The ICST DSP library is a compact collection of C++ routines with focus on rapid development of audio processing and analysis applications. Unlike other similar libraries it offers a set of technical computing tools as well as speed-optimized industrial-grade DSP algorithms, which allow one to prototype, test and implement real-time applications without the need of switching development environment. The package does not rely on any third-party libraries, supports multiple platforms and is released under FreeBSD license.

1. INTRODUCTION

In the last few years, several DSP libraries, frameworks and toolkits have been published, which are aimed at the implementation of real-time audio applications. However, the ICST DSP library was developed because none of the existing open-source packages were found to offer at the same time a versatile, streamlined and computationally optimized set of DSP tools.

The most relevant feature distinguishing the ICST DSP library from other similar packages, is the inclusion of technical computing tools (similar to those provided with e.g. MATLAB), as well as of highly optimized DSP routines (e.g. computationally efficient mathematical operations) that can be used as low-level building blocks for real-time audio processing applications. In addition, the library offers several optimized higher-level algorithms (e.g. audio oscillators and filters, extraction of audio features, management of audio files, etc.) that can be directly implemented as components in audio applications.

Given its wide offering of different and self-contained tools, the ICST DSP library can be therefore used by developers throughout the whole process of prototyping, testing and implementing DSP applications, in this way avoiding the error-prone and tedious recoding processes associated with the usual iterations between different programming languages and development environments.

One important requirement for the implementation of new libraries and frameworks is their compatibility with a variety of hardware and software systems. Since

the C++ language and compilers are ubiquitous across different platforms, C++ libraries like the one described in this paper have an advantage in meeting such objective. Indeed the above observation also applies to Java, but it is generally acknowledged that the latter does not offer the level of performance required by computationally intensive real-time applications. On the other hand, the common idea within the DSP community that C++ was not suitable for high-performance applications is definitely changing, and it has now been proven that C++ code and modern compilers allow to implement extremely efficient real-time applications. The ICST DSP library has moreover no dependence on third-party libraries, thus improving its portability to different systems.

To give grounds for the development of the ICST DSP library, Section 2 points out merits and shortcomings of some relevant existing packages that can be compared with it in terms of characteristics, while Sections 3, 4 and 5 respectively describe the routines, optimizations and some additional features offered by the library.

2. EXISTING COMPARABLE TOOLS

For the purpose of this overview, several popular libraries and frameworks have been selected that share some similarities with the library described in this paper. In particular, only cross-platform, open-source packages were considered, that are written in C++ and offer functions and algorithms similar to those included with the ICST DSP library.

For the sake of comparison, the following categories of algorithms are taken into account: audio synthesis; signal processing (addressing audio or generic signals); signal analysis (audio or generic signals); other music- or audio-related functions (e.g. sound files management, MIDI I/O); other routines (e.g. maths and statistics).

The Synthesis ToolKit (STK) is a library designed for rapid prototyping of music synthesis and audio processing applications [1]. The primary focus and strongest points of STK are its many routines for audio synthesis (e.g. classic waveforms / wavetable oscillators, additive / FM / modal [2] / physical modeling / voice synthesis). In addition, some audio processing algorithms are offered (e.g. delay-based effects, filters, envelope generator). No routines for signal analysis are present, while some functions for audio and MIDI I/O, and sound files management are
included. A physically-inspired particle model is provided for the control of audio synthesis processes. The STK also offers support for network communication (pipes, sockets) and for Tcl/Tk GUIs.

The main aim of the STK is not the implementation of high-performance real-time audio applications, but rather the offering of useful and versatile material for teaching audio and music processing. As a consequence of this, the included algorithms are quite basic.

Several examples of plugins (VST, AU) and applications implemented with the STK are provided, with respective projects for Visual Studio and XCode.

Licensing is managed by the copyright owners (Perry Cook and Gary Scavone).

The Sound Object (SndObj) library [3] comprises functions and algorithms pertaining to all the considered categories: sound synthesis (e.g. basic / band limited [4] / wavetable oscillators, sinusoidal modeling [5], string resonator, granular synthesis), processing (e.g. ring modulator, vocoder, filters, delays), analysis (spectral analysis, envelope follower), sound files management and MIDI I/O, other functions and algorithms (e.g. various transforms like FFT and wavelet, multiplication of spectra).

The SndObj library depends on some third-party libraries and includes platform-specific code (e.g. OSS, ALSA and Jack for Linux; MME and ASIO for Windows; CoreAudio for MacOSX). Notably the library offers multi-threading functionality and support for the Python and Java languages.

Some projects for Visual Studio, and examples for building VST and LADSPA plugins and PureData externals are provided.

The SndObj library is licensed under the GPL.

The CREATE Signal Library (CSL) [6] provides several classes for synthesis (e.g. classic waveforms / wavetable oscillators, additive / granular / Karplus-Strong synthesis), processing (e.g. delays, filters, various spectral processing algorithms, and higher-level routines for spatialization, including Ambisonics) and analysis (e.g. linear prediction, features extraction). Support for audio, MIDI, OSC and UDP I/O is present, as well as tools for the management of sound files.

The library depends on some third-party libraries. Despite the fact that its main aim is simplicity as opposed to computational efficiency, the library supports multi-threading. The CSL has been designed to be easy to learn and flexible, moreover its design is scalable and network-oriented, so that applications developed making use of it can be distributed among different machines.

Application examples and project templates for XCode, Visual Studio and Linux are provided, which make use of the JUCE framework ¹ for cross-platform portability.

Licensing for the CSL is managed by the copyright owner (University of California).

CLAM [7] is a complex and comprehensive framework (i.e. not just a library) whose purpose is the efficient design and implementation of audio and music applications. With regard to the included algorithms, CLAM is strongly biased toward spectral-domain processing, in particular exploiting the spectral modeling analysis and synthesis (SMS) technique [5]: for example, several routines for SMS-based synthesis and processing (e.g. time-stretching, harmonizer, delay) are provided. In addition to that, CLAM offers classic oscillators and filters, while several audio / music analysis and feature extraction algorithms are present (e.g. spectral / cepstral analysis, chord segmentation, fundamental frequency tracker, envelope follower). Furthermore CLAM includes functions for music information retrieval, management of sound files, audio / MIDI / OSC I/O, and support for XML, codecs, visualization of algorithms and data.

CLAM offers advanced programming features and supports multi-threading. The framework depends on some third-party libraries, and incorporates platform-specific code (e.g. Jack and LADSPA for Linux, ASIO for Windows).

Several examples for building applications and plugins (VST, LADSPA, etc.) are provided.

CLAM is licensed under the GPL.

Jamoma [8] is a compound package with a layered structure, comprising a foundation framework and a DSP framework, which are taken into consideration for this overview. The DSP framework includes basic algorithms, but several additional extension libraries are available. Some extensions of interest are the libraries for audio synthesis, digital filters, audio effects, signal analysis, windowing, math functions and gesture mapping.

Jamoma provides advanced programming features but is not strictly optimized for computational efficiency (for instance, it only offers frame-based audio processing), instead it is aimed at offering an easy environment for fast prototyping.

Wrappers for AU, VST, Max/MSP, PureData, SuperCollider plugins are provided, together with example projects, as well as bindings for the Ruby language.

Jamoma is licensed under the LGPL, which enables its use in both open source and commercial software projects.

3. LIBRARY CONTENT

The ICST DSP library is largely composed of public domain algorithms which have been tested and optimized for speed when running on modern processors (more details on this subject are given in Section 4).

The library currently comprises about 380 functions within the following classes: audio analysis, audio synthesis and processing, audio file I/O, block-based signal processing, specialized math, neural networks, 2D data visualization.

Considering the amount of routines included, it is clearly not possible to describe each of them herein, and since the available algorithms span over a very broad range of topics, any selection would always disappoint some readers.

¹ www.rawmaterialsoftware.com/juce.php
from different fields. For these reasons, below it was chosen to give only a brief summary of the library’s content, and to refer the publications related to a few advanced routines and algorithms. However the library comes with a companion book \(^2\) which describes in detail many of the included routines for audio synthesis and processing, with emphasis on their implementation in terms of accuracy and computational efficiency.

**Audio synthesis:** The library offers different types of audio synthesis routines, from the synthesis of basic waveforms to virtual analog (band limited \([4]\)), sample playback and wavetable oscillators, FM operators and noise generator. Furthermore, a resynthesis engine is included.

**Signal processing:** Various filters are present, having different purposes: ready-made audio-oriented filters are included for use e.g. in classic subtractive synthesis chains (namely, first order low-pass / high-pass and second order multimode \([9]\) filters, virtual analog Moog-inspired filter \([4]\)); static FIR and IIR (all-pole) filters, static first and second order sections for use in the implementation of more complex filters; median filters.

Audio-oriented static and interpolating delay lines are provided, together with a delay building-block for use in higher-level structures. Moreover, the library includes a ring modulator and an envelope generator, in addition to routines for converting audio to phase modulation signals and for phase-shifting signals using the the Hilbert transform.

**Signal analysis:** The library comprises several routines aimed at the analysis of signals: spectral and cepstral analysis; analysis by decomposition to arbitrary functions; linear prediction; feature extraction, like tracking of partials and fundamental frequency (e.g. using McLeod-Wyvill \([10]\) or YIN \([11]\) methods), detection of cutoff frequency and transients, envelope follower; energy and power calculations.

**General and signal-oriented math:** Several routines for fast computation of mathematical functions are included (e.g. exponential, gamma, beta, Bessel functions), for function and polynomial root-finding, generation of polynomials from given roots, symbolic calculations on polynomials, and a ordinary differential equations solver using Runge-Kutta fourth-order method. Furthermore different interpolation methods, polynomial fitting and efficient operations on real / complex arrays and real matrices are implemented, including convolution and correlation in the time-domain and frequency-domain.

The library also offers tools for the design of continuous-time and digital filters, and for the analysis of continuous-time and discrete-time systems (calculation of frequency response and group delay).

Besides, functions for the calculation of common transforms (e.g. forward and inverse FFT, DCT, DST, wavelet) are provided, as well as different windowing methods and utilities for the generation of useful signals, and the implementation of circular buffers.

**Statistics and neural networks:** The library includes different functions aimed at statistical computations (e.g. covariance, correlation, linear regression, t-test, identification of outliers) and offers tools for principal / independent component analysis.

Furthermore, several tools for the implementation of neural networks are present.

**Utilities:** Finally, the library includes some tools for audio file management and for data visualization as 2D diagrams and charts (Windows only, using MFC).

4. SPEED OPTIMIZATIONS AND CONFIGURATION OPTIONS

Many of the included signal processing algorithms and math routines have been optimized for high performance on modern x86 processors running Wintel- or Unix-based systems. However, being written in standard C++ and not having dependencies on third-party libraries, the ICST DSP library can be easily used in its unoptimized version on other platforms as well.

Whenever appropriate for efficiency, and without sacrificing the signal quality required by professional audio applications, the library internally makes also use of single-precision floating point arithmetic.

The library comprises several computationally efficient block-based processing routines. On the other hand, whenever possible such routines accept a minimum block size of one sample, thus enabling their use in sample-by-sample processing algorithms (e.g. digital filter structures with feedback). To ensure maximum flexibility, the routines also offer zero-overhead support for streaming blocks of varying sizes.

4.1 SSE2 optimizations

The ICST DSP library’s core functions make use of hand-optimized code exploiting SSE2 intrinsics for maximum speed.

In this regard, many core functions internally exist in two different flavors: a faster one which exploits aligned data, and a slower one with unaligned memory access. The appropriate version is dynamically selected, and although this process is entirely transparent, the user is responsible for providing aligned data to achieve optimum speed. To this end, the library offers functions and a macro to allocate and free aligned arrays and variables.


\(^3\) The routines for FFT, DCT and DST computation are ports of Ooura’s FFT package, available at www.kurims.kyoto-u.ac.jp/~ooura/
4.2 Common DSP issues

Calculations involving so-called “denormal” (or subnormal) numbers can take a very long time on some popular processors. Denormal numbers (or denormals) fill the underflow gap around zero in floating point arithmetic: for example, a IEEE 754 single precision floating-point number – which is commonly used to represent the C++ float data type – is denormal if it is non-zero and has an absolute value below \(10^{-38}\). If denormals occur frequently, the CPU load skyrocket and the time constraints necessary for real-time applications are easily violated. While there may be a flag to tell a particular CPU to force denormals to zero, a DSP plugin making use of a library that relies on such flag can nevertheless incur a number of problems: for instance, a host program that calls such plugin may still decide to set the flag the other way, or even worse other additional libraries may conflict if they rely on denormals being processed as such. This issue has plagued signal processing on general-purpose CPUs for a long time, and still no commonly accepted and satisfying solution is in sight. For this reason many different approaches exist, offering different trade-offs between ease of use and efficiency.

The approach and considerations adopted internally to the ICST DSP library are summarized here below:

- **Safe numbers** are defined as those which are either zero, or very unlikely \((p < 0.001)\) to have an absolute value \(< 10^{-18}\).

- Input arrays that consist of safe numbers produce a negligible amount of denormals with any library function, except for the rare case of evaluating a polynomial that has more than two consecutive zero coefficients.

- No precautions are taken for functions that always produce safe numbers when the input switches between values that produce safe numbers.

- If a function could produce a sequence of denormals internally or unsafe numbers at the output, when the input switches between values that produce safe numbers in the steady-state an anti-denormal scheme is used, as described in the remarks section of the affected function documentation.

- The library neither depends on specific CPU settings nor it changes any flags associated with the handling of denormals.

Users of the ICST DSP library should take care of not producing streams of unsafe numbers. In order to achieve that, they should choose \([-1, 1]\) as the standard interval for calculations. Additionally they should consider that functions that use an anti-denormal scheme may produce small but safe numbers: if this output is not an immediate input to another function that uses an anti-denormal scheme, it is advisable to check whether successive operations might result in unsafe numbers, and if so, to call the function `prune()` with a limit as high as possible (i.e. as allowed by the application being developed for setting small numbers to zero).

4.3 Configuration options

The library provides several customization options for compatibility and speed. This is made possible by some preprocessor definitions that the user can comment/un-comment in the file `Common.h`, or add to the compiler’s settings. Below is a brief description of such preprocessor definitions:

**ICSTLIB_NO_SSEOPT** With the default settings (definition disabled) the target processor must support SSE2 and may be in any rounding mode. If either the target processor is not guaranteed to support SSE2 or the compiler cannot translate SSE intrinsics, this preprocessor definition must be enabled so that no SSE code is generated. This results in improved compatibility across different platforms, with the trade-off that many core functions run slower, typically 2 to 5 times.

**ICSTLIB_DEF_ROUND** This preprocessor definition can be used if one can ensure that the floating-point rounding mode will be in the default state (i.e. round to nearest) whenever a library function is called. This maximizes the speed of the conversion routines that round floating-point numbers to integers, of the function `fexp()` for fast exponential calculation, and of functions that depend on these.

**ICSTLIB_ENABLE_MFC** This definition is used to indicate that the C++ development environment supports the Microsoft Foundation Class library. This is only used to make the `Chart` class available, for data visualization purposes.

**ICSTLIB_USE_IPP** This definition adds support for Intel’s Integrated Performance Primitives (IPP)\(^4\) in order to speed up certain transforms. This option should be considered if the application being developed spends a considerable amount of time (i.e. at least 20% of its execution time) to compute transforms. For maximum performance it is necessary first to initialize the IPP CPU dispatcher, which detects CPU features at run-time and then selects the corresponding IPP-optimized library set, and then call the function `PrepareTransforms()`. Conversely, the function `UnPrepareTransforms()` must be called upon termination of the application.

5. REMARKS AND ADDITIONAL FEATURES

In addition to the speed optimizations described in Section 4, the ICST DSP library has been optimized for having a very small memory footprint.

The library is thread-safe, and several of its routines which are aimed at real-time audio synthesis and processing exploit the multi-threading capability of

\(^4\)It should be noted that Intel’s IPP package is a commercial product that is not compliant with the BSD license used for the ICST DSP library. More information at software.intel.com/en-us/articles/intel-ipp/
x86/AMD64/ECMA memory models: to this end, their internal processing method `Update()` runs in parallel with any other function called from a separate thread.

In order to avoid possible clashes with other third-party libraries, the ICST DSP library introduces the namespace `icstdsp`.

The library was designed to be self-contained, that is it does not rely on third-party libraries. In particular, considering the large variety of existing hosts and protocols for audio I/O and control, and since the scope of the library is processing, it was decided to omit any interface functionality and leave this layer to dedicated frameworks and libraries (e.g. for interfacing with Core Audio or ASIO drivers, MIDI and OSC protocols).

The ICST DSP library is open-source software and is released under FreeBSD license, thus allowing its use for the development of new applications from open-source to commercial ones.

The library is currently being maintained by the author at the Institute for Computer Music and Sound Technology (ICST) of the Zurich University of the Arts (ZHdK).

The current version of the library (v1.2.1) is in beta status, and has been tested on both x86 and x64 systems under Windows (in particular, with Visual Studio 6, 2008 and 2010 and Intel C++ Compiler 10), Mac OS (with GCC 4.0.1 and 4.2.1, and GCC LLVM 4.2.1 under XCode) and Linux (with GCC 4.4.5).

A `git` repository has been set up, which offers public read-only access. The repository presently hosts the library, a quick user guide, and some example projects for XCode and Visual Studio. Some of the currently included example projects show how to build the ICST DSP library as a static or (on Unix systems only) dynamic library, while others implement a few of the library’s higher-level routines (e.g. multimode filter, Moog filter, virtual analog oscillator, fundamental frequency tracker) as externals for Max/MSP and PureData. These example externals have been implemented making use of the C++ programming layer `flixt`, which enables one to build cross-platform externals for Max/MSP and PureData on different operating systems (Mac OS, Windows and Linux).  

6. FUTURE PLANS

Several additional features could be included in the ICST DSP library, addressing different aspects.

Further high-level components, such as waveguide [12] and modal resonators [2]), and additional analysis tools, such as the CQT transform [13] and Gammatone filter bank [14], could be added. Also, some of the provided routines could be improved: for example the current fractional delay lines could employ alternative interpolation methods [15], and digital filters could be obtained from their continuous-time counterparts by applying the impulse invariance method as an alternative to the bilinear transform [16], which is the only option currently available.

At present time, the available documentation consists of a HTML document which offers a brief introduction to the library and its usage, plus a functional reference to the included routines. The latter would especially benefit from an inline-type of documentation such as Doxygen, so as to ensure that the documentation is always up-to-date with the source code.

In order to promote the diffusion of the library, new sample code and projects will be provided for building additional plugins for e.g. Max/MSP, PureData, SuperCollider, CSound, and in the AU/VST format.

7. CONCLUSION

This paper describes the ICST DSP library, a compact C++ library aimed at rapid development of audio processing and analysis applications.

Several features contribute to distinguish it from other comparable libraries and frameworks, in particular: a) the offering of hand-optimized high-level routines (e.g. oscillators and filters), as well as computationally efficient building-blocks for real-time signal processing; b) the provision of technical computing tools that enable developers to prototype, test and implement real-time applications without the need to switch among different development environments.

Acknowledgments

The author wishes to thank Beat Frei, who originally implemented the library for the ICST, and whose documentation was partially included in this paper.

8. REFERENCES


