DSP Prototyping with Python

Jorge Garcia, MSc
Independent

ADC'18 Audio Developer Conference

Abstract

This poster introduces the prototyping process of Digital Signal Processing (DSP) algorithms.

An initial background and general information of the advantages and caveats in building effective prototypes for audio applications is presented. A variety of audio languages and frameworks are available as useful tools that can be leveraged in production projects, depending on the platform and the approach of choice.

The Anaconda distribution and the Python language are showcased for designing and implementing a simple Butterworth low-pass filter (LPF). This filter is shown as an example of what kind of DSP prototypes can be achieved with a few lines of code, and before a final C++ optimised implementation takes place.

Introduction

Why prototype?
DSP development can be very time consuming. Having a process to test out, experiment and reject ideas early is convenient and can help saving time in the development of the final product. Being able to test out different alternatives quickly makes it easier to find the most suitable algorithm for the application and platform requirements.

What are the caveats?
More development time has to be invested. Prototype code quality can usually be lower and perform worse than production code. Learning a prototyping language or framework takes time, which is not directly invested in the final product.

Prototyping life cycle
1. Research available algorithms.
2. Initial prototype design and implementations.
3. Iteration and optimisation of alternatives.
4. Evaluation of the approaches (back to step 2).
5. Initial production implementation.

What are known prototyping tools/frameworks?
Matlab, Octave, Max/MSP, Pure Data...

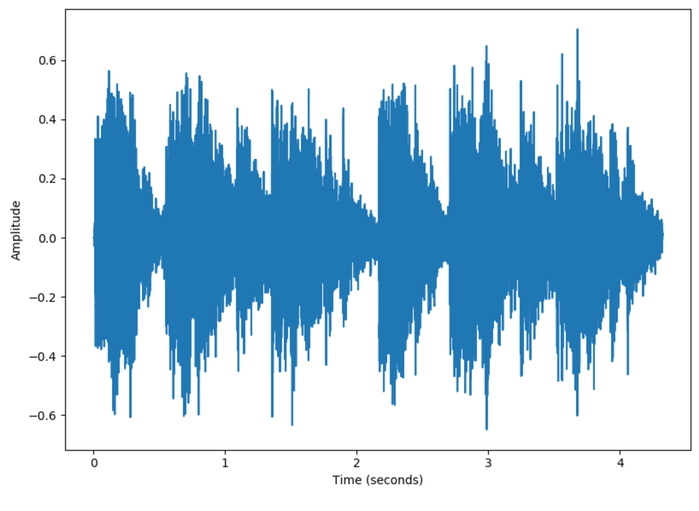
Why use Python and Anaconda for prototyping?
Python is a high level, dynamic language that provides an interactive interpreter. The code can be ported easily to other languages (e.g. C/C++). The community is welcoming and helpful. Moreover, Python has become one of the top programming languages in various rankings and surveys [1] [2]. The free and open source Anaconda distribution [3] offers a range of scientific computing libraries.

Plotting basics

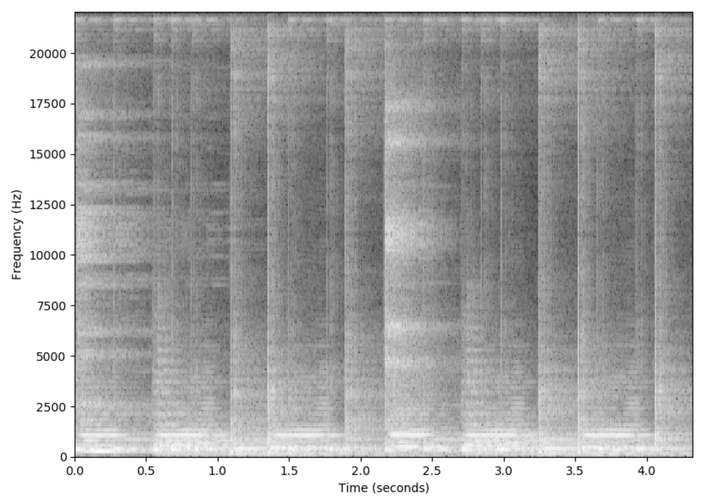
```
from scipy.io import wavfile
import matplotlib.pyplot as plt
import numpy as np

# Obtain sample rate (fs) and data from mono file
fs, data = wavfile.read("music.wav")
# Normalise 16-bit input to range -1,1
data = data / 2.0 ** 15

# Configure axes and plot
plt.axes(xlabel="Time (seconds)",
ylabel="Amplitude")
t = np.linspace(0, len(data) / float(fs), len(data))
plt.plot(t, data)
plt.show()
```




```
plt.axes(xlabel="Time (seconds)",
ylabel="Frequency (Hz)")
plt.specgram(data, NFFT=512, Fs=fs,
cmap=plt.cm.gist_gray)
plt.plot()
plt.show()
```



Quick Butterworth LPF

```
from scipy.signal import butter, lfilter

def butter_lowpass(cutoff, fs, order):
    nyquist = 0.5 * fs
    cut = cutoff / nyquist
    b, a = butter(order, cut, btype='low')
    return b, a

def butter_lowpass_filter(data, cutoff, fs, order):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = lfilter(b, a, data)
    return y
```

Low-level Butterworth LPF

```
# The following implementation follows the
# equations from [4] [5] and [6]:

from scipy.io import wavfile
import numpy as np

fs, data = wavfile.read("music.wav")
data = data / 2.0 ** 15
cutoff = 2000.0 # Cutoff Frequency in Hz

# Second-order Butterworth filter coefficients
filterLambda = 1 / np.tan(np.pi * cutoff / fs)
a0 = 1 / (1 + 2 * filterLambda + filterLambda ** 2)
a1 = 2 * a0
a2 = a0
b1 = 2 * a0 * (1 - filterLambda ** 2)
b2 = a0 * (1 - 2 * filterLambda + filterLambda ** 2)

xn_1 = 0.0
xn_2 = 0.0
yn_1 = 0.0
yn_2 = 0.0

y = np.zeros(len(data))

for n in range (0, len(data)):
    y[n] = a0*data[n] + a1 * xn_1 + a2 * xn_2
            - b1 * yn_1 - b2 * yn_2
    xn_2 = xn_1
    xn_1 = data[n]
    yn_2 = yn_1
    yn_1 = y[n]

wavfile.write("filtered_output.wav", fs, y)
```

Conclusions

1. Effective and quick prototypes can be achieved with a few lines of Python code.
2. Prototyping with the Python language makes the development process more flexible and portable to other languages and frameworks.
3. High-level Python frameworks and distributions like Anaconda can be leveraged for applying and developing different DSP algorithms.
4. Some time and resources can be saved in the early stages of development after an initial investment in learning prototyping frameworks.
5. Prototyping leads to exploring the design options more creatively and quickly.

About the author

Jorge has been working in the audio field for more than a decade under different roles in broadcasting, professional audio, music, and more recently, games. He has participated in the programming and engineering of video game franchises like Spacelords, Guitar Hero, DiRT and FIFA. Additionally, he has worked as a developer for MIDAS consoles and Behringer in the past. His passions, interests and skill set span from R&D to DSP, tools, and audio engine design and development.

Contact

@JGarciaMartin

info@jorgegarciamartin.com

www.linkedin.com/in/jorgegarciamartin

References

1. https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages
2. https://insights.stackoverflow.com/survey/2018
3. http://www.anaconda.com
4. *Think DSP: Digital Signal Processing in Python*. Allen B. Downey. Green Tea Press (2014) http://greenteapress.com/wp/think-dsp/
5. *The Audio Programming Book*. V. Lazzarini. MIT Press (2011)
6. *Introduction to DSP Prototyping* chapter in *"Game Audio Programming Volume 2"*. G. Somberg, J.Garcia. CRC Press, Taylor & Francis (2018)