Encoding data in dubstep drops

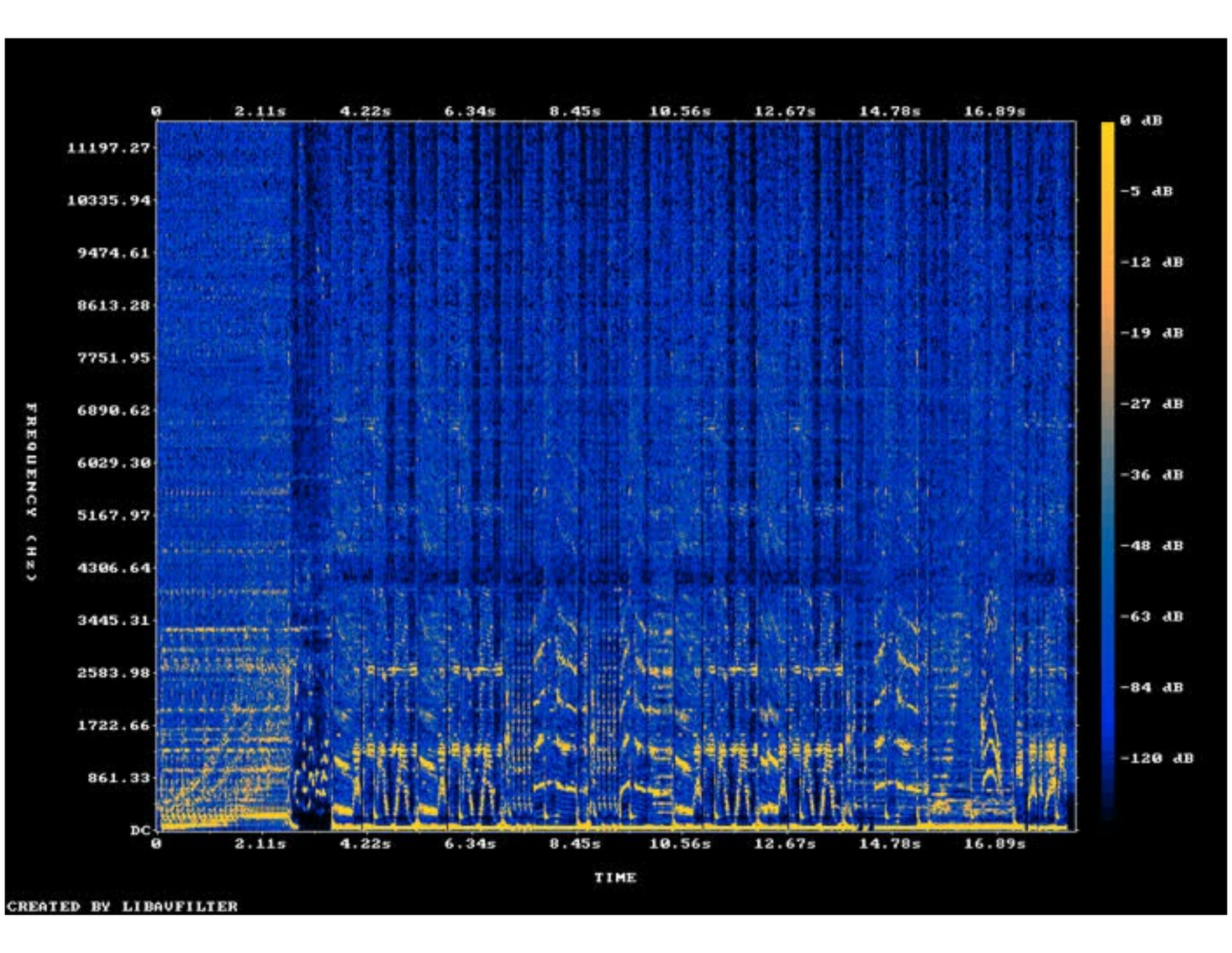
[Warning: Those who can't stand EDM/dubstep, oh boy do I have bad news for you in regards to this blog post]

Dubstep songs are often criticized as sound extremely computer generated and often just too aggressive/"digital" for a lot of people to enjoy. It's not uncommon for people to joke that they sound like someone had added a bassline and drums to modem noises

For some tracks this is truer than others. After all, it's a genre with more aggressive interpretations and more relaxed ones.

But that had me thinking, how much effort would it be to actually embed machine readable data inside a dubstep track, while ensuring that the sound could be enjoyed by humans as well…

Let's take a track to work with, this is the buildup and drop of `Skrillex - Right In`:



No HTML5? Come on!

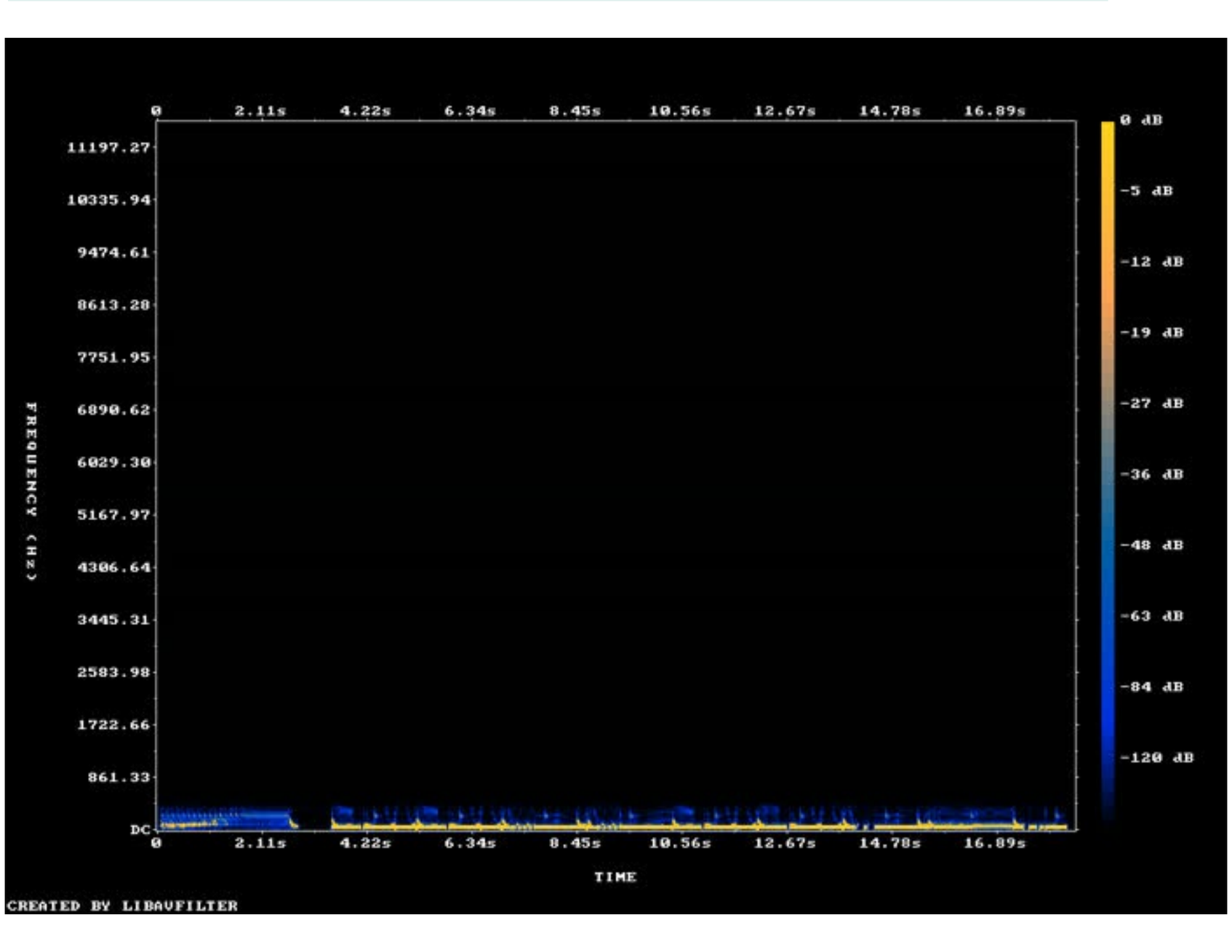
If can you tolerate listening to that, Good news! You will likely be fine with everything below.

Above here is a spectrogram of the song that was embedded, so you have a better idea of what is going on with all the frequency bands in the song.

As shown, there is plenty of spectrum to hide data in, but it exists on the high bands where audio compression may remove it. So, what if we looked lower in the frequency bands, at the bassline?

Here is what 0-100hz sounds like (you have a better chance listening to this on head/ear phones):

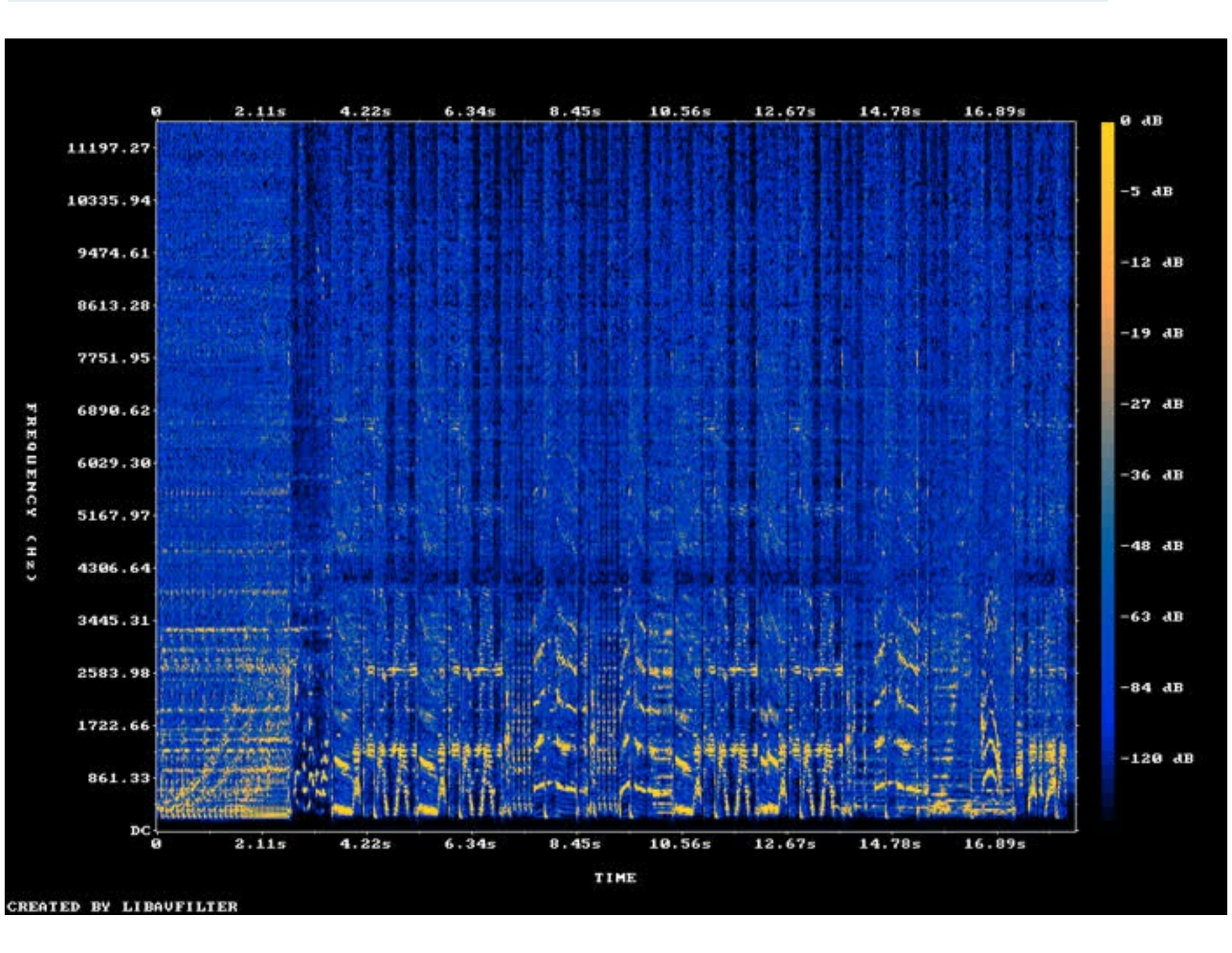
```
$ sox orig.wav onlylower.wav sinc 0k-0.1k
```



No HTML5? Come on!

If we go back to the original and remove the 100hz bands and below, it sounds like:

```
$ sox orig.wav onlyhigher.wav sinc -b 0 0.2k-22k
$ cp onlyhigher.wav tmp.wav
$ # Do a 2nd round to ensure the lower band is really empty
$ sox tmp.wav onlyhigher.wav sinc -b 0 0.2k-22k
$ rm tmp.wav
```

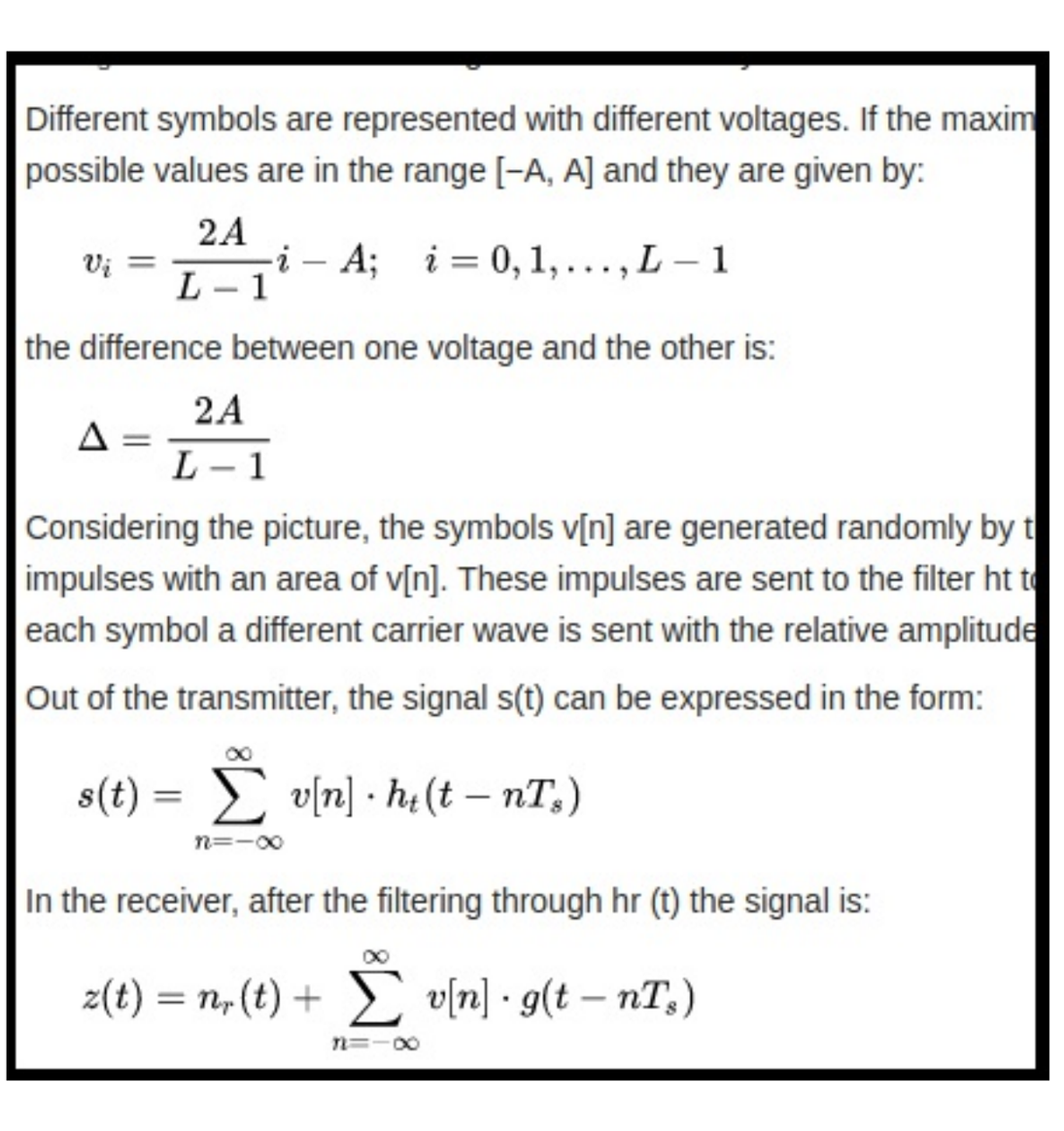


Note: If you turned up volume to hear the last one, you will want to revert that now.

No HTML5? Come on!

ASK Modulation for non maths people

If you are like me and have a pretty poor maths background then every time you see things like this:


Different symbols are represented with different voltages. If the maxim possible values are in the range [−A, A] and they are given by:

$$v_i = \frac{2A}{L-1}i - A; \quad i = 0, 1, \ldots, L-1$$

the difference between one voltage and the other is:

$$\Delta = \frac{2A}{L-1}$$

Considering the picture, the symbols v[n] are generated randomly by t impulses with an area of v[n]. These impulses are sent to the filter ht t each symbol a different carrier wave is sent with the relative amplitude

Out of the transmitter, the signal s(t) can be expressed in the form:

$$s(t) = \sum_{n=-\infty}^{\infty} v[n] \cdot h_t(t - nT_s)$$

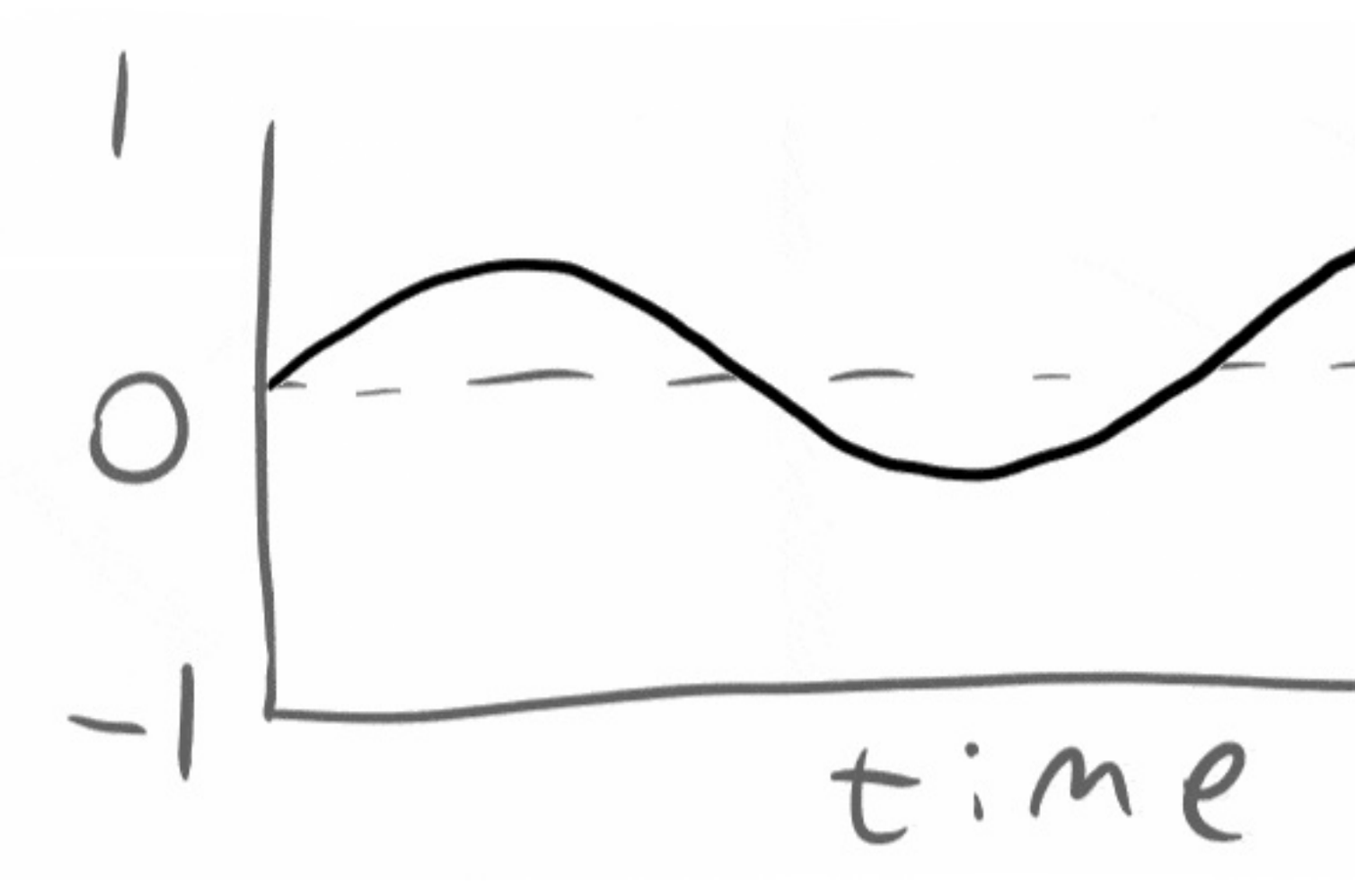
In the receiver, after the filtering through hr (t) the signal is:

$$z(t) = n_r(t) + \sum_{n=-\infty}^{\infty} v[n] \cdot g(t - nT_s)$$

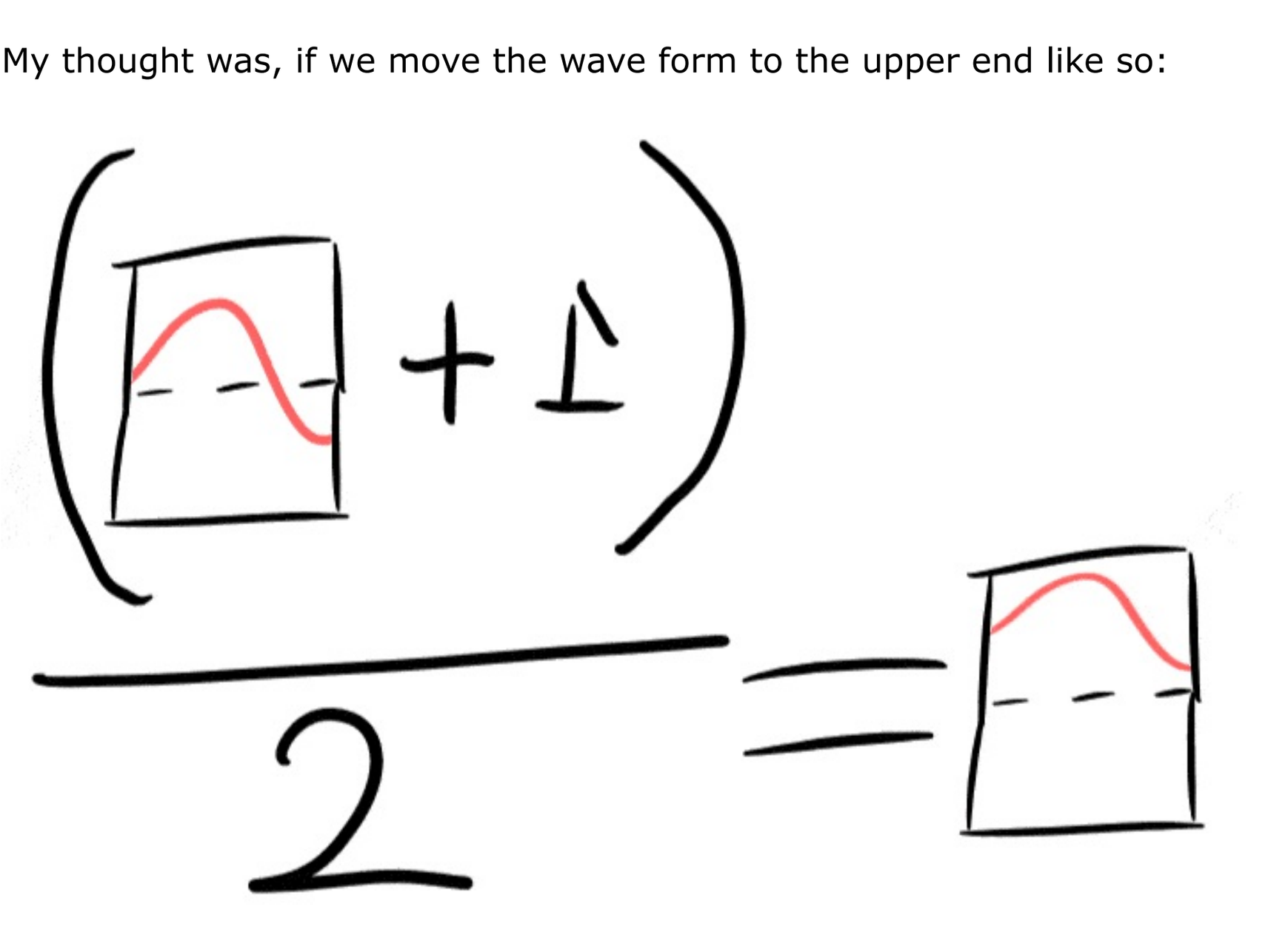
You suddenly become a lot less interested in looking-at-the-thing.

Unfortunately, this is pretty much entirely the DSP world. However given that I struggled through this, I will try and describe what I'm doing without any fancy symbols.

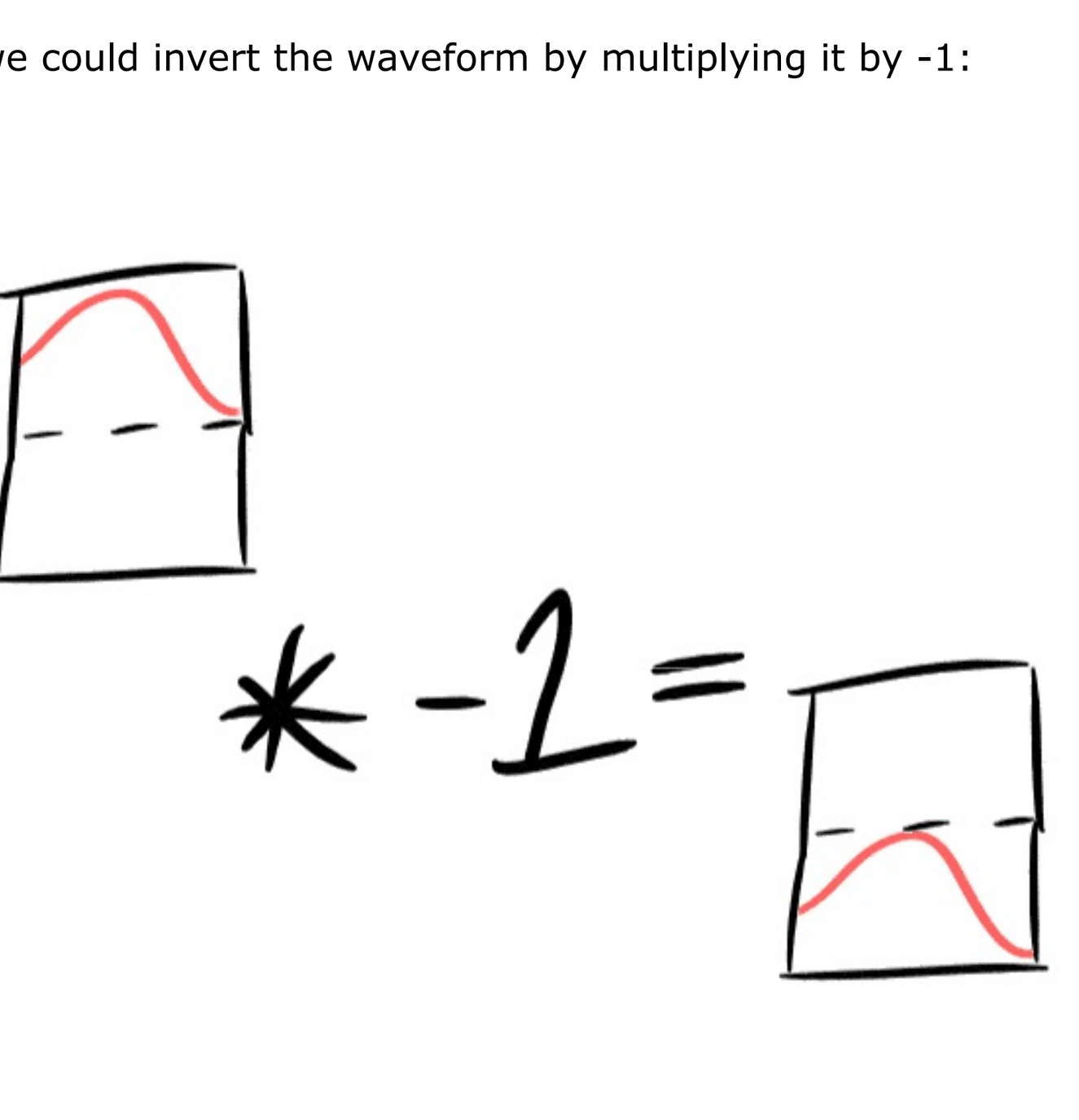
To start, this is basically sound in its most basic form; a signal to a speaker telling it what position it should be in. Ranging from 1 (all the way out) to -1 (all the way in). Moving over these positions displaces air and makes sound!



My thought was, if we move the wave form to the upper end like so:



Then we could invert the waveform by multiplying it by -1:



This means we now have two states we can observe! The best part of this is that the difference is not noticeable to (at least my) human ears.

This base code is simple and switches between 1 and 0:

```
package main

import (
    "encoding/binary"
    "flag"
    "log"
    "os"
)

func main() {
    ins := flag.String("input", "./in.f64.data", "")
    outs := flag.String("out", "./out.f64.data", "")
    flag.Parse()

    inf, err := os.OpenFile(*ins, os.O_RDONLY, 0644)

    if err != nil {
        log.Fatalf("Unable to open output file %s", err.Error())
    }

    outf, err := os.OpenFile(*outs, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0644)
    if err != nil {
        log.Fatalf("Unable to open output file %s", err.Error())
    }

    Symbolrate := 11000
    SamplesUntilChange := Symbolrate
    UpperFlip := false
    bits := 0
    for {
        var raws float64
        err := binary.Read(inf, binary.LittleEndian, &raws)
        if err != nil {
            log.Printf("Leaving %s", err.Error())
            break
        }

        // First, obtain a upper flip
        normie := (raws + 1) / 2

        SamplesUntilChange--
        if SamplesUntilChange == 0 {
            UpperFlip = !UpperFlip
            SamplesUntilChange = Symbolrate
            bits++
        }

        if !UpperFlip {
            normie = normie * -1
        }

        if SamplesUntilChange < 1000 {
            dest := normie * -1
            normie = lerp(dest, normie, float64(SamplesUntilChange)/1000.0)
        }

        binary.Write(outf, binary.LittleEndian, normie)
    }
    log.Printf("Finished with %d bits / %d bytes", bits, bits/8)
}

func lerp(a, b, n float64) float64 {
    return (1-n)*a + n*b
}
```

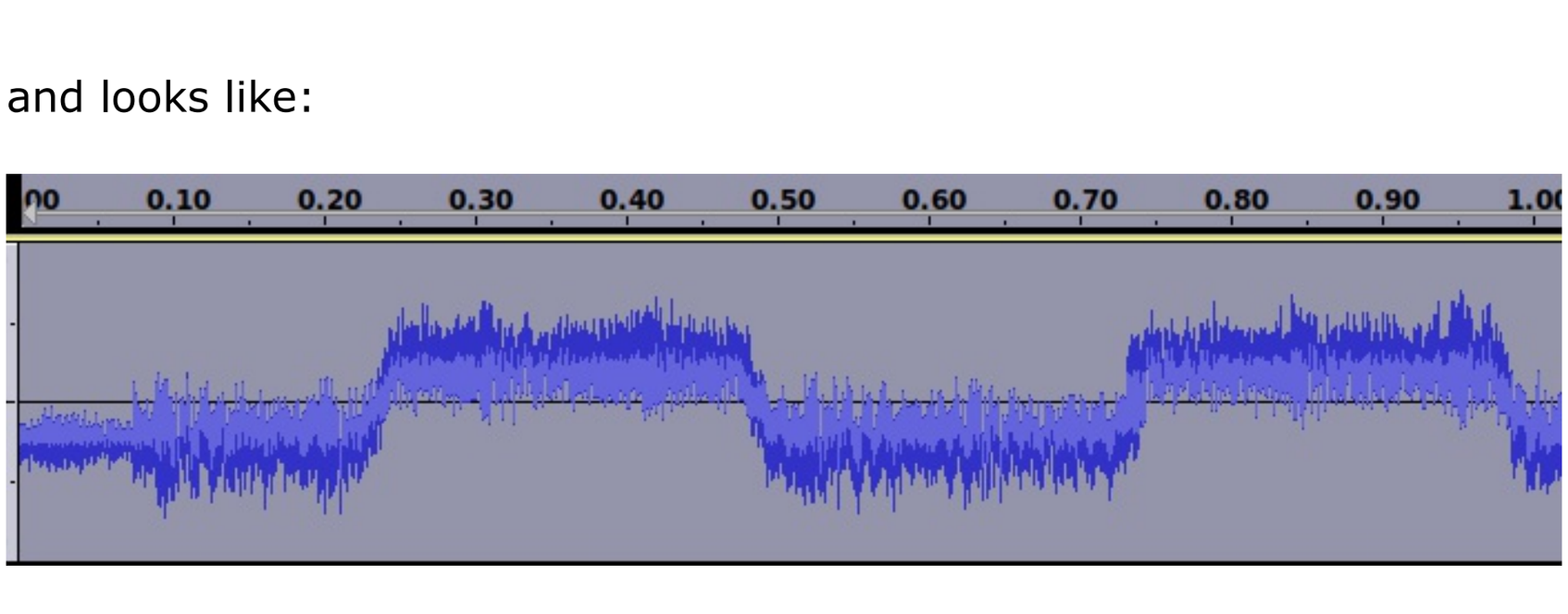
At this point we can reassemble the output bassline file and the upper frequencies, and see if it is playable:

```
$ sox orig.wav onlylower.wav sinc 0k-0.1k
$ sox orig.wav onlyhigher.wav sinc 0.1k-22k
$ ffmpeg -i onlylower.wav -f f64le -ar 44100 -ac 1 -y in.f64.data
$ ./ASK-dubstep -input in.f64.data
$ ffmpeg -f f64le -ar 44100 -ac 1 -i out.f64.data -y encoded-bass.wav
$ sox -m encoded-bass.wav onlyhigher.wav encoded-bassline.wav
```

It results in something that sounds like:

No HTML5? Come on!

and looks like:



I might be wrong, but as far as I can tell, this is a form of Amplitude-shift keying.

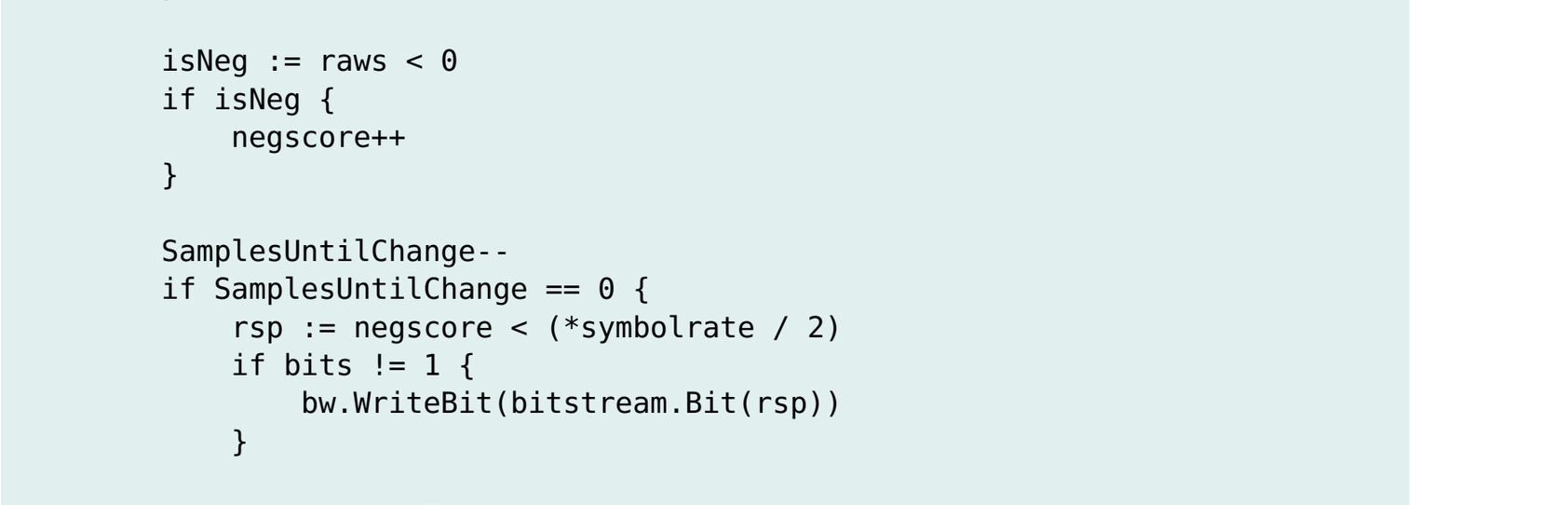
To make it encode our own data, we can use a simple library to help us read bits in a nice easy to use way, and then include them in the audio:

```
package main

import (
    "encoding/binary"
    "flag"
    "log"
    "os"
    "strings"

    "github.com/dgryski/go-bitstream"
)

func main() {
    ins := flag.String("input", "./in.f64.data", "")
    outs := flag.String("out", "./out.f64.data", "")
    encodetarget := flag.String("data", "Hello World!", "")
    flag.Parse()

    inf, err := os.OpenFile(*ins, os.O_RDONLY, 0644)

    if err != nil {
        log.Fatalf("Unable to open output file %s", err.Error())
    }

    outf, err := os.OpenFile(*outs, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0644)
    if err != nil {
        log.Fatalf("Unable to open output file %s", err.Error())
    }

    sr := strings.NewReader(*encodetarget)
    bitreader := bitstream.NewReader(sr)

    Symbolrate := 5500
    SamplesUntilChange := Symbolrate
    UpperFlip := false
    bits := 0
    nextbit := false

    for {
        var raws float64
        err := binary.Read(inf, binary.LittleEndian, &raws)
        if err != nil {
            log.Printf("Leaving %s", err.Error())
            break
        }

        // First, obtain a upper flip
        normie := (raws + 1) / 2

        SamplesUntilChange--
        if SamplesUntilChange == 0 {
            UpperFlip = nextbit
            b, _ := bitreader.ReadBit()
            nextbit = bool(b)

            SamplesUntilChange = Symbolrate
            bits++
        }

        if !UpperFlip {
            normie = normie * -1
        }

        if SamplesUntilChange < 1000 && UpperFlip != nextbit {
            dest := normie * -1
            normie = lerp(dest, normie, float64(SamplesUntilChange)/1000.0)
        }

        binary.Write(outf, binary.LittleEndian, normie)
    }
    log.Printf("Finished with %d bits / %d bytes", bits, bits/8)
}

func lerp(a, b, n float64) float64 {
    return (1-n)*a + n*b
}
```

The decoder looks like this:

```
package main

import (
    "encoding/binary"
    "flag"
    "fmt"
    "log"
    "os"

    bitstream "github.com/dgryski/go-bitstream"
)

func main() {
    ins := flag.String("input", "./in.f64.data", "")
    symbolrate := flag.Int("srate", 5500, "Symbol rate in samples")
    flag.Parse()

    inf, err := os.OpenFile(*ins, os.O_RDONLY, 0644)

    if err != nil {
        log.Fatalf("Unable to open output file %s", err.Error())
    }

    bw := bitstream.NewWriter(os.Stdout)
    bw.WriteBit(bitstream.Bit(false))

    SamplesUntilChange := *symbolrate
    bits := 1
    negscore := 0

    for {
        var raws float64
        err := binary.Read(inf, binary.LittleEndian, &raws)
        if err != nil {
            fmt.Print("\n")
            log.Printf("Leaving %s", err.Error())
            break
        }

        isNeg := raws < 0
        if isNeg {
            negscore++
        }

        SamplesUntilChange--
        if SamplesUntilChange == 0 {
            rsp := negscore < (*symbolrate / 2)
            if bits != 1 {
                bw.WriteBit(bitstream.Bit(rsp))
            }

            negscore = 0
            SamplesUntilChange = *symbolrate
            bits++
        }

    }
    log.Printf("Finished with %d bits / %d bytes", bits, bits/8)
}
```

This decoder works by counting how many samples in a "frame" are negative, and based on that declaring if it's a 1 or a 0 bit.



It is then reassembled back into bytes, and output to the terminal.

To finish off, here is a live demo of it all working with a different song (`Smooth - Nowhere`):

I have pushed the code on github as always (though it's not really in a usable state): https://github.com/benjojo/dubstep-data

And if you enjoyed this, you will be glad to know that I am going to be at Recurse Center in NY for the next 10 weeks! Meaning you can follow my Twitter or RSS to keep up with the other silly things I will do!