                 self.ix = 0

             def get(self, x, distance):
                 d = int(np.ceil(distance / self.SPEED_OF_SOUND * fs))    # delay in samples
                 self.xbuf[self.ix] = x
                 x = self.xbuf[(self.L + self.ix - d) % self.L]
                 self.ix = (self.ix + 1) % self.L
                 return x / float(distance)
```

1.4 4 - play it, Johnny

OK, we're ready to play. We will generate a few seconds of sound, one sample at a time, following these steps:

- generate a guitar sample
- process it with the nonlinear amplifier
- feed it back to the guitar via the acoustic channel using a time-varying distance

During the simulation, we will change the distance used in the feedback channel model to account for the fact that the guitar is first played at a distance from the amplifier, and then it is placed very close to it. In the first phase, the sound will simply be a decaying note and then the feedback will start moving the string back in full swing and drive the amp into saturation. We also need to introduce some coupling loss between the sound pressure waves emitted by the loudspeaker and the string, since air and wound steel have rather different impedences.

Let's see if that works:

In [23]:
```
g = guitar(110)     # the A string
f = feedback()      # the feedback channel

# the "coupling loss" between air and string is high. Let's say that
#  it is about 80dBs
COUPLING_LOSS = 0.0001

# John starts 3m away and then places the guitar basically against the amp
#  after 1.5 seconds
START_DISTANCE = 3
END_DISTANCE = 0.05

N = int(fs * 5)          # play for 5 seconds
y = np.zeros(N)
x = [1]             # the initial plucking
# now we create each sample in a loop by processing the guitar sound
# thru the amp and then feeding back the attenuated and delayed sound
# to the guitar
for n in range(N):
    y[n] = amplify(g.play(x))
    x = [COUPLING_LOSS * f.get(y[n], START_DISTANCE if n < (1.5 * fs) else END_DISTANCE


IPython.display.Audio(data=y, rate=fs, embed=True)
```

Out[23]:
```
<IPython.lib.display.Audio object>
```

Pretty close, no? Of course the sound is not as rich as the original recording since

- real guitars and real amplifiers are very complex physical system with many more types of nonlinearities; amongst others:
- the spectral content generated by the string varies with the amplitude of its oscillation
- the spectrum of the generated sound is not perfectly harmonic due to the physical size of the string
- the string may start touching the frets when driven into large oscillations
- the loudspeaker may introduce additional frequencies if driven too hard
- ...
- we have neglected the full frequency response of the amp both in linear and in nonlinear mode
- it's the BEATLES, man! How can DSP compete?

Well, hope this was a fun and instructive foray into music and signal processing. You can now play with the parameters of the simulation and try to find alternative setups:

- try to change the characteristic of the amp, maybe using a sigmoid (hyperbolic tangent)
- change the gain, the coupling loss or the frequency of the guitar
- change John's guitar's position and verify that feedback does not occur at all distances.

In [ ]: