.

CsoundAC User Guide (Mostly for Python)

Michael Gogins
michael.gogins@gmail.com

February 16, 2025

1 Introduction

CsoundAC is a class library and software toolkit designed to support the algorithmic composition of music. CsoundAC has C++, Python, and JavaScript application programming interfaces (APIs). This ***Guide*** is based on the Python interface.

References herein are all in the form of colored hyperlinks to resources on the World Wide Web. Reference documentation for CsoundAC is here.

The major features of CsoundAC are:

- Music graphs, a hierarchical representation of musical scores and processes, modeled upon the computer graphics idea of scene graphs.
- A variety of nodes for use in music graphs, including random variables, dynamical systems, Lindenmayer systems, iterated function systems, and others.
- Operations on scores using aspects of mathematical music theory, including chord spaces, chord transformations, voice-leading, and automatic modulations.
- As the name suggests, CsoundAC interfaces closely with Csound. However, CsoundAC can also be used with any other computer music system that supports Python scripting, including digital audio workstations (DAWs) such as Reaper.

The core of CsoundAC is written in platform-neutral standard C++. The CsoundAC repository also includes a Python interface to CsoundAC. A JavaScript interface to a WebAssembly (WASM) build of CsoundAC (and of Csound) is available in the csound-wasm repository.

2 Getting Started

2.1 Installation

2.1.1 Installing Prebuilt Binaries

Prebuilt binaries of CsoundAC for macOS and Linux are available as GitHub releases. These are built for a specific version of Python, and will work only with that version. If you don't have the correct version of

Python, you should install that version alongside your existing Python. There are also dependencies on a number of other packages.

2.1.2 Building from Source Code

Binaries for CsoundAC on Linux can also be built from source code by the user as follows:

1. Clone the csound-ac repository, and open a terminal in the repository's root directory.
2. Install dependencies by running the `update-dependencies.sh` script.
3. Perform a fresh build by running the `clean-build-linux.sh` script.

2.2 Configuration

The following instructions are for binary releases that you have downloaded from GitHub and unzipped in your home directory.

2.2.1 Configuring CsoundAC

On macOS, use the Terminal app to gain access to the macOS command line. Add the following lines to your `.zprofile` script:

```
export DYLD_LIBRARY_PATH=~/csound-ac-0.5.0-Darwin/lib:$DYLD_LIBRARY_PATH
export PYTHONPATH=~/csound-ac-0.5.0-lDarwin/site-packages:$PYTHONPATH
```

Then in your home directory, execute `source .zprofile`.

On Linux add the following lines to your `.profile` script:

```
export LD_LIBRARY_PATH=~/csound-ac-0.5.0-linux/lib:$LD_LIBRARY_PATH
export PYTHONPATH=~/csound-ac-0.5.0-linux/site-packages:$PYTHONPATH
```

Then in your home directory, execute `source .profile`.

Test your installation by running the correct version of Python and importing the CsoundAC and GeneralizedContextualGroup modules. There should be no error.

```
michaelgogins@macbookpro ~/csound-ac % python3.12
Python 3.12.8 (main, Dec  3 2024, 18:42:41) [Clang 16.0.0 (clang-1600.0.26.4)] on
    darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import CsoundAC
>>> import GeneralizedContextualGroup
>>>
```

2.2.2 Configuring ctcsound

Each published release of Csound comes with a Python interface to the complete Csound API, defined in the `ctcsound.py` module. This module will work with any version of Python, but is built for a specific version of Csound. To enable ctcsound:

1. Go the Csound GitHub repository, select the `csound6` branch, and copy or download the `interfaces/ctcsound.py` file to the `site-packages` directory for the version of Python that you will use, e.g. `/opt/homebrew/lib/python3.12/site-packages/ctcsound.py`.
2. Test your installation by running the correct version of Python and importing the ctcsound module. There should be no error.

```
michaelgogins@macbookpro ~/csound-ac/user-guide % python3.12
Python 3.12.8 (main, Dec  3 2024, 18:42:41) [Clang 16.0.0 (clang
    -1600.0.26.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ctcsound import *
>>>
```

2.2.3 Configuring Reaper

Additional configuration is required to enable Python and CsoundAC in Reaper. To enable Python scripting:

1. In Reaper, open the *REAPER* menu *Settings...* dialog (might be *Configuration...* on other versions of Reaper).
2. Go to the end of the list of settings, and select *ReaScript*.
3. In the ReaScript settings, check the *Enable Python for use with ReaScript* box.
4. On macOS:
   a) In the *Force ReaScript to use specific Python .dylib* text field, enter the filename of the Python shared library for the version required by CsoundAC, e.g. `libpython3.12.dylib`.
   b) In the *Custom path to Python dll directory* field, enter the full path to the Python library root directory, e.g. `/opt/homebrew/Cellar/python@3.12/3.12.8/Frameworks/Python.framework/Versions/3.12/lib`. You can use the *Browse* button to select this directory.
   c) If the configuration is correct, you will see something like *Python: libpython3.12.dylib installed*.
5. On Linux:
   a) In the *Force ReaScript to use specific Python .so* text field, enter the filename of the Python shared library for the version required by CsoundAC, e.g. `libpython3.12.so`.
   b) In the *Custom path to Python dll directory* field, enter the full path to the Python library root directory, e.g. `/usr/lib/libpython3.12.so` or `/usr/lib/x86_64-linux-gnu/libpython2.12.so`. You can use the *Browse* button to select this directory.
   c) If the configuration is correct, you will see something like *Python: libpython3.12.so installed*.

To enable CsoundAC:

1. Exit Reaper, and run it again.
2. Use the *Actions* menu, *New action...* dialog, "Load ReaScript..." dialog to load the `ac_reaper.py` script.
3. Create a track with a blank MIDI item, and select that item.
4. Use the*Actions* menu, *Show action list...* dialog, select the *ac_reaper.py* script, and click on the *Run* button.
5. You should see the *ReaScript console output* dialog, with output from the *ac_reaper.py* script. It should look something like this:

```
Hello, Reaper!
Selected item: <class 'str'> (MediaItem*)0x000000014383A400
Using active take of selected MIDI item.
Start time of the selected MIDI item: 0.00 seconds.
Event count: note_count: 4 cc_count: 0 text_sysex_count: 0
Cleared all contents from the MIDI item.
Inserting note: Start: 0 Start PPQ: 0.0 End PPQ; 960.0 Key: 60 Velocity: 100
Inserting note: Start: 0.5 Start PPQ: 960.0 End PPQ; 1920.0 Key: 62 Velocity
    : 100
Inserting note: Start: 1 Start PPQ: 1920.0 End PPQ; 2880.0 Key: 64 Velocity:
     100
Inserting note: Start: 1.5 Start PPQ: 2880.0 End PPQ; 5760.0 Key: 67
    Velocity: 100
MIDI notes added successfully! Check the editor.
```

6. If you see the generated notes in the MIDI editor, and there is no error message, then your configuration is correct.
7. If you see the generated notes but also see *WARNING: Could not import CsoundAC!*, then Reaper can run Python code, but the installation or configuration for CsoundAC is not correct. The most likely causes are that the custom path to the Python library is not correct, or CsoundAC files are not located in the correct `site-packages` directory.

3 Examples

The following examples proceed step by step, repeating each example in standalone Python, Csound's score bin facility, and Python in Reaper:

1. Generate a chromatic scale using just Csound and Python.
2. Generate the same chromatic scale using CsoundAC.
3. CsoundAC compositions demonstrating some capabilities of CsoundAC.

For insight into the code, go to the hyperlinked source code or open the examples in an editor, and read them in order; there are many explanatory comments.

> And in the following, specifically *compositional* remarks are framed like this.

3.1 Basic Python Usage

Here, the same very basic example is run in three environments. All three examples use the same Csound orchestra, `basic.csd`, which is defined in the Python code as a Python string literal (`(r"some␣string␣with␣perhaps␣many␣lines␣and␣escape␣characters"`). All three examples use a simple **`for`** loop to generate an ascending chromatic scale of notes. The only differences between the examples are in how they are run, and how they interface with Csound. CsoundAC is not used — these examples show only the most basic, low-level method of generating a score with Python.

> The ascending scale is used as a deliberately simple starting point. However, note that the alternation of Csound instruments in overlapping notes, some of which contain internally varying sounds, creates a more complex texture than might be expected.

3.1.1 Running in Python: `basic-python.py`

Running in Python has the advantage of being easy to understand. A composition is just a Python program. This example uses the ctcsound module to perform the example.

Open a terminal, change to the `csound-ac/user-guide` directory, and execute `python3.12 basic-python.py`.

3.1.2 Running in Csound: `basic-score-bin.csd`

Running in Csound has the advantage of requiring no additional installation or configuration apart from Csound, Python, and CsoundAC. All of the elements of a composition are defined in one .csd file. In this case the Csound orchestra is defined outside of the Python code, which is embedded in the .csd file.

Thus, code for running Csound is not needed, as Csound is already running; instead, the script writes the generated score to a temporary filename, and Csound reads that score and performs it.

Open a terminal, change to the `csound-ac/user-guide` directory, and execute `csound basic-score-bin.csd`.

3.1.3 Running in Reaper: `basic-reascript.py`

Running in Reaper has the advantages of automatically providing a piano roll score, or even music notation, for generated scores; enabling other synthesis plugins besides Csound to render CsoundAC scores; and in general, integrating with contemporary music production practice. The Csound orchestra is defined in the state of a CsoundVST3 plugin, which receives MIDI input from the DAW. The compositional code is defined in a Python ReaScript file, which is loaded into Reaper as an Action.

This separation of concerns between composition and synthesis enables CsoundAC to send MIDI to the DAW and to other plugins, and for Csound to receive MIDI and audio from any track or plugin in Reaper.

Use the *Actions* menu, *New action...* dialog, "Load ReaScript..." dialog to load the `basic-reascript.py` script. Create a track with a blank MIDI item, and select that item. Use the *Actions* menu, *Show action list...* dialog, select the *basic-reascript.py* script, and click on the *Run* button.

You should see the *ReaScript console output* dialog, with output from the script. You should also see the notes of the chromatic ascending scale in the MidiItem. You can then assign a synthesizer, which could be CsoundVST3 or another synthesizer, to play the notes.

3.2 Minimal CsoundAC Examples

These three examples re-create the three basic examples above, changing them only to generate the ascending chromatic scale as a CsoundAC ScoreNode object, and to render that using CsoundAC's integration with Csound.

1. Running in Python: `csoundac-basic-python.py`.
2. Running in Csound: `csoundac-basic-score-bin.csd`.
3. Running in Reaper: `csoundac-basic-reascript.py`.

3.3 Compositional Examples

My paper *Metamathematics of Algorithmic Composition* presents an analysis of the advantages and limitations of algorithmic composition in general. The paper is written at the level of undergraduate theoretical computer science. This reference can be skipped, but provides a deeper context for some of the remarks on algorithmic composition that follow below.

To sum it up, algorithmic composition is capable of approximating, as closely as one likes, any possible piece of music. Furthermore, different algorithms can be placed in intelligible relationships to one another, such that it is possible, *in principle*, to compose by exploring a parametric map of the algorithms. Nevertheless, the number of possible algorithms is so vast that it is simply not possible, *in practice,* to construct a usable map of related algorithms. That would take an impossibly long time.

3.3.1 How CsoundAC Works

CsoundAC is based upon the concept of a *music graph,* which is a tree (directed acyclical graph) of Nodes in score space. This concept is quite similar to the concept of a *scene graph* in 3-dimensional computer graphics, in which objects in a scene are similarly organized as a tree of nodes in 3-dimensonal space. Score space, however, has 12 dimensions:

TIME Time in seconds relative to the origin of the space.

DURATION The duration of this event in seconds.

STATUS MIDI status number, normally 144 for "note on."

INSTRUMENT MIDI channel number or Csound instrument number - 1.

KEY MIDI key number from 0 to 127 in semitones; may have a fractional part to represent microtones; middle C is 60.

VELOCITY MIDI velocity number from 0 to 127; can loosely be considered decibels of sound pressure level.

PHASE Phase of the sound, used mostly for Events that represent grains of sound.

PAN Location in physical space from left to right, in the interval [0, 1] where .5 is the center.

DEPTH Location in physical space from in front to behind, in the interval [0, 1] where .5 is the center.

HEIGHT Location in physical space from below to above, in the interval [0, 1] where .5 is the center.

PITCHES Represents a pitch-class set as a sum of powers of 2 where each power represents one of the pitch-classes in 12 tone equal temperament.

HOMOGENEITY Set to 1 to enable transformation of Events using homogeneous matrix transformations.

A point in score space most often represents a musical note, but can also represent a chord or scale, a change in a control parameter, or even a grain of sound. The semantics of these dimensions is based on MIDI channel messages, but they have floating-point precision and add duration and other fields.

The following algorithm is used by CsoundAC to generate scores. A composition is an instance of the MusicModel class, which contains a tree of child Nodes representing the music graph, a Csound orchestra, and an embedded instance of Csound. The algorithm performs a depth-first traversal of the music graph by recursively calling each child's `void Node::traverse(const Eigen::MatrixXd &global_coordinates, Score &global_score)` method:

- Multiply the global coordinates by the local coordinates to obtain composite coordinates.
- Descend into each of this Node's child Nodes with the composite coordinates and an empty Score.
- Call `Node::transform` to transform any or all Events produced by the child nodes, which are in the composite coordinate system.
- Add the child Events to the global score.
- Create another empty Score in which to optionally generate new Events, and call `Score::transform` to move these new Events into the composite coordinate system.
- Add these generated Events to the global score.

In standalone Python, once the global Score has been generated, the MusicModel uses an embedded instance of Csound to compile its orchestra, translates the global CsoundAC Score to a Csound score, and renders that score with the orchestra.

In Csound using the `<CsScore bin=python3>` facility, the generated Score must be translated to a Csound score and written to the temporary filename.

In Python ReaScript, the generated Score must be passed to the `ac_reaper.score_to_midiitem` function.

The following three examples demonstrate some typical uses of CsoundAC. Two implementations are provided for each example: as standalone Python, and as Python ReaScript for Reaper.

3.3.2 Transforming Source Material: python-trans-scale.py

This piece takes the ascending chromatic scale of the previous examples, and uses CsoundAC to transform and elaborate it.

1. Play the scale.
2. Repeat that scale, shuffled so that notes are in random order.
3. Repeat that transformed scale, but change the note durations so that the random notes form a connected line.
4. Repeat that connected scale, but make the movements of the voice from note to note occur within a more restricted range.

5. Repeat that transformed scale three times, as a canon at the sixth, on offset beats.
6. Repeat that canon, and apply to it semi-randomized tonal chord progressions and modulations.

Read the code for more details.

> CsoundAC provides two facilities for working with harmony. The first is based on neo-Riemannian chord transformations, e.g. the Generalized Contextual Groups of Fiore and Satyendra, implemented in the chord spaces studied by Callendar, Quinn, and Tymoczko. The second is based on tonal harmony as schematized by Tymoczko, also implemented by me in chord space. The final section of `python-trans-scale.py` piece uses my implementation of Tymoczko's scheme.

3.3.3 Translating Images to Scores: `python-imagery.py`

This piece takes a high-resolution digital photograph, and uses CsoundAC to:

1. Load the photograph in an ImageToScore node.
2. Set a threshold to filter out much of the image, and use only the brighter features.
3. Set a maximum number of simultaneous voices.
4. Call `ImageToScore.generateLocally` to translate the brighter features to CsoundAC Events (notes). Hue (color) is instrument, and Value (brightness) is loudness. Saturation is not used.
5. Add the ImageToScore node to a Rescale node.
6. Set appropriate ranges for the various dimensions of the score space in the Rescale node. *Please note: time is not rescaled!* That is so that the duration of the score created by the ImageToScore node can be used to determine the intervals between chords.
7. Use the CsoundAC Scale class to generate a series of chord progressions and modulations.
8. Add the resulting Chords to a VoiceleadingNode, which will apply them to the Events in the generated and rescaled score.
9. Add the Rescale node to the VoiceleadingNode.
10. Add the VoiceleadingNode to the MusicModel.
11. Call `MusicModel.generate` to the generate the score, applying each node to the notes produced by its children.
12. Set the final generated score to the desired length.
13. Call `MusicModel.perform` to render the score using the Csound orchestra.

> In a piece like this, the composer must have some sort of starting point such as a specific image, an idea of the density of texture desired, or a Csound orchestra. After that, the piece can only be improved by an iterative process of trial and error. Best practices:
>
> - Before changing anything, make a backup copy of the current state of the piece. This can be done either by commenting out a line or block of code that has been changed, or by saving versions under different names, or by committing each variant of the piece to a version control system such as Git.
> - Change one thing.

- Listen to both versions, if necessary all the way through, or even more than time.
- It may be useful to load two versions into a multitrack audio editor that can show both the spectrum and the wavefrom of each track, and to toggle back and forth between the versions while playing.
- What may sound worthless at first can sometimes be improved a great deal.
- If a series of changes ends up spoiling the piece, go back to the last version that you did not make worse, and try changing something other than what you had changed.
- In the ImageToScore node, there are parameters that can produce different textures from the same image, such as threshold or maximum number of voices.
- Each pixel in an image might become a separate note, and in a high resolution image, the tempo and density can be high. For example, the image in this example has 4,032 columns (time steps). In fact, with a fast tempo and a large number of voices, notes will blur into granular type sounds; a score 3 minutes long at this density could have some notes lasting only 0.045 seconds. This may be desirable, or not. If you need a slower tempo, multiply the duration of each note in ImageToScore's score by some factor, and then call `ImageToScore.getScore().tieOverlappingNotes()`; or, of course, increase the duration of the piece.

3.3.4 Lindenmayer Systems in Chord Space: `python-duo-dualities.py`

This piece uses a Lindenmayer system defined in the voice-leading spaces of Callendar, Quinn, and Tymoczko to implement the Generalized Contextual Groups of chord transformations identified by Fiore and Satyendra. Here I use the Generalized Contextual Groups to capture the duality of major and minor so constitutive of Western music, in an abstract and generative form that is not specifically "tonal." This Lindenmayer system, presented by me at the 2009 International Computer Music Conference, generates a score as movements of chords by transformation through chord space and time, writing notes of the chords to create the score.

The piece is a modified and elaborated form of *Two Dualities* by me, presented at the 2010 International Computer Music Conference in the listening rooms as part of 360 degrees of 60x60 (Crimson Mix).

For some hints as to the Lindenmayer system commands in `GeneralizedContextualGroup.py`, see the comments in the source code.

In the Python version, the same Csound orchestra as the other examples is used to render the piece.

In the Reaper version, MIDI channels 1 through 3 are routed to one synth (the free and open source Dexed plugin), and channels 4 through 7 are routed to another synth (the free and open source Surge XT plugin). This demonstrates that the use of CsoundAC is by no means limited to Csound.

In the Lindenmayer system for Generalized Contextual Groups, the times that the chords are played are determined solely by the order in which they are written to the score, and their durations. The reason for this is to prevent more than one chord from playing at the same time. Mathematically speaking, in most music, even atonal music, the harmony is always a function of time. This restriction in the Lindenmayer system means that in it, also, the harmony is always a function of time.

As a result, the structure of a piece cannot be determined by specifying the times of the chords. But the structure can be indirectly controlled by using the push and pop commands, which can insert a sequence in the middle of a sequence. Changing the durations and voicings of chords also has an effect on structure.

Please note, the Generalized Contextual Groups have no notion whatsoever of scale, key, cadence, or modulation. Nevertheless, by stringing together chord transformations that can produce a sense of switching between“major” and “minor,” and by generally performing the close voice-leadings typical of tonal music, this Lindemayer system can generate a flow of what might be called “quasi-tonal” music.