QuDSP: A Quick DSP Application Framework

Kyle Kastner and Johanna Hansen

***Abstract*—DSP applications require extensive debugging and testing after initial development. However, most environments for DSP algorithm development and visualization lack performance for large-scale use. This framework provides basic DSP functions in performant, compiled code, while also having a dynamic interface for visualization and high-level computation, in order to bridge the gap between development and deployment.**

***Index Terms*—Framework, Dataflow, Python, C++, DSP, Numpy.**

I. Introduction

THE FIELDS of digital signal processing and software development are highly related. As computing power grows, the amount of data to process also grows larger. Algorithms become increasingly more elaborate, and the amount of time, knowledge, and effort necessary to create a useful application can be extreme.

A. Background

Work in DSP often requires both standard debugging tools and expert verification of the processing stages involved in the data pipeline. As a result, algorithm development and deployment tend to occur in two separate stages. First, the algorithm is developed in a high-level language such as MATLAB or Python, using tools for visualization and basic debugging in order to verify the operations work as expected. Next, the same application is ported to a low-level compiled language, in order to meet the performance required for most DSP applications. During this translation, many new and unforeseen bugs can occur, lengthening development time and increasing project complexity.

B. Overview

By combining performant data processing code blocks and a high-level scripting interface into a unified toolkit, it is possible to utilize the strengths of both dynamic and compiled code, reducing the difficulty and hassle of DSP application development and deployment. This paper documents the driving concerns and specific goal applications that lead the development of this toolkit. It also explores design, capabilities, example applications, and future work.

C. Goal

The goal of this framework is to provide a configurable, performant environment to foster development of new DSP algorithms. Visual debugging tools and simple system configuration are available to minimize the amount of time spent on non-development tasks such as testing, bug-fixing, and setup of development tools.

D. Key Concerns

1) Performance: Performance is a difficult topic. What works well on one set of hardware may work poorly or fail completely on another system. Design decisions made by other dataflow frameworks can also reduce performance, while gaining data safety and user friendliness. One of the key ideas behind the performance pipeline is to minimize data copying, which can greatly degrade performance of a program. On some modern processors, it can also be faster to keep a set of representative data and re-apply the transforms on the other side of a data transfer, rather than going back to non-cache memory (or worse, disk!). By choosing languages with a history of acceptance in the scientific community, minimizing redundant operations, and designing with parallel operation in mind, intricate code can remain performant enough for large-scale use.

2) Visualization: The availability of visualization tools can maximize gains for time spent during the development process. Being able to make changes to an algorithm and immediately view the results of these changes can allow an engineer to rapidly zero in problem areas of a given implementation. Visuals also provide an excellent way to communicate results and design decisions in an understandable way.

3) Flexibility: Flexibility is the trademark of many great tools. By designing a tool with maximum flexibility, individuals are able to use existing work to successfully complete tasks that the original design never considered. This trait also requires careful consideration during the design stage, as too much flexibility often hurts performance, and vice-versa.

II. Design

Design of this framework involved the concept of four separate ”stages” of processing: configuration, data acquisition, static processing, and dynamic processing. The dynamic and static processing stages are designed to intertwine, but the current implementation occurs as a linear sequence of events. A configurable option to continuously repeat the dynamic section is also shown in Figure 1.

A. Implementation

The implementation of these blocks is done using two primary languages (C++ and Python) and a host of libraries and modules. As of now, the bulk of the effort is in C++. Hopefully, applications will reuse the C++ codebase, while the Python toolkit expands for new applications.

1) C++: The configuration, data acquisition, and static processing stages are all accomplished using C++. Configuration uses libconfig, which has it’s own syntax for configuration files. These configurations files are read in when a program begins in order to configure blocks and components. Data

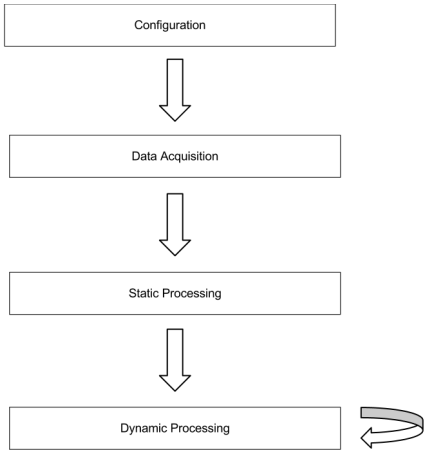

Fig. (1): System design in four stages

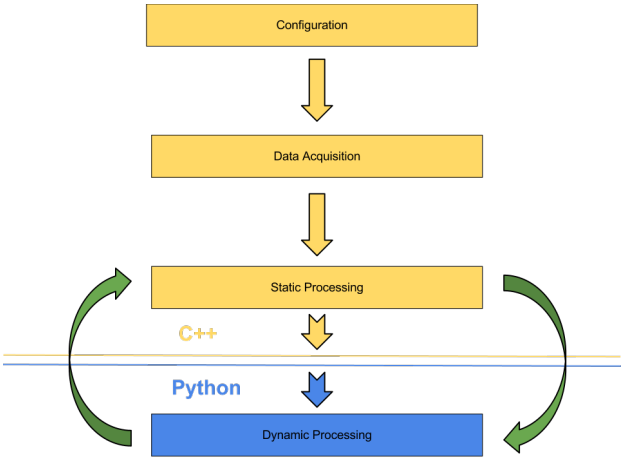

Fig. (2): System implementation with boundaries

acquisition currently has two options: WAV file input and mp3 encoded file input. Depending on the extension of the named file, data is stripped using libmpg123 for mp3s, or libsndfile for WAV format. Eventually, the data acquisition stage will also include the ability to processes streams of data, as well as CSV and raw binary files. Once the initial timeseries data is obtained, it is loaded into a block which will travel through the entire processing chain, gaining new entries as the data is transformed end manipulated.

Next, static processing is applied to the data using compiled blocks of processing code. Currently, the static blocks available perform FFTs, Cepstral Transforms, Bark Frequency Cepstral Coefficients, and DCTs. These blocks use libfftw3 and the Eigen template library to perform Fourier transforms and matrix operations. Once the static processing chain has completed, the data and it's transform representations are loaded into a Python interpreter, which then exposes the computed arrays directly to a configured list of Python scripts to be executed in the dynamic stage.

2) Python: The Python sections allow a high-level, interactive interface to the data and it's representative transforms. Often, advanced statistical modeling, plotting routines, and complex analysis are performed in this stage. Most of these operations take advantage of the wide variety of scientific and mathematical packages available for Python. There are also configurable options to run the dynamic stage repeatedly, which allows developers to change the Python scripts and immediately see the results of these changes, without having to redo the expensive data transform computations.

3) Diagram: The system specification shown in Figure 1 can now be represented with implementation boundaries, as in Figure 2.

B. Processing

1) FFT and Power Spectral Density: To understand the FFT, it is important to first describe the DFT, the mathematical equation on which the FFT algorithm is based. The basic DFT can be represented by Equation 1. [1]

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-j2k\pi n}{N}}; k = 0, ..., N-1 \quad (1)$$

The FFT is a mathematical transformation of the DFT which reduces the number of computations required to compute the result. Functionally, the DFT and FFT are equivalent. The FFTW library used in this framework attempts to combine many different FFT algorithms into an optimum plan for a particular set of data, given an expected FFT length and number of FFTs to compute.

Once the FFT is completed, the PSD is also calculated using Equation 2. [1]

$$X_k = 20\log_{10}(\sqrt{real(x_k)^2 + imag(x_k)^2}) \quad (2)$$

2) Cepstrum: The cepstrum is the FFT of the PSD of a given set of data. It can be thought of as an operation to see the frequency of the envelope of frequency components. The cepstrum captures both time and frequency information, but the additional layer of abstraction can cause confusion in visualization and interpretation. The signal, once transformed, is represented in the the quefrency domain, and analysis in this domain is often used for speech processing. [1], [2]

3) DCT: The DCT and the DFT are highly related to each other - in fact, the DCT can be considered a direct subset of the DFT. By removing the sine components of a DFT computation, the DCT is performed. There are many different formulations for the DCT, represented by roman numerals I-VIII. The most widely used equation for the DCT, DCT(II), is represented in Equation 3. [1]

$$X_k = \sum_{n=0}^{N-1} x_n \cos(\frac{(n+\frac{1}{2})\pi k}{N}); k = 0, ..., N-1 \quad (3)$$

4) Bark Filter Bank: A bark filter is one of a class of nonlinear filters which model the physical characteristics of hearing. This filter can be conceptualized in the frequency domain as a series of log-spaced overlapping triangles. It can also be represented as shown in Figure 3.

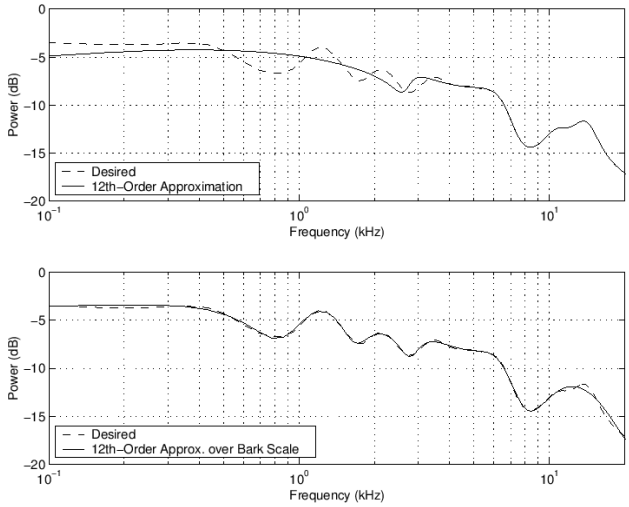

Fig. (3): Visual representation of a bark filter [2]

Efficient implementation of the bark filter can be accomplished in the frequency domain, using matrix multiplication as below.

$$\begin{bmatrix} B_{1,1} & \cdots & B_{1,C} \\ \vdots & \ddots & \vdots \\ B_{M,1} & \cdots & B_{M,C} \end{bmatrix} = \begin{bmatrix} T_{1,1} & \cdots & T_{1,N} \\ \vdots & \ddots & \vdots \\ T_{M,1} & \cdots & T_{M,N} \end{bmatrix} \begin{bmatrix} F_{1,1} & \cdots & F_{1,C} \\ \vdots & \ddots & \vdots \\ F_{N,C} & \cdots & F_{N,C} \end{bmatrix} \quad (4)$$

$\boldsymbol{T}$ is the FFT results matrix
$\boldsymbol{F}$ is a repeated Bark filter; with all rows identical i.e.

$$F_{1,1} = F_{2,1} ... = F_{N,1}$$

$\boldsymbol{B}$ contains the resulting Bark frequency coefficients
C=Number of Bark filter coefficients
N=Number of FFT bins
M=$\frac{datalen}{fftlen}$; Number of FFT frames contained in data

III. Images

Figure 4 and Figure 5 show time series and frequency displays, respectively, from one of the dynamic processing blocks. For displaying data in the dynamic processing stage, the Python modules matplotlib and pylab are used. Note that the spectral display has not been shifted, so zero frequency is in the center of the graph.

IV. Future Work

There are many improvements to be made to this toolkit. Near future improvements include an interface for streaming data, additional configuration parameters, more dynamic processing tools, the ability to splice static and dynamic processing stages, and the addition of a polyphase filter block utilizing FFT overlap-save to the static processing tools. Possible further extensions include making the data stream more dynamic, lowering the memory usage of the application, and adding support for static processing tools implemented in CUDA.

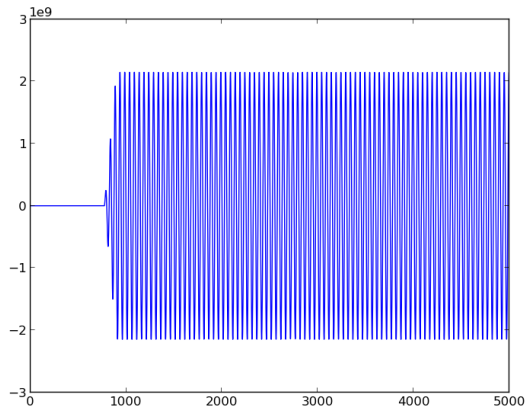

Fig. (4): Time series display

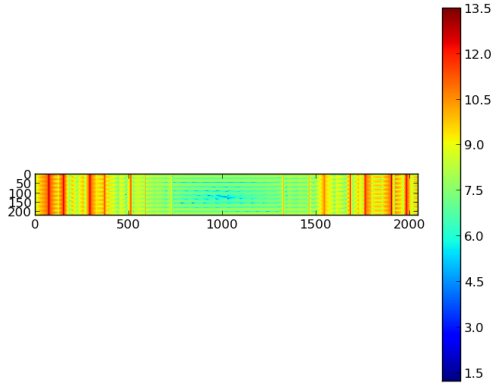

Fig. (5): Spectral display

V. Conclusion

Computers grow more and more complex, and great research often requires extensive knowledge of computer operation and software development. Many researchers are unhappy focusing time and effort on the study of these tools, instead of their preferred application. By providing a well-structured framework for accomplishing common tasks, time spent researching can be focused exploring research topics, instead of reimplementing existing ideas for each project.

References

[1] A. Oppenheim, R. Schafer, and J. Buck, *Discrete Time Signal Processing*, 2nd Edition. Prentice-Hall, 1999.
[2] J. Smith, *Spectral Audio Signal Processing*. W3K Publishing, 2011
[3] J. Smith and J. Abel, *Bark and ERB Bilinear Transforms*, IEEE Transactions on Speech and Audio Processing, November, 1999.