Music 420A / EE 367A: Introduction to the Synthesis Tool Kit (STK)

Julius O. Smith III

Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music, Stanford University, Stanford, California 94305 USA

April 15, 2017

Getting Started with the Synthesis Tool Kit (STK)

This intro provides a "getting started" guide for the Synthesis Tool Kit (STK) [1, 2] under Linux [3] and Mac OS X. This guide is aimed at *developers* of STK software, as opposed to mere users. As a result, a couple of modifications will be introduced to facilitate software development (*i.e.*, *debugging* `C++` code). It is assumed that the reader already has some familiarity with programming in `C` and ideally also `C++`.

Installing and Testing the STK

As of this writing (April 14, 2017), the preferred way to obtain the STK software is by cloning the GitHub project:

```
git clone git://github.com/thestk/stk
```

Doing it this way gets around a "broken pipe" problem in one of the percussion demos. It also gives you the latest and greatest at all times, including any new bugs introduced by the developers. Any bugs encountered should be reported on Piazza, and if appropriate we file a bug report on the "Issues" page at the STK GitHub site. (Feel free to do this yourself if you are an experienced software developer.)

Alternatively, the STK software distribution may be obtained by going to the STK home page,[1] navigating to the download page,[2] and downloading the latest "source distribution" (`stk-4.5.1.tar.gz`[3] as of 4/15/2017).

See the file `INSTALL.md` for instructions on installing it. The basic drill for users (see below for developers) is to `cd` to the top-level `stk` directory in a shell (such as an Emacs shell, or `Terminal.app` on the Mac) and type

```
autoconf
./configure
make
```

[1] `https://ccrma.stanford.edu/software/stk/`
[2] `https://ccrma.stanford.edu/software/stk/download.html`
[3] `https://ccrma.stanford.edu/software/stk/release/stk-4.5.1.tar.gz`

where the `autoconf` utility is part of the GNU Autotools (already installed at CCRMA). On a Mac, you can use either MacPorts or Homebrew to install `autoconf`, e.g.,

```
> sudo port install autoconf
```

where the `port` command is installed when MacPorts[4] is installed.

> **Note:** As of this writing (April 15, 2017), on the Mac, there is a missing `configure` dependency for the Jackmp framework, so that the `play` command fails at run-time. This does not happen on the CCRMA Linux systems.
>
> It is recommended anyway to use the `play` command installed by the `sox` distribution:
>
> ```
> port install sox
> ```
>
> where the `port` command is installed by MacPorts.

Familiarizing with the STK

- Read the file `README.md` to get oriented regarding the STK.
- Study the STK documentation in `doc/`.
- Read the files `INSTALL.md` about configuring and compiling the STK. The `configure` script, which itself was generated by `autoconf` from the "autotools" distribution [4], generates all the various STK `Makefile`s based on automatically detected characteristics of your system.
  - If you want to use the JACK audio server (typical on Linux systems), for example, you can add the `--with-jack` option to `configure` when generating the `Makefile`s.
  - For the Mac, you can add the `--with-core` option to `configure` when generating the `Makefile`s.
- A fairly comprehensive STK test is to make and run the main demo:

  ```
  cd projects/demo
  make
  StkDemo
  ```

  Be sure to try selecting different instruments in the `Instruments` pull-down menu.

STK Subdirectory Overview

Now `cd` to the top-level STK directory and follow along:

- The `doc` directory contains a copy of the STK documentation, including the release notes, platform-specific README files, and some papers (`doc/html/papers/`). Other doc, such an introductory tutorial[5] and `C++` class documentation[6] can be browsed online. (The `C++` class documentation was generated from the source files by the `doxygen` utility.)

[4] `https://www.macports.org/`
[5] `https://ccrma.stanford.edu/software/stk/tutorial.html`
[6] `https://ccrma.stanford.edu/software/stk/classes.html`

- The `src` directory contains `C++` source files for the STK *library*. The library is built by going to that directory and typing *make*.
- The `include` directory contains `C++` header files for the STK library.
- The `rawwaves` directory contains "raw" 16-bit soundfiles, *i.e.*, no header of any kind.[7]
- The `projects` directory contains a variety of subdirectories holding programs which exercise the STK:

```
> ls -F
demo/  effects/  eguitar/  examples/  ragamatic/
```

  - Above, `>` denotes the shell command prompt (*i.e.*, don't type it).
  - The `demo` directory contains some demo programs. Below we will go into that directory and try out a demo.
  - The `effects` directory contains source for the `effects` program which applies various digital audio effects to microphone input in real time (much fun!). See `README-effects.txt` in that directory for more info.
  - The `eguitar` directory contains a basic electric-guitar example.
  - The `examples` directory contains a variety of short programs illustrating various programming techniques in the STK.
  - The `ragamatic` project is a larger demo which plays automatically generated meditative music on sitar, drones, and tabla. There are sliders for configuring parameters of the performance and various raga presets.

Overview of STK Demos

A variety of interesting demo scripts (extension `.bat`) are contained in the demos directory `projects/demo/`:

- `Drums`: Provides Tcl/Tk selection-buttons for the Bass, Snare, LoTom, MidTom, HiTom, Homer, Hat, Ride, Crash, CowBel, Tamb (tambourine), and Homer ("doh!") patch presets
- `Modal`: Has demo-select buttons for the Marimba, Vibraphone, Agogo, Wood1, Reso, Wood2, Beats, 2fix, and Clump presets
- `Shakers`: Demos the Maraca, Sekere, Water Drops, Tambourine, Sleigh Bells, Maraca, Sekere, Cabasa, Bamboo, Water Drops, Tambourine, Sleigh Bells, Guiro, Sticks, Crunch, Wrench, Sand Paper, Coke Can, NeXT Mug, Mug & Penny, Mug & Nickle, Mug & Dime, Mug & Quarter, Mug & Franc, Mug & Peso, Big Rocks, Little Rocks, and Tuned Bamboo patches
- `Physical`: Includes Clarinet, BlowHole, Saxofony, Flute, Brass, BlowBotl, Bowed, Plucked, StifKarp, Sitar, and Mandolin patches

[7] The STK has a compile-time macro called `RAWWAVE_PATH` which defaults to "`../../rawwaves/`". This works for programs located in directories such as `projects/someproject/` within the top-level STK directory. One can set the `rawwaves` path at run time with the message `Stk::setRawwavePath()` sent to the global `Stk` object; similar messages set other global variables such as the sampling rate.

- `Voice`: Includes FMVoice and Formant example patches for singing voice
- `Banded`: Demos the Bar, Marimba, GlassHarmonica, and PrayerBowl
- `StkDemo`: This master demo script uses a menu to select patches instead of buttons, and offers everything listed above.

The demos most closely related to Music 420A are `Modal` and `Physical`. Type, for example,

```
> cd projects/demo
> make
> ./Modal.bat
  [Press the NoteOn button - you should hear a Marimba strike.]
  [Try out some of the other selections using buttons along the top.]
> ./Physical.bat
  [Press NoteOn for a clarinet toot---then NoteOff.]
  [During a clarinet NoteOn, try the "Breath Pressure" slider}
```

STK Tutorial Examples

The STK documentation (online at the STK home page[8] or in the release itself at `doc/html/tutorial.html`) includes a series of tutorials[9] providing an overview of some STK-specific coding conventions, and several of the example programs in the `projects/examples/` directory are discussed. This is an excellent time to go read through those tutorial pages.

Debugging STK Programs in `gdb`

[The remainder of this intro can be omitted on a first reading. However, remember that it's here when you are ready to debug an STK program by single-stepping in `gdb`.]

One of the nicest aspects of working with STK programs is that all source (`C++`) is available, facilitating debugging. Single-stepping someone else's STK program is a good way to learn how it works. Object oriented software is often hard to read because its functionality is spread out over many member functions in many class files, some of which may be in separate libraries. Single-stepping the code in a debugger solves this problem by showing you exactly what code is being executed and in a natural order, complete with the ability to inspect variables, stack frames, and even larger data structures such as arrays, structs, and objects.

On Windows platforms, development tools such as Microsoft Visual `C++` provide all the debugging support you need.

On the Mac, the standard IDE ("Integrated Development Environment") is `Xcode.app`. In that environment, `C++` source files have extension `.mm` to distinguish them from purely Objective C source files (`.m`).

An IDE commonly used in Linux environments is `eclipse`. It can handle all kinds of languages by means of its plugin architecture, and is used extensively for Web programmin (Javascript, Java, and many others, in addition to `C++`).

[8] `https://ccrma.stanford.edu/software/stk/`
[9] `https://ccrma.stanford.edu/software/stk/tutorial.html`

Another IDE of sorts is the powerful text editor (and process manager) Emacs. Emacs (like other IDEs) can serve as a front end for the standard UNIX-style software development tools such as `make`, compilers, linkers, and debuggers such as `gdb`. Since Emacs is available on all major platforms, we will use that case below. For an introduction to many of its features, see Travis Skare's video intro:

`https://docs.google.com/file/d/0B85zuZBDyib8Q0Q2Q0lRSnNsdnM/edit?pli=1`

On UNIX platforms (including Linux and Mac OS X, and even Windows systems using Cygwin), the standard `C++` debugger for decades has been `gdb`.[10]

STK Debug Configuration

For ease of debugging STK software, use the following `configure` command:[11]

```
configure --enable-debug CPPFLAGS="-g -O0"
```

The `--enable-debug` flag turns on some diagnostic printout in the STK source. The compiler flag `-g` results in symbols and line-numbers being retained in the compiler output `.o` file (useful for single-step debugging in `gdb`), and the `-O0` compiler flag turns off optimization—also often needed for single-stepping at the source level in `gdb`.

An Example `.gdbinit` File

Below is an example `.gdbinit` file used with the simple example STK patch (the "acoustic echo simulator") described above:

```
echo    set args 2 \n
        set args 2

echo    dir ../../src \n
        dir ../../src

echo    b FileWvIn::tick(void) \n
        b FileWvIn::tick(void)
```

The `.gdbinit` file goes in the same directory as the main program where `gdb` is started. It assumes that the library of standard stklib modules is located in `../../src/` (which works if your main directory is located in or parallel to subdirectories of the STK `projects` folder). A correct pointer to the STK source directory is necessary for viewing source code while single-stepping standard STK modules.

Note that a breakpoint is always set at the `tick` function of the `FileWvIn` object. When the program is run, it will halt just before sound reading begins, after preliminary set-up is finished. You can also set the breakpoint at `main` in order to see absolutely everything that happens.

The "echo" commands are not required, but they remind the user that the `.gdbinit` file is being executed every time `gdb` is started in this directory.

[10] For further introductory information regarding compiling, linking, and debugging programs in the UNIX environment, see http://cslibrary.stanford.edu/107/.

[11] At the time of this writing, the STK-4.4.4 `configure` script does not allow overriding CXXFLAGS which would normally be the flags for the C++ compiler. Overriding CPPFLAGS (normally aimed at the C preprocesor) is a workaround we can use here.

Preparing your Program for Debugging

If you have followed the instructions of the preceding intro, you are already prepared for debugging in gdb, and you can skip to the next section. Otherwise, before using gdb, you need to

(1) compile your program *and the STK library* with symbols using the '-g' compiler switch, and

(2) compile without optimization (no '-O', '-O2', '-O3' options, etc. -O0 explicitly turns it off).

Optimization can cause gdb's line-number information to become incorrect (when lines of code are optimized away). Furthermore, intermediate variables are also often optimized away, in which case you can no longer inspect them while single-stepping.

For example, suppose your Makefile contains the following:

```
CFLAGS = -O3 -Wall ...
```

Before running gdb, we need to change this to

```
CFLAGS = -g -O0 -Wall ...
```

and recompile by typing "make" in that directory.

Executing gdb from the command line

Suppose the main program is called main. Then in a terminal emulation program (any "shell"), cd to the working directory containing main and type

```
gdb main
r
```

Executing gdb from Emacs

A great advantage to using gdb in emacs is that it works the same on all platforms (Linux, Windows, etc.), and it displays the source in a separate emacs buffer window. As you single-step the program, a pointer shows you which line of source code will be executed next. Many nice new features have evolved for debugging in Emacs, as illustrated at the video link given near the beginning of this section.

To start gdb within emacs, say M-x gdb <Enter>, edit the gdb execution line, if necessary, and type <Enter> again. A gdb buffer window is created by emacs, and you will see the (gdb) prompt. Set a breakpoint (if not already provided by your .gdbinit file), and type r <Enter> to run the program. When the program stops, a second emacs buffer window will be created containing the source code with a pointer "=>" indicating the next line of code to be executed. This is your first breakpoint.

In the source display window within emacs, a breakpoint can be set for the line containing the cursor by typing C-x <space>.

Useful commands in `gdb`

Below is a useful subset of `gdb` commands, listed roughly in the order they might be needed. The first column gives the command, with optional characters enclosed in [square brackets]. For example, the `run` command can be abbreviated `r`. The second column gives a short description of the command. Type `help <command>` in `gdb` to obtain more information on each command.

h[elp]	Get help on gdb commands
h[elp] <cmd>	Get help on a specific gdb command
r[un]	Run to next breakpoint or to end
s[tep]	Single-step, descending into functions
n[ext]	Single-step *without* descending into functions
fin[ish]	Finish current function, loop, etc. (useful!)
c[ontinue]	Continue to next breakpoint or end
up	Go up one context level on stack (to caller)
do[wn]	Go down one level (only possible after `up`)
l[ist]	Show lines of code surrounding the current point
p[rint] <name>	Print value of variable called <name>
p ∗ <name>	Print what is pointed to by <name>
p/x <name>	Print value of <name> in hex format
p <name>@<n>	print <n> values starting at <name>
p <chars><tab>	List all variables starting with <chars>
b[reak] <name>	Set a breakpoint at function <name>
b <class>::<name>	Set a breakpoint at <name> in <class>
b <class>::<tab>	List all members in <class>
h[elp] b	Documentation for setting breakpoints
i[nfo] b	List breakpoints
i	List all info commands
dis[able] 1	Disable breakpoint 1
en[able] 1	Enable breakpoint 1
d[elete] 1	Delete breakpoint 1
d 1 2	Delete breakpoints 1 and 2
d	Delete all breakpoints
cond[ition] 1 <expr>	Stop at breakpoint 1 only if <expr> is true
cond 1	Make breakpoint 1 unconditional
comm[ands] 1	Add a list of `gdb` commands to execute each time breakpoint 1 is hit (usually just `print <var>`)

Dealing with `C++` in `gdb`

To print instance variables of a `C++` class in `gdb`, it is sometimes necessary to include an explicit object pointer. For example, after setting a breakpoint in the `tick` method of the class `Guitar` and continuing to the breakpoint, `gdb` prints something like

```
Breakpoint 2, Guitar::tick (this=0x805cde8) at Guitar.cpp:100
(gdb)
```

The `this` variable points to the current instance of the `Guitar` object. If an instance variable, say `pluckAmp`, cannot be found when you try to print it, try instead

```
(gdb) p this->pluckAmp
$1 = 0.296875
```

Also, `this` can be dereferenced to list all instance variables, *e.g.*,

```
(gdb) p *this
$2 = {<Plucked3> = {<Instrmnt> = {<Stk> = {static STK_SINT8 = 1,
       static STK_SINT16 = 2, static STK_SINT32 = 8,
       static STK_FLOAT32 = 16, static STK_FLOAT64 = 32,
       static srate = 22050, _vptr.Stk = 0x8059020},
     lastOutput = 0}, delayLine = 0x805ce70, delayLine2 = 0x8064f18,
   combDelay = 0x806cfb0, filter = 0x8075040, filter2 = 0x80750f8,
   resonator = 0x80751a0, length = 442, loopGain = 0.99999000000000005,
   baseLoopGain = 0.995, lastFreq = 2637.0204553029598,
   lastLength = 8.36170988194581, detuning = 0.996, pluckAmp = 0.296875,
   pluckPos = 0.234567}, soundfile = 0x8075248, dampTime = 8,
  waveDone = 0, feedback = 0, noise = 0x8075088,
  excitationFilter = 0x8075398,
  excitationCoeffs = 0x0}
```

Suppose we're interested in `filter2` above:

```
(gdb) p filter2
$3 = (BiQuad *) 0x80750f8
(gdb) p *filter2
$4 = {<Filter> = {<Stk> = {static STK_SINT8 = 1,
      static STK_SINT16 = 2, static STK_SINT32 = 8,
      static STK_FLOAT32 = 16, static STK_FLOAT64 = 32,
      static srate = 22050, _vptr.Stk = 0x80590e0},
    gain = 0.60459499999999999, nB = 3, nA = 3,
    b = 0x8075140,  a = 0x8075120,
    outputs = 0x8075180, inputs = 0x8075160}, <No data fields>}
(gdb)
```

We see that `filter2` is an instance of the `BiQuad` class. Note that the superclass instance variables are enclosed in curly brackets (the leaf class instance variables begin with `gain` in this example). We are omitting `this` here, since doing it once seems to make `gdb` know about it thereafter. Suppose we want to see the filter coefficients:

```
(gdb) p filter2->b[0] @ 3
$5 = {1, -1.03, 0.21540000000000001}
(gdb) p filter2->a[0] @ 3
$6 = {1, -1.3337300000000001, 0.446191}
(gdb)
```

and so on.

Multithread Backtrace in `gdb`

Obtain a backtrace on all active threads in an application as follows:

```
(gdb) thread apply all backtrace
(gdb) # or, equivalently:
(gdb) thr ap al bt
```

Miscellaneous tricks in `gdb` under Emacs

- To get back to debugging after a spate of editing, go to the `gdb` buffer (the one named `*gud-<program name>*`) and type `up` followed by `down`. This restores the source pointer in the other window.
- Reissuing the `M-x gdb` command to `emacs` will go to the pre-existing debugging session, if any. Therefore, after the program is recompiled, `emacs`'s `gdb` buffer must be explicitly killed.
- To break when an input signal file is nonzero after a long string of zeros, use a conditional breakpoint. *E.g.*,

  ```
  cond 2 insamp != 0
  cont
  ```

- When single-stepping, everything on one line is executed by the `next` command. Thus, for example, typing `n` at the line

  ```
  for ( i=0; i<6; i++ ) { string[i] = 0 };
  ```

  will execute the entire loop, while

  ```
  for ( i=0; i<6; i++ ) {
          string[i] = 0
  };
  ```

  must be traversed all six iterations. (Of course, you can also set a breakpoint after the loop and then `continue` to it.)

Debugging Plugin Modules in Emacs

[Note: We will not be using software plugins in Music 420A. However, the information is included in case it may help with your optional-unit project work.]

DSSI and LADSPA Plugins

A *plugin* is a loadable module that is dynamically loaded by a *host program* such as the `rosegarden` music sequencer on Linux. A *LADSPA plugin* (Linux Audio Developer's Simple Plugin API[12]) is a loadable `C` or `C++` module that implements an *audio processor* such as a digital filter, reverberator, or other digital audio effect. A *DSSI plugin* (Disposable Soft Synth Interface) usually implements a *synthesizer plugin* such as for doing wavetable, virtual analog, or FM synthesis. The DSSI API has been described as "LADSPA for instruments," providing capabilities comparable to that of the popular VSTi[13]) API by Steinberg. DSSI is implemented as a superset of the LADSPA API. As a result, sound port management and basic plugin controls are handled identically as in LADSPA plugins. Extensions for DSSI plugins include support for MIDI controllers, MIDI "program select", and session management (saving and loading of configuration data). To maximize cross-platform compatibility, GUIs for editing DSSI plugins are specified to be separate stand-alone programs (or

[12] API stands for "Application Programming Interface"
[13] VST stands for "Virtual Studio Technology," and VSTi stands for "VST Instrument."

loadable modules) managed by the *host* (as opposed to the plugin itself). The plugin communicates with the host by means of Open Sound Control (OSC) messages, so the GUI can run on a separate computer (commonly done for rack-mounted hosts).

On typical Linux/UNIX systems, one can find DSSI plugins as

`/usr[/local]/lib/dssi/*.so`,

and LADSPA plugins as

`/usr[/local]/lib/ladspa/*.so`.

(Plugins of both types are installed via the Planet CCRMA distribution.) Most music/audio applications offer plugin selection via pop-up menus that are automatically generated from listings of these standard plugin installation directories. In Miller Puckett's `pd`, LADSPA plugins may be loaded using the `plugin~` "tilde object" by Jarno Seppänen, and DSSI plugins may be loaded using `dssi~` by Jamie Bullock.

A lot of information about the DSSI and LADSPA plugin APIs can be found in the respective header files

`/usr/include/dssi.h`

and

`/usr/include/ladspa.h`.

Additionally, the RFC in the `doc` directory of the DSSI "specification & example code repository" (`dssi-0.9.1` as of this writing) provides a good high-level orientation.

The remainder of this section is written specifically about DSSI plugin debugging. However, it applies with little modification to LADSPA plugins, and to loadable C modules in general. While it is possible to use `C++` in plugins, it seems to be more common to use plain `C` written in an "object oriented style". Therefore, we will postpone consideration of `C++` to a later section below.

You will need a simple host application for debugging your plugin. A good choice for this is `ghostess` by Sean Bolton. Even if `ghostess` is already installed on your system, you will want to download and compile it from source so that you can single-step within it as well as your dynamically loaded plugin.

The `jack` audio server must be running for `ghostess` (as for many other Linux audio programs). One can use `qjackctl` to start the jack daemon[14] `jackd`. Be sure the `jack` sampling-rate is set to a value supported by your sound hardware.

Single-Stepping a Plugin in `gdb`

The first step is to compile your plugin with symbols retained and optimization turned off. For plugin distributions using `configure`, it should work to type

```
configure "CFLAGS=-g -O0 -DDEBUG"
```

[14] a *daemon* is a program that runs in the "background" to provide services to other programs.

(Note that '-O0' is a hyphen followed by the letter 'O', followed by the number zero.) This works for the example programs such as `trivial_synth.c` in the DSSI example code repository. By default, `ghostess` is compiled with `-g -O2`, which is fine for the host.[15]

In order for your host program to find your plugin in the current working directory where it is compiled, the `DSSI_PATH` environment variable needs to be set up containing that directory first in the search path. Since I use `tcsh` as my shell, my `~/.tcshrc` file contains the line

```
setenv MODULENAME "${HOME}/projects/dssi/modulename"
setenv DSSI_PATH \
       "${MODULENAME}:/usr/local/lib/dssi:/usr/lib/dssi"
```

Users of the `bash` shell may add something like the following in their `~/.bashrc` file:

```
MODULENAME="${HOME}/projects/dssi/modulename"
DSSI_PATH="${MODULENAME}:/usr/local/lib/dssi:/usr/lib/dssi"
export MODULENAME DSSI_PATH
```

Note that `ghostess` requires all `DSSI_PATH` elements to begin with `/`, so you must provide a full absolute path to your working directory. Of course, if you have `root` privileges on your computer, you can simply type `make install` each time you change your plugin, and the host program will find your `.so` file in `/usr/local/lib/dssi` by following the default DSSI search path.

The debugging cycle tends to go like this:

- Say `make` to compile your module.
- If you are using the default DSSI search path, type, in another window (as root), `make install`.[16]
- Start `gdb` on `ghostess` in the usual way:

  ```
  M-x gdb <Enter>
  gdb ghostess -debug -1 modulename.so <Enter>
  ```

- Use the `dir` command to provide paths to any source files not in the current working directory (typically the `ghostess` source directory).
- Set a breakpoint somewhere in your module, such as at its `instantiate`, `activate`, or `run` function. Since the module is not loaded yet, `emacs` will ask you if you want a *pending* breakpoint as follows:

  ```
  Function "modulename" not defined.
  Make breakpoint pending on future shared library load?
  ```

  Type `y` and the breakpoint will be installed when the module is loaded.
- `run ghostess` under `gdb`.

[15] While optimization level 2 interferes with detailed single-stepping and variable inspection, it does not interfere with setting breakpoints and general looking around, which is all we need for the host program.

[16] Some prefer to type `su -c "make install"`, but then you have to give the root password every time.

- Click on the `UI` button in the small main window for `ghostess`. This opens a simple editor window for your plugin that was automatically generated by `ghostess` based on its port declarations.
- Click on the `Test Note` button in the plugin editor window. This issues the equivalent of a MIDI note-on for your plugin.
- Find and fix all bugs in your plugin!

To streamline the startup, I use a `.gdbinit` file like the following:

```
echo dir /l/dssi/ghostess-20050916/src/ \n
     dir /l/dssi/ghostess-20050916/src/
echo set args -debug -1 ./modulename.so \n
     set args -debug -1 ./modulename.so
```

(I maintain a large set of convenient symbolic links in the directory `/l` on my system.) I also have a symbolic link `g -> /l/u/dssi/.../src/ghostess` in the module source directory, so in `emacs` I can simply say

```
M-x gdb <Enter>
gdb g <Enter>
```

to get started.

References

[1] P. Cook and G. Scavone, *Synthesis Tool Kit in C++, Version 4*, `https://ccrma.stanford.edu/software/stk/`, 2010, see also `https://-ccrma.stanford.edu/~jos/stkintro/`.

[2] P. R. Cook, "Synthesis Tool Kit in C++, version 1.0," in *SIGGRAPH Proceedings*, Assoc. Comp. Mach., May 1996.

[3] E. Siever, A. Weber, and S. Figgins (Ed.), *Linux in a Nutshell, Fourth Edition*, Cambridge: O'Reilly, 2003.

[4] G. V. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor, *GNU Autoconf, Automake, and libtool*, SAMS, 2000.