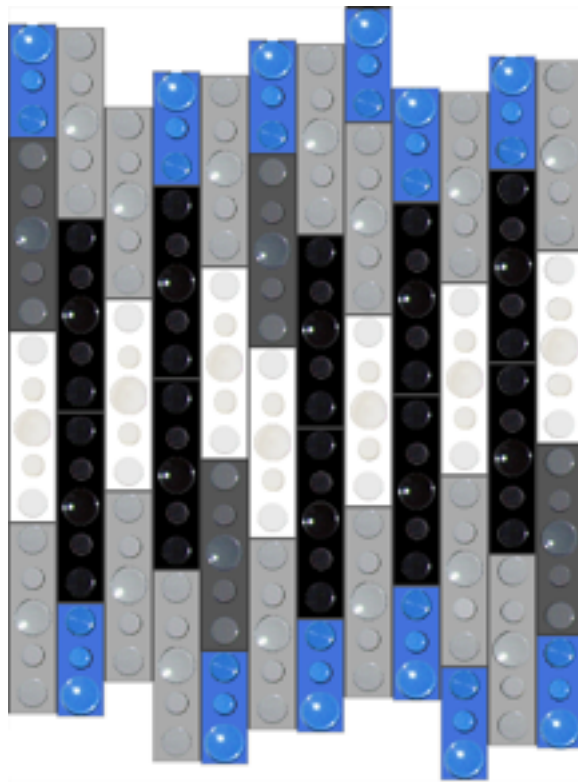


TONAL PLEXUS EDITOR



H π INSTRUMENTS

Aaron Andrew Hunt

| | |
|---|-----------|
| 1. Installation | 3 |
| <i>Mac OSX</i> | 3 |
| <i>Windows</i> | 3 |
| 2. Basics | 4 |
| <i>Built-in Help</i> | 4 |
| <i>Bug Reporting and Feature Requests</i> | 4 |
| <i>Programming a Tuning</i> | 4 |
| <i>Updating Files</i> | 4 |
| <i>Live MIDI Input</i> | 5 |
| <i>U-PLEX Owners</i> | 6 |
| <i>Split Keyboard</i> | 6 |
| <i>Using the Kontakt Script</i> | 7 |
| 3. Interface Basics | 8 |
| <i>Custom Colors</i> | 8 |
| <i>8-Octave View</i> | 8 |
| <i>Scrolling Octave View</i> | 8 |
| <i>Keys List View</i> | 9 |
| <i>Region Map List</i> | 9 |
| <i>Editing Region Boundaries</i> | 10 |
| <i>Freehand Field</i> | 10 |
| <i>Scale List</i> | 11 |
| <i>Instrument List</i> | 11 |
| <i>Live Programming</i> | 11 |
| <i>Scratchpad Window</i> | 12 |
| <i>History Window</i> | 12 |
| <i>Mouse and Computer Keyboard</i> | 12 |
| 4. Tuning Entries | 12 |
| <i>Basic Entries</i> | 13 |
| <i>Result Range</i> | 13 |
| <i>Operators</i> | 13 |
| <i>Entry Fields</i> | 14 |
| <i>About Octave Numbers</i> | 15 |
| <i>Constants Basics</i> | 15 |

| | |
|---|-----------|
| <i>Code Snippets Basics</i> | 15 |
| <i>Parsing</i> | 16 |
| <i>Example Entry</i> | 16 |
| <i>Simplifying Entries</i> | 17 |
| 5. Global Parameters | 17 |
| <i>Position of A, and A4 Hz</i> | 17 |
| <i>Key and Fifths on the Circle</i> | 18 |
| 6. Shapes | 20 |
| <i>Shape Groups</i> | 20 |
| <i>Individual Shapes</i> | 21 |
| <i>Drawing and Playing Shapes</i> | 21 |
| <i>Holding and Inverting Shapes</i> | 21 |
| 7. Constants | 22 |
| 8. Functions | 23 |
| 9. Algorithms | 25 |
| Credits | 28 |
| APPENDIX | 29 |
| <i>Language Reference</i> | 29 |

1. Installation

TPXE requires a minimum screen resolution of 1024 x 768, and is best viewed at this resolution.

Mac OSX

TPXE does not require a MIDI interface; however, to upload tunings to a Tonal Plexus you must have a MIDI interface and have properly configured your equipment in Audio MIDI Setup, which is in the following location:

Your Hard Drive > Applications > Utilities > Audio MIDI Setup

It's a good idea to drag the Audio MIDI Setup icon to your dock so that you can easily access this application, as you will need to do so periodically when working with any audio software.

The Mac version of TPXE will load Soundfont .sf2 files located in either of the following locations:

Your Hard Drive > Users > Your User Name > Library > Audio > Sounds > Banks

Your Hard Drive > Library > Audio > Sounds > Banks

Windows

To have proper display of note names in TPXE, the ARIELL font must be in the Fonts folder. TPXE will attempt an installation at startup, but this may cause an error. If installation is unsuccessful, simply drop the file into the Fonts folder, which is in this location:

My Computer > Control Panel > Fonts

TPXE does not require a MIDI interface; however, to upload tunings to a Tonal Plexus you must have a MIDI interface and have properly configured your equipment. Visit your MIDI setup here:

My Computer > Control Panel > Sounds and Audio Devices > Audio

2. Basics

Built-in Help

TPXE has built-in help windows for all new features. Most, but currently not all, of the information given here is duplicated there. Click the question mark icon to open Help.

Bug Reporting and Feature Requests

Should you experience a crash, a copy of the file is saved for recovery and a bug reporting dialog appears. You can also use the menu item > *Report Bug* at any time. A list of reported bugs and feature requests is maintained online at <http://hpi.zentral.zone/reports/tpxe>.

Programming a Tuning

Tunings created with TPXE must be uploaded to TPX to be used. Follow these steps:

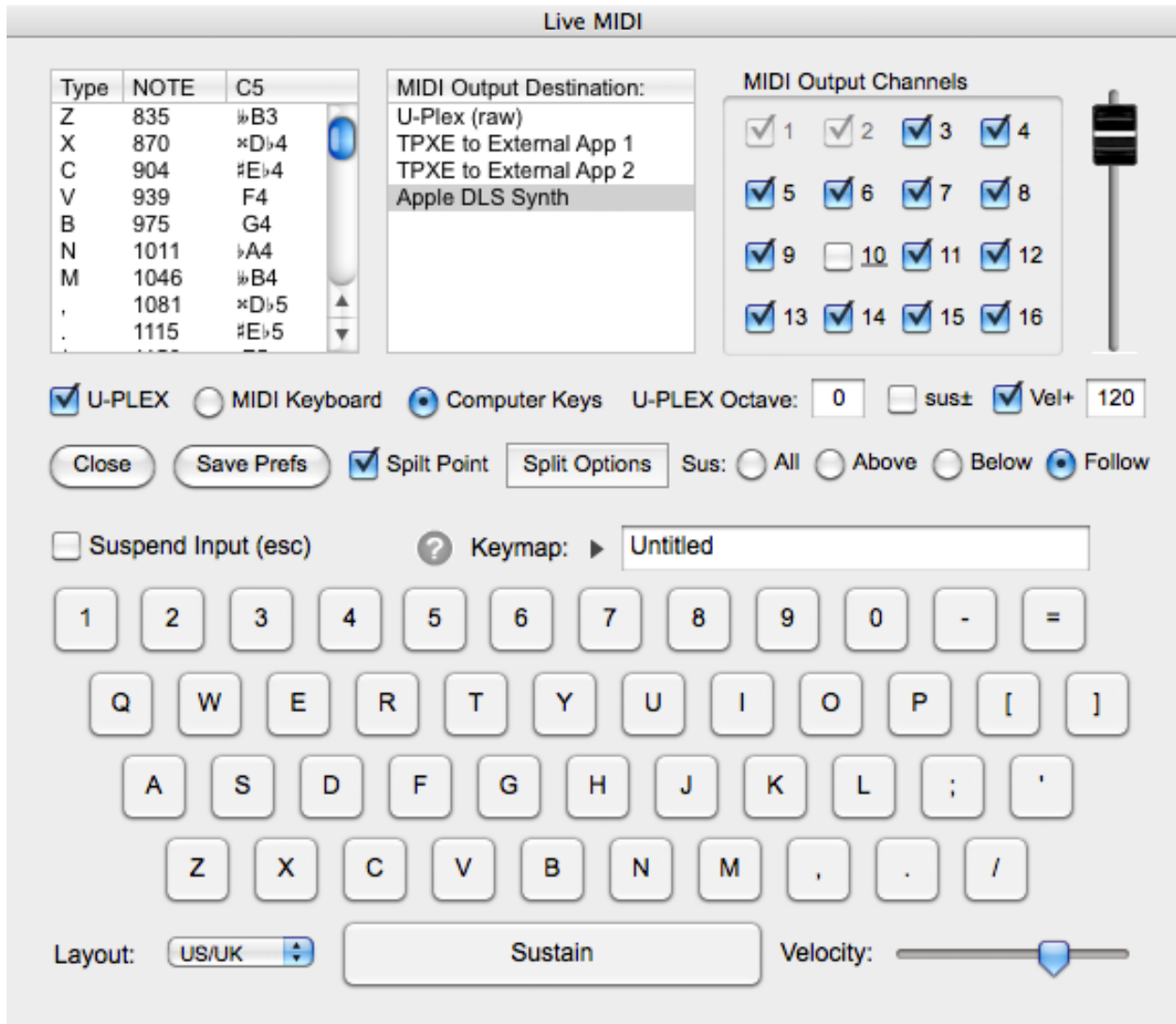
1. connect your MIDI interface to your computer, have it properly installed, etc.
2. connect MIDI OUT from the interface to MIDI IN on TPX
3. boot TPXE and load your tuning
4. choose menu item MIDI or type Command M and select the interface as the output port
5. choose menu item TonalPlexus and choose Program, or type Command P
6. select the options you want and program the tuning. TPX will display PROGRAMMING

NOTE: if MIDI devices are disconnected or newly connected while TPXE is running, choose MIDI > MIDI Routing and click Rescan MIDI.

Updating Files

Previous file formats can be loaded in the old format and saved in the new one. A batch updater is available to simplify this process. Use the menu File > Batch Update Files. Some of the previous formats did not store some important information that is now stored in the new format. In particular, files in older formats which used global Keys other than C will need to have the global Key reset manually.

Live MIDI Input



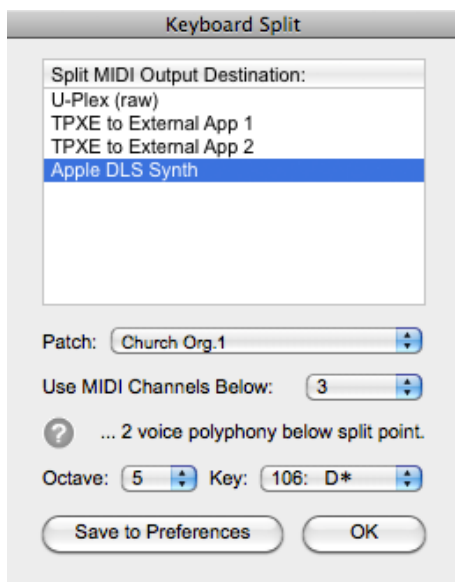
The menu item MIDI > Live Input opens a window which allows you to use TPXE as a virtual live Tonal Plexus, routing incoming and outgoing MIDI to and from external applications using TPX (connect UNTUNED OUTPUT to the MIDI Interface), or U-PLEX, or the computer keyboard as a virtual MIDI keyboard. Mac Users will see virtual ports in the MIDI connections lists. Windows users must install the free utility MIDI Yoke in order to see virtual MIDI connections. <www.midiox.com/myoke.htm>

NOTE: Only registered TPX and U-PLEX owners can unlock this feature. An internet connection must be available to unlock the feature. Have your warranty card or serial number handy.

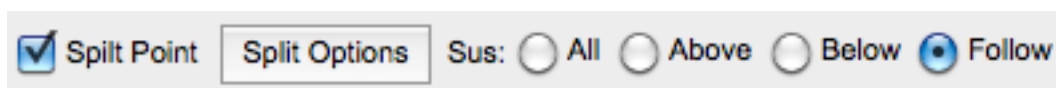
U-PLEX Owners

TPXE acts as both a tuning editor and as a realtime tuning performance engine for U-PLEX keyboards. Simply open the Live Input window to play and reroute MIDI data. Velocity and octave controls are included, and other features can be added according to customer requests. For example, in version 1.5 Split Keyboard functions were added, by request.

Split Keyboard



Live Input can route MIDI to two output destinations using the Split Keyboard option. A Split Key, number (1-211) within an Octave (1-8) is chosen as a Split Point. Including that key and above, the output will be sent to whatever is chosen in the Live Input window. The parameters in the Split window apply to all of the keys *below* the Split Point, including patch, channels and MIDI output destination.



Open the Split window by clicking **Split Options**. Set the sustain pedal behavior in the Live Input window by selecting a radio button. The sustain pedal can be set to affect all the keys, only keys above or below the split, or it can follow your playing. When **Follow** is selected, the sustain pedal will follow the last keys you have played; for example, if you are playing keys below the split, then when you press the sustain pedal, it affects only the keys below the split; likewise if you are playing above the split, the sustain pedal affects only above the split. This is available so that drones can be allowed to sound above or below the split, while the sustain pedal is still useable in the region you are currently playing in. You can turn ON the sustain in one region, and then move away from it, and the sustain follows your playing.

Using the Kontakt Script

In the TPXE folder is Kontakt a script file called *H-Pi Universal Microtuner.nkp* Using this script, Kontakt gives polyphonic microtonal output for any Kontakt instrument without the need for exporting any tuning tables to Kontakt. Live Input can be routed from TPXE to Kontakt, or Kontakt can be controlled by a Tonal Plexus TPX keyboard sending TUNED MIDI Output. A few things in Kontakt must be configured for this this work properly.

7.To route MIDI from TPXE to Kontakt, the correct MIDI port must be selected as the input for the instrument in Kontakt .If OMNI is on, the output will be incorrect. On Mac, the virtual port ic called *TPXE to External App*. On Windows, MidiYoke must be installed, and you select which MidiYoke port to connect between TPXE and Kontakt.

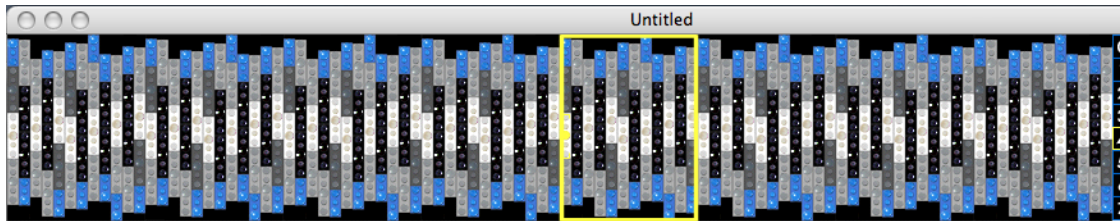
8.For correct tuning, the Kontakt instrument must have its pitch bend range set to -1.00 and + 1.00 semitone.

3. Interface Basics

Custom Colors

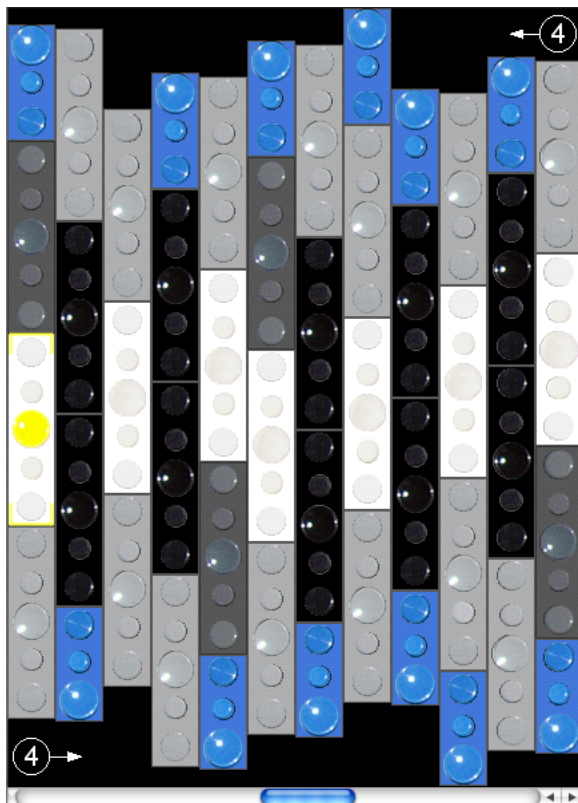
Open the Preferences window to customize keyboard keys and selection colors.

8-Octave View



A bird's eye view of the entire available 8-octave range of all Tonal Plexus keyboards is shown at the top of the window. This area allows selection of key groups and areas across any range of the keyboard. A gold box shows a 1-octave area of this view which is magnified in the scrolling octave view below.

Scrolling Octave View



This close-up view of a 1-octave portion of the keyboard corresponds to the position of the gold box in the 8-octave view. All editing features are the same as the 8-octave view. Note that arrow keys can be used to navigate the keyboard in both 8 and 1-octave views.

Keys List View

| KEY | OCTAVE | NAME | TUNING FORMULA | CENTS | C4 | MIDI | 12ET ± ¢ |
|-----|--------|------|----------------|-----------|----|------|----------|
| 845 | 4-1 | ♭~C | (199,205) | ▶ 1159.02 | 4 | 60 | C-41.0 |
| 846 | 4-2 | ♭~C | (200,205) | ▶ 1164.88 | 4 | 60 | C-35.1 |
| 847 | 4-3 | ~C | (201,205) | ▶ 1170.73 | 4 | 60 | C-29.3 |
| 848 | 4-4 | ♯~C | (202,205) | ▶ 1176.59 | 4 | 60 | C-23.4 |
| 849 | 4-5 | ×~C | (203,205) | ▶ 1182.44 | 4 | 60 | C-17.6 |
| 850 | 4-6 | ♭C | (204,205) | ▶ 1188.29 | 4 | 60 | C-11.7 |
| 851 | 4-7 | ♭C | (205,205) | ▶ 1194.15 | 4 | 60 | C-5.9 |
| 852 | 4-8 | C | (1,205) | ▶ 0.00 | 4 | 60 | C |
| 853 | 4-9 | ♯C | (2,205) | ▶ 5.85 | 4 | 60 | C+5.9 |
| 854 | 4-10 | ×C | (3,205) | ▶ 11.71 | 4 | 60 | C+11.7 |
| 855 | 4-11 | ♭+C | (4,205) | ▶ 17.56 | 4 | 60 | C+17.6 |
| 856 | 4-12 | ♭+C | (5,205) | ▶ 23.41 | 4 | 60 | C+23.4 |
| 857 | 4-13 | +C | (6,205) | ▶ 29.27 | 4 | 60 | C+29.3 |
| 858 | 4-14 | ♯+C | (7,205) | ▶ 35.12 | 4 | 60 | C+35.1 |
| 859 | 4-15 | ×+C | (8,205) | ▶ 40.98 | 4 | 60 | C+41.0 |

All tuning entries for every key are listed to the right of the scrolling octave view. Note that the arrow keys can be used to navigate the list, but the direction of up and down are reversed order as compared to the 8 and 1-octave views. This is because the keys list is numbered from top to bottom, and the pitches thus rise from top to bottom. If this seems confusing, consider that listing from top to bottom is the standard way to list anything which is ordered, and Western musical scales have been listed this way since the first recorded scales of ancient Greece.

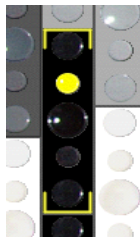
Region Map List

| | | | |
|--------------------------|-----|----|------------|
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |
| <input type="checkbox"/> | ↑ ↓ | 41 | ▶ Untitled |

A small list to the right of the 8-octave view is used to keep track of octave Region Maps, which are templates allowing various configurations of regions within an octave (from a keyboard C to C). The default region map is 41, corresponding to the 41 colored regions of the keyboard. Click the arrows in this list or type in a new number to change the number of regions in a given octave, then move region boundaries around. This is useful for programming tunings in which groups of keys are tuned to the

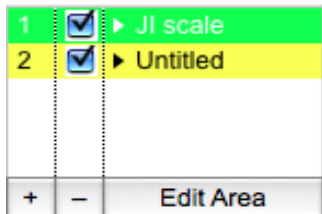
same pitch, or exhibit some kind of smaller structure within an octave. Region Maps can be named, saved and opened. All octaves with the same region map can be updated at once.

Editing Region Boundaries

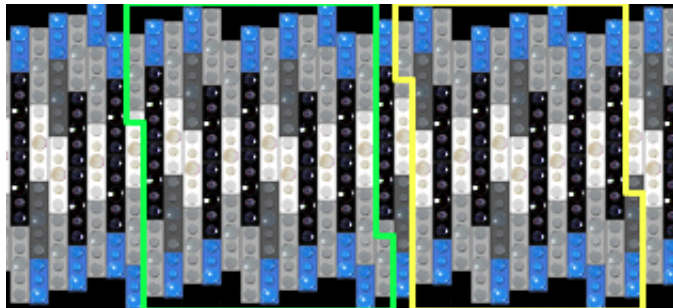


When a key is clicked in either keyboard view, its region boundaries are shown in gold brackets. These brackets may be repositioned, deleted or added by using the mouse and keyboard. Drag to reposition, press X to toggle the mouse cursor to delete, and press I (capital letter i as in index) to insert a new region boundary. Press N to return to a normal cursor.

Areas List

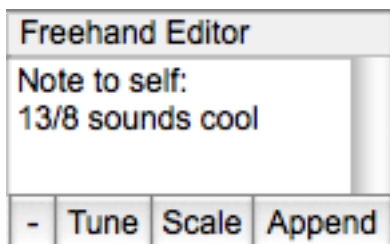


You may want some arbitrary areas of the keyboard tuned one way and other areas tuned another way. You can define areas to make this easier. When an area is defined, a colored line is drawn around the area on the birds eye view. For example, the two arbitrary areas above are shown below.



A list below the Region Map list shows defined areas of the keyboard, and these can be shown or hidden by using the check boxes. Each area can be described and is given a color code. On the birds eye keyboard, lick a low key and then Shift-click a higher key to select the area, and then click the [+] button in the Area list to define the area.

Freehand Field



The freehand field is a text edit field which allows you to type any text you want to cut and paste. It is most useful for cutting and pasting text from other programs into TPXE to create scales and tuning entries. Chapter 6 gives details on tuning entries.

Scale List

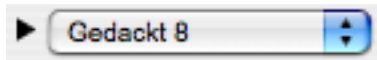
| Scale | Description |
|--------|------------------|
| 08-11 | ▶ 8 out of 11-ET |
| abell4 | ▶ RossAbells ... |

.....

| | | | |
|---|---|--------|------|
| + | - | Import | Edit |
|---|---|--------|------|

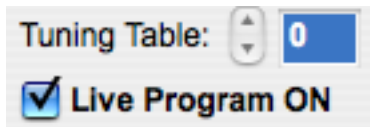
Below the Area list is a list of imported scales. You may want to use more than one Scala file to create your TPX tuning. Click Import to add a .scl file to the list, or create a new scale from scratch, or transfer the contents of the Freehand field using the Scale button. Once scales are in the list, they can be tuned to the keyboard by selecting an area and choosing Action > Tune Scale.

Instrument List



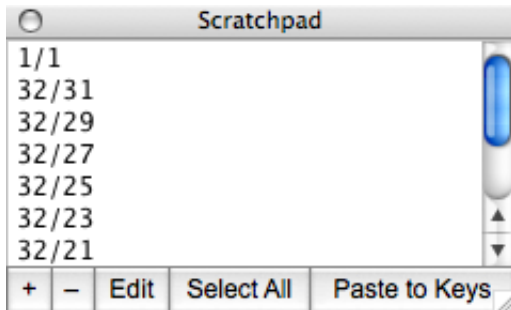
Select a playback instrument from the instrument list. On PC, this is a GM sound set, but output can be sent to any external port if MidiYoke is installed. On Mac, this list contains Quicktime Instruments plus any soundfonts located in ~ Library > Audio > Sounds > Banks. The soundfont must have the extension .sf2. On Mac, there is a triangle next to the popup menu as shown above, which when clicked opens a window giving control of Mac OSX Synthesizer parameters, including the sample set, patch, default duration and velocity for automatic notes, and AU parameters Cents Offset, Reverb, and Volume.

Live Programming



Choose a table number and turn on live programming if you want your tuning changes to take place immediately and destructively in TPX memory. This is only recommended for tuning changes involving a few keys, and should be used with caution. For this reason, the indicator flashes when this option is ON.

Scratchpad Window



When the cursor is not in the Freehand field, copying and pasting is done using the Scratchpad window, which can only contain valid tuning entries. Think of it as a dedicated tuning clipboard.

History Window

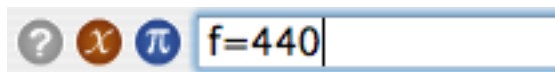
| No. | C4 | Description |
|-----|----|----------------------------------|
| 1. | 0 | Open Tonal Plexus Editor |
| 2. | 1 | Octave 1 Region 20 start dragged |
| 3. | 1 | Octave 1 Region 23 start dragged |
| 4. | 1 | Octave 1 Region 23 start dragged |

Many user actions are stored in a history list, which is viewable in the History Window. Actions can be undone with Command-Z or previous states can be accessed by clicking on the history list.

Mouse and Computer Keyboard

Click a row in the Key List or a key in either Octave View to select a key. Shift Click selects a contiguous range of keys. Option (Alt) Click selects a key in every octave. Control Click selects all keys within a region. Command click selects a discontinuous range. Up and down arrows navigate the keys when a single key has been selected. Note that these will work differently depending on whether you are working in the Key List or in an Octave View. Spacebar toggles Sustain Mode ON / OFF, useful for testing out chords. A complete list of Command key shortcuts and menu items is in the Appendix.

4. Tuning Entries



A tuning entry is an expression which represents a tone in a number of different forms as follows.

Basic Entries

Decimal values such as 1.2365256341287

Interval Ratios a:b, where $a < b$, such as (2:3)

Tone Ratios a/b, where $a > b$, such as (3/2)

Degrees of an ET, or equal division of the octave, in set notation, where the first scale degree is numbered 1, such as the second scale degree of twelve tones per octave (2,12)

Exponents, such as $(3^{7/2^4})$ or the ratio of 3 to the 7th power to 2 to the 4th power.

+ and - Units values (default unit is Cents).

Frequency values in Hz, such as $f=261.6255653$, which may also be entered in the form $f=<CodeSnippet>$ (see below). The available frequency range is 7.9430 Hz to 12911.4169 Hz.

Result Range

Tuning entries automatically place the resulting value of the entry (x) within the range $1 \leq x < 2$, that is, within an octave. If you want to return a value which is not automatically placed within this range, you can add the character @ to the start of the entry. This spiral-like character tells the parser that the tone is allowed to spiral past the octave threshold. For example, the entry (3/1) returns 1.5, the justified value of $(3/1) = (3/2)$, while the entry @(3/1) returns 3. The example using exponents given above $(3^{7/2^4})$ could just as well be written $(3^{7/2})$.

Operators

+ and - are used for units values, such as +35.2 or -4.3, used alone or placed AFTER any other expression. For example, (1,12)+3 adds 3 units to the first degree of 12ET. (7:8)-5.3 subtracts 5.3 units from the ratio 7:8. The tuning value depends on currently selected Tuning Unit. The default unit is Cents (1200 Units, 2/1 Ruler).

NOTE: + and - cents must always be at the END (right side) of the expression.

You will see that the transpose function does this automatically.

Multiplication and division mean transposition up or down respectively, for example $(1.232) \cdot (2:3)$ is the decimal fraction 1.232 transposed up by the interval 2:3. for example $(4/3) / (1.112)$ is the tone 4/3 transposed down by the decimal fraction 1.232. Note that the / between the segments means 'transposed down by' while the / in the 4/3 segment does not

indicate transposition. Parentheses must be used when there are 2 terms and some operator, as shown above. The command is parsed according to the segments which are separated by an operator such as * or /. Parentheses define segments and cannot be nested, except inside code snippets (see below).

Entry Fields



There are several places to type in tuning entries. The first is the **Main entry field**, which has three small circles directly at its left: (?) (x) and (pi), which stand for Help, Functions, and Constants, respectively. Use the (x) and (pi) circles to add known functions and constants to the main entry field, and then replace the variables with the values you want to use. The Main entry field restricts keystrokes only to allowable keys in allowable situations. It may be thought of as the safest way to make a tuning entry. Using this field it is also possible to tune a range of keys to the same value.

Cutting and pasting using either the Main or Keyboard fields uses a special clipboard called the **Scratchpad**, which appears as a floating window. Tuning entries can be copied and pasted from this window without affecting the contents of the computer's clipboard, which may hold text from anywhere, including other applications. The Scratchpad is a kind of safe clipboard which only allows valid tuning entries to be listed.

In contrast to these safe methods of entry, another possibility is the **Freehand field**. When the cursor is active in this field, the menu choices change to standard text editing options only. The purpose of this field is to provide the familiar feel of a standard text-editor which is free from the safeguards of the other entry fields. Multiple entries may be input in a quick list using return or enter, and text from this field can be copied, cut and pasted to and from other applications. The text can be turned into a Scale and added to the Scale List, and can be copied to the Scratchpad. Note that characters typed in this field can be anything, not just valid tuning entries. So, it can also be used just to jot down random thoughts.

About Octave Numbers

H-Pi software uses the C based octave numbering system, where middle C is called C4. In terms of tuning, the octave boundaries exist not at C+0 cents to C+1199 cents, but at C-50 cents to C+1149 cents. This results from sample mapping logic which is centered on 12ET pitches, bending samples up or down within the narrow range of a quartertone in order for the sound to remain as close as possible to the original sample. Because of the quartertone offset, tuning entries for pitches around the boundaries may result in octave assignments an octave higher or lower than desired. To get the octave you want, just change the octave number using Command-' or Command-/. You can also precede entries with the @ character to allow them to cycle past the octave limit without changing the octave number.

Constants Basics

Constants are decimal values like {pi} and {e} or {yourconstant} which can be plugged in anywhere in a tuning entry. see the section on Constants for more information.

Functions Basics

Functions are user programmable variable calculators like YOURFUNCTION[x] which can be part of a tuning entry as any other segment. See the section on Functions for more information.

Code Snippets Basics

A code snippet is an entry segment which is a line of BASIC computer programming code enclosed in angle brackets, as in <Your Code>. The code is processed as the statement z = <Your Code>, where z can be either a decimal (x) or a string (s), and the value of the segment returned by your code is z. You can think of this like a fancy calculator with all the expected math functions as well as string functions and built-in tuning functions. A standard language reference for code snippets is found in the Appendix.

Of the many uses for code snippets, some basic uses are:

1. Perform calculations with tuning functions and nested parentheses directly in a single statement, such as: `<CentsToDecimal(round(1200*log2(46573))-34.6)>`

2. Dynamically link a key to an imported scale or scales. For example, you have imported a scale (Scala) file called FirstScale. You can link a key to, say, the third tone in this scale, with the code snippet `<scX("FirstScale", 3)>` Then if you want to edit your scale and change the third tone, the key updates to the new tone without having to copy and paste in a new value.

3. Combine calculations with scales in something like `<round(310*log2(scX("FirstScale")))>`, or even `<round(310*log2(scX("FirstScale", 3)/scX("SecondScale", 4))>`, etc.

Parsing

Tuning entries are parsed as follows:

1. Constants {c} are replaced with actual values
2. Code Snippets are processed and each `<CodeSnippet>` is replaced with an actual value
3. Functions THISFUNCTION[x] are processed and replaced with actual values.
4. The remaining basic entry segments are processed and replaced with actual values.
5. All of the values are calculated according to the operators used between the segments, and the final value is returned.

Example Entry



This is the 4th degree of 13ET transposed down by 11/8, then transposed up by 9/8, and then transposed down by 23 cents. Up to 32 operations can be strung together for each key.

NOTE: *Parentheses must be used when there are 2 terms and some operator, as shown above. The command is parsed according to the segments which separated by an operator such as * or /.*

Simplifying Entries

Entries with multiple ratios can be simplified using the menu option Action > Simplify Ratios, or using Command =. The tuning formula is searched for ratios, which may be anywhere and a mixture of a:b

and a/b types. The ratios are multiplied or divided together and then reduced. The example tuning entry given above, $(4,13)/(11/8)*(9/8)-23.4$, simplifies to: $(4,13)/(99/64)-23.4$

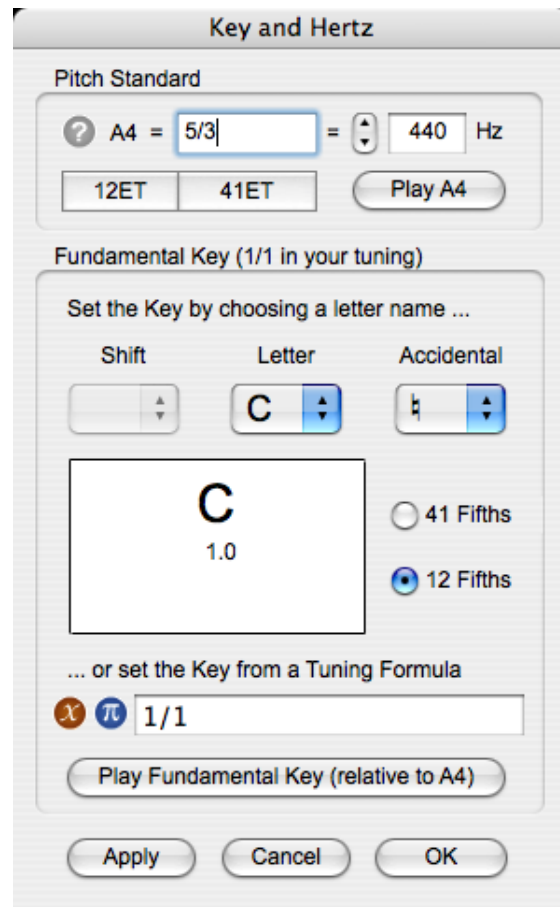
5. Global Parameters

The actual pitch resulting from a tuning entry depends on four parameters in addition to the entry itself. These are:

- the Hertz standard for the note called A4
- the exact position within an octave of the note called A
- the size of the circle of fifths (12 or 41 fifths)
- the global Key

Position of A, and A4 Hz

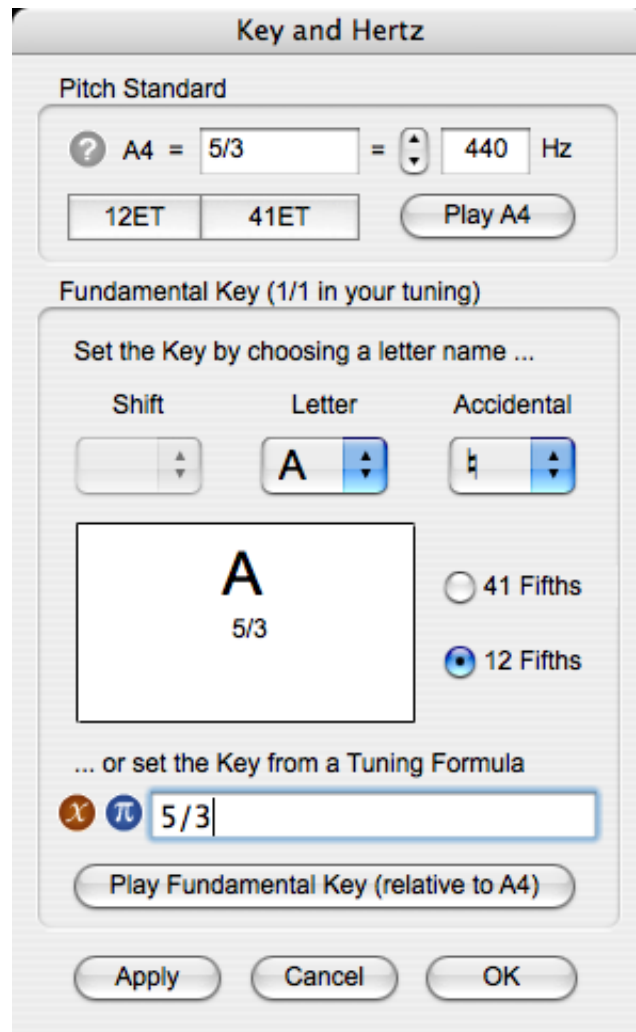
Two pitch defaults in Western musical history are C, as a 'Major Key' and A, as a 'Tuning Pitch'. H-Pi software uses these defaults. The default Key of 1/1 or 1.0 is C. Because C is 1/1, the exact position of A within an octave must be specified in order to place the pitch of C. By default, A is the tenth tone of 12ET = (10,12) = 1.6817928305 ... , and the Hertz standard for A4 is 440 Hz. If you want the relationship between the pitches called C and A to be something other than that of 12ET, then you must set the absolute position of A to what you want. For example, many people prefer A to be located at 5/3 from C. To do this, you would open the Key and Hz window and type 5/3 in the field defining the absolute position of A, as shown below.



NOTE: the Hertz value of A also depends on the global pitch settings of the synthesizer. For the Hertz setting of .cse files to work correctly, the synthesizer must have its A4 set to 440 Hz.

Key and Fifths on the Circle

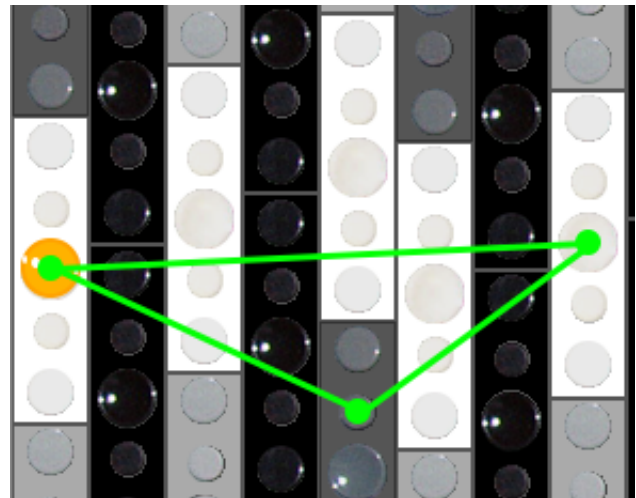
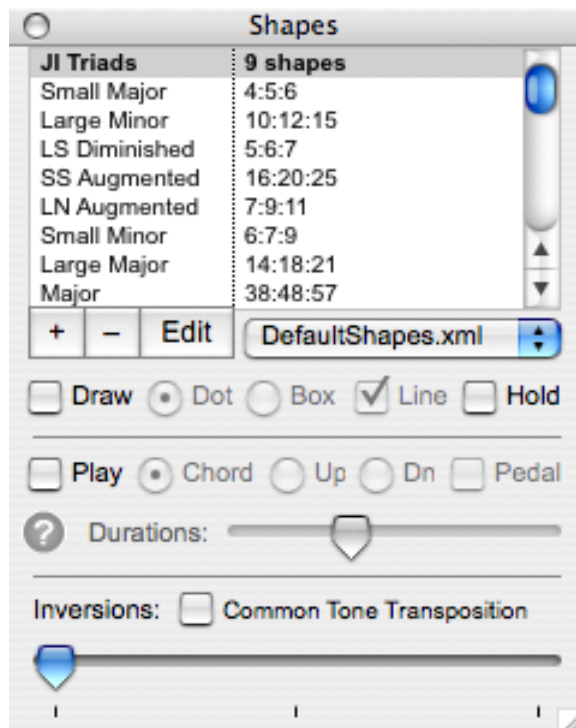
The global Key determines what pitch is 1/1 in your tuning, according to 12 or 41 fifths on a circle, sounding relative to the exact position of A within the octave, and the Hz value of A4. The note name of the global Key can be either a 41ET note name or a 12ET note name. Sounding pitches are always displayed both as a 41ET note name (in column 7), and as a 12ET MIDI note name (in the rightmost 2 columns). For example, if you want your 1/1 to be the pitch A, then you can type in the tuning entry for the absolute position of A in the Tuning Formula field, as shown below:



The result of the above settings will be a Key of A where the tone 6/5 will be +C. If you wanted the tone 6/5 to be notated as C instead of +C, then the absolute position of A should not be 5/3, but instead the default (10,12) or the 41ET version (31,41). If you choose either of these, make sure that you choose the corresponding 12 or 41 fifths radio-button, or some note names may be incorrect.

By default the Key is C, number of fifths on the circle is 12, and the position of A is (10,12), so the tuning entry 1/1 results in a sounding pitch of $C \pm 0\text{¢}$. If the position of A is changed to something other than (10,12), then the sounding pitch for C will not be $C \pm 0\text{¢}$. For example, if A is (32,41), then in the Key of C, 1/1 will result in a sounding pitch of $C -7.3\text{¢}$, because A (32,41) is 7.3¢ higher than A (10,12), and the Hz value of A4 is fixed to the same frequency in both situations.

6. Shapes



Chord and scale structures can be displayed and played back on the keyboard using dots, lines, and boxed labels for each key in the shape. The shape window shows a list of existing shapes from .xml files stored in the location: Preferences > H-Pi Instruments > TPXE > Shapes. Shape files can be constructed in a text editor and added to this directory, or new files can be created within TPXE. Any files in the directory are listed in the popup menu, along with options for displaying, embedding, saving, and deleting shape files. Each file can be loaded separately, or the option 'Show entire library' can be used to load all shapes from all files at once. A default set of shapes is provided, consisting of several conventional harmonic categories: triads, seventh chords, etc.

Shape Groups

Since some shapes are likely to share similar characteristics, shapes are categorized first by a shape group, which are shown as bold text rows in the list, with the number of shapes in the group noted in the second column. To add a new group, select any existing shape in the shape list and click the + button. A new line appears allowing you to name the new group. Once a new group is created, new shapes can be added to the group.

Individual Shapes

Each shape is given a name and a set of labels for the keys comprising the shape (called 'nodes'), to be displayed when 'Boxes' is selected. The labels appear in the second column of the shape list, as a colon (:) separated list. To add a new shape to a group, select the group row (bold text) and click the + button, and follow the directions given by the dialog windows. Shapes are easily defined simply by clicking on the keys you want to define the shape, and then you can label each key if you want to.

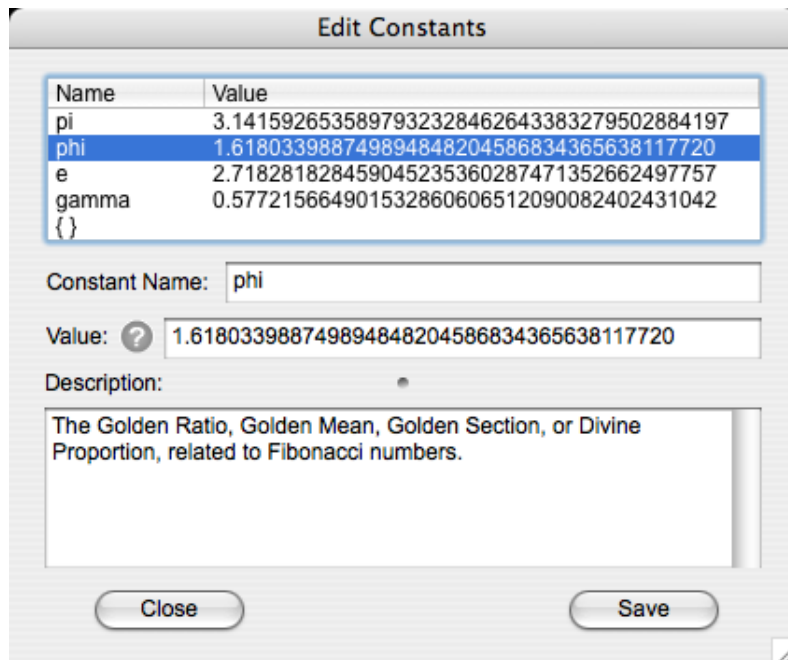
Drawing and Playing Shapes

Shapes can be drawn using dots or boxes with labels, and the keys can be joined with lines. The shape is drawn starting from the currently selected key, so clicking around on either keyboard view transposes the shape anywhere on the keyboard. Shapes can be played as a chord, or as a scale or arpeggio, with or without the sustain pedal on. The durations of each pitch are controlled by the Duration slider.

Holding and Inverting Shapes

The 'Hold' option makes the shape stationary so that clicking around on the keyboard does not transpose the shape, but instead mixes the selected key with the shape, so changes to the shape can be tested. Moving the Inversions slider inverts or rotates the shape to begin on any of its nodes, as in the traditional idea of harmonic inversion, where for example a triad has a root position, first inversion, and second inversion. The selected key acts as a root, and inversions are calculated from that root, or the selected key can be turned into a fixed common tone node for transposed inversions by checking the 'Common Tone Transposition' option.

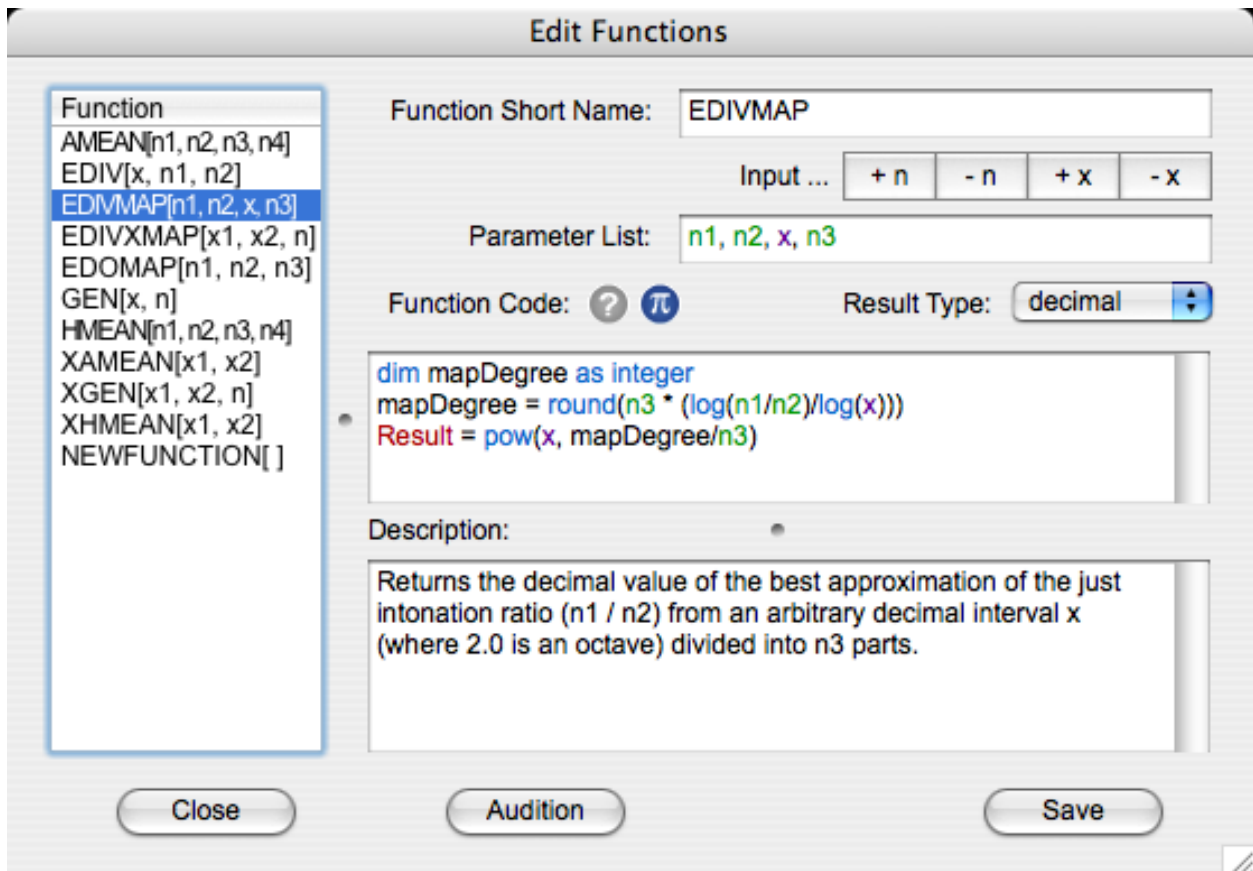
7. Constants



Constants are named double precision values. A constant name consists of lowercase letters only with no spaces, surrounded by curly brackets, as in {yourconstant}, and a constant value may be any positive number between 0 and 1.7976931348623157e+308. Note that scientific notation is not recognized.

Constants can be used in Tuning Formulas, such as {pi}/(3/2), which would be the interval 3.14159... transposed down by the interval 3/2. Constants can also appear in Functions and Algorithms. For more information, see the Function and Algorithm editors.

8. Functions



Functions are user programmable single key tuning structures which perform computations on any number of input values to return a single output value. The input values (Parameters) can be positive integers (whole numbers) or decimals (double precision numbers), and the output value (**Result**) may be an integer or a decimal.

Functions are named using uppercase letters only with no spaces, such as YOURFUNCTION, followed by a comma separated list of parameters enclosed in brackets, such as [n1, n2, x1, x2] called a Parameter List. Note that brackets distinguish Functions from other tuning operations which use parentheses. The name and parameter list together designate a function. The example would appear as YOURFUNCTION[n1, n2, x1, x2]. Each n in the Parameter List is an integer, and each x is a decimal. You assign the order and the data type of each parameter in the list, and you determine how the parameters are used to compute the output using the Function Editor.

Function instructions (entered in the Function Code editing field) must conform to the list of BASIC programming language structures outlined in the Function Help hierarchical popup menu. Rudimentary knowledge of computer programming is helpful; non-programmers should study the examples and learn by exploring. In order to work properly, the function must set the value of **Result**, with **Result** = ..., for example:

Result = n1/n2

Constants can be used in the function code, using the Constants popup menu. See the Constants Editor for more information.

Functions may be tested using Audition, which provides dynamic control of each variable in real time. The results of Auditioning a function may be appended to a list, copied to the Scratchpad or Clipboard, and pasted into your tuning.

Once a function is finished, it is usable as part of any Tuning Formula. Functions are parsed independently and may be combined with other tuning operations. For example, the function EDIV[x, n1, n2] may be used in a tuning entry such as (9/8)*(3,17) to create something like:

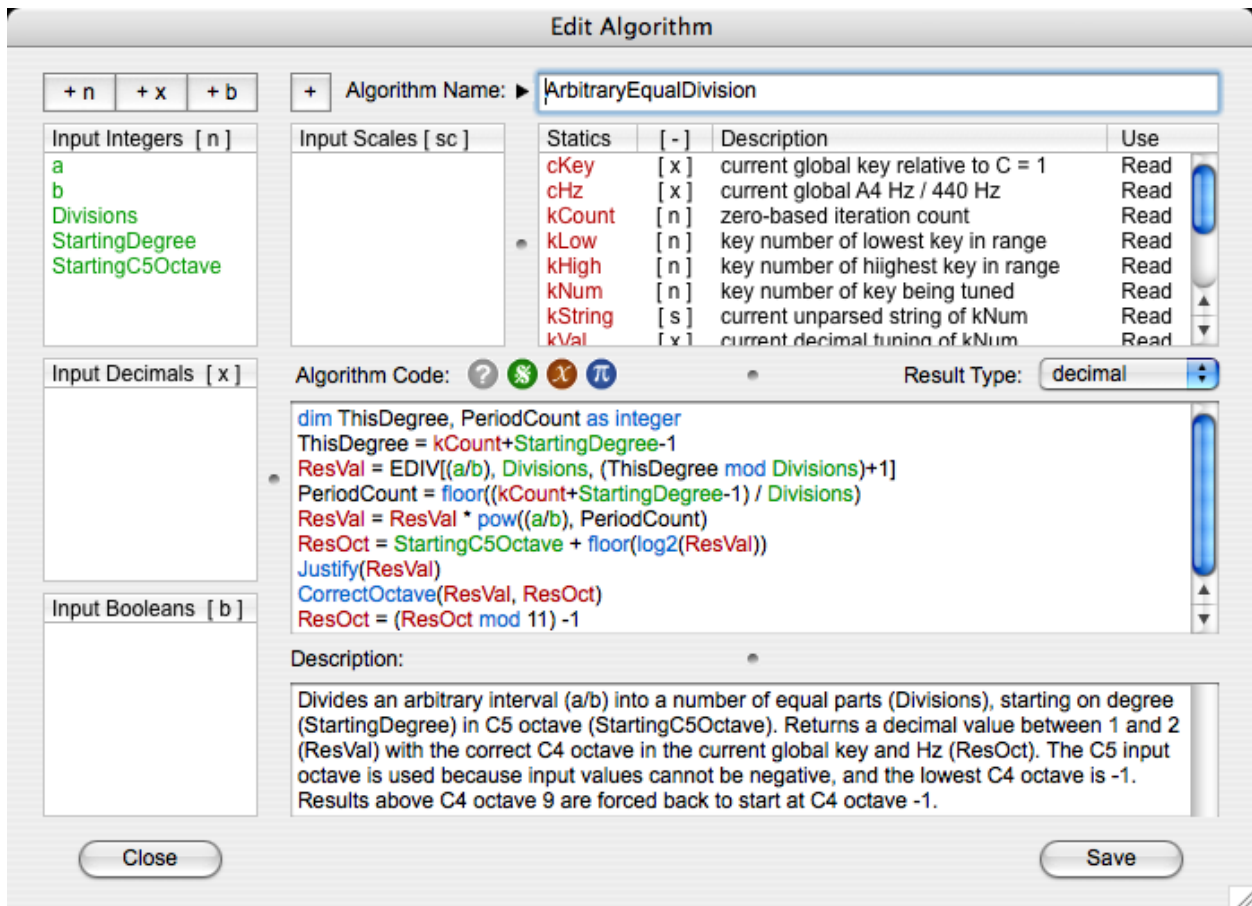
(9/8)*EDIV[3, 13, 4]*(3,17)

This would be (9/8) transposed up by the 4th degree of an equal division of 3 into 13 parts (the Bohlen-Pierce scale), transposed up again by the 3rd degree of 17ET. Further, any function variable which is an x (a decimal) may also be entered as a tuning operation itself, such as EDIV[(13/8), 5, 2]. This would be the 2nd degree of an equal division of the interval 13/8 into 5 parts. Constants may also be entered as variables, using the Constants popup menu, giving possibilities such as EDIV[{pi}, 12, 7], which would be the 7th degree of an equal division of the interval pi (3.14159...) into 12 parts. See the Constants Editor for more information.

Functions may be used iteratively and their functionality further expanded using higher level structures called Algorithms.

Function files are stored in the H-Pi Instruments folder inside the Preferences folder.

9. Algorithms



Algorithms are user programmable tuning structures which perform computations on any number of input values to return multiple output values for a range of keys. The input values (Parameters) can be positive integers (whole numbers), decimals (double precision numbers), or boolean values (true or false). After input parameters are passed to an algorithm, the algorithm may be called any number of times (iterations) using those input parameters, so that a range of keys may be tuned through a process.

Algorithms are named using both uppercase and lowercase letters, as in `YourAlgorithm`, followed by a comma separated list of parameters, named with any combination of letters and numbers, enclosed in angle brackets, such as `<FirstParameter, SecondParameter, Your3rdParameter>`. You assign the names and data types of each parameter, but the order of their appearance in the parameter list is automatic; from left to right, the order is first the integers, then the decimals, and lastly the booleans.

To give multiple unique results, a variety of static values (Statics) are available for algorithm computations. Two decimal statics provide global information about the current key (**cKey**) and current A4 Hertz (**cHz**). The static **cKey** is a ratio of the current key to C where C = 1. The static **cHz** is a ratio of the current A4 Hz value to A4 = 440. For example, you may have chosen a key of D = 9/8 and A4 = 445 Hz. In this case, **cKey** will be $9/8 = 1.125$ and **cHz** will be $445/440 = 1.0113636\dots$

For each iteration, there are two output values: the tone value (**ResVal**) which may be either a decimal or a string, and the C4 octave number (**ResOct**), which should be an integer between -1 and 9. **ResVal** as a string may be used to produce results including ratios or functions or any combination of normal tuning entries which can then be parsed as any other tuning entry. **ResVal** as a decimal will create tuning entries which are already decimals and do not need to be parsed.

There are two main statics which are used to make the results of each iteration unique. The first is called **kCount**, which keeps track of iterations beginning with 0 and incrementing with each iteration, and the other is called **kNum**, which is the actual key number of the key which will receive the results of an iteration. **kLow** and **kHigh** are the numbers of the lowest and highest keys in the currently selected range, which can be used to determine the size as well as the register of the range. Statics related to **kNum** are **kString**, **kVal** and **kOct**, useful for computing transformations in which the current string, decimal or octave values of a key are used in some way to determine the output value for that key.

When **ResVal** returns a decimal, it should have a value $1 \leq \text{ResVal} < 2$, meaning its value is within an octave (2/1). To ensure this range of values for **ResVal** or any other decimal you work with, you can use the built-in tuning function **Justify(x)**. If you are working with ratios, you can also use the function **JustifyRatio(n1, n2)** where the ratio is in the form $n1/n2$ where $n1 > n2$. Other useful tuning functions convert decimals to cents, or cents to decimals, and instead of changing the value passed, return new values so that you can assign one variable to the conversion of another, as in:

```
FirstParameter = 1.618033988749894848204586834365638117720
SecondParameter = 88 + DecimalToCents(FirstParameter)
```

$$\text{Your3rdParameter} = (3/2) * \text{CentsToDecimal}(\text{SecondParameter})$$

Assigning C4 octave numbers can be a somewhat tricky business, because the values of **cKey** and **cHz** change the octave break within a decimal based octave. To simplify the process of assigning C4 octaves to **ResOct**, within the algorithm octave numbers may be calculated as if in the key of C and A4=440 Hz (so that **cKey=1** and **cHz=1**). Then, before returning **ResOct**, you may use the tuning function **CorrectOctave(ResOct)** to automatically correct the octave number according to the current key and Hertz.

When **ResVal** returns a string, the string has to be in valid entry format; that is, it must be able to be parsed in the same way that a tuning entry is parsed. Since string manipulation is not the same as numerical calculation, computations should be performed using **kVal**, and you may need to introduce other variables into the code in order to work effectively with both **kString** and **kVal**. For an example of how to work with both strings and decimals, see the PrimeLimit, OddLimit, and OddHarmonics algorithms, which also show how to sort string arrays to match the order of a corresponding array of decimals.

Algorithm instructions (entered in the Algorithm Code editing field) must conform to the list of BASIC programming language structures outlined in the Algorithm Help hierarchical popup menu. Rudimentary knowledge of computer programming is helpful; non-programmers should study the examples and learn by experimentation. In order to work properly, the algorithm code must somewhere set the values of **ResVal** and **ResOct**, using **ResVal = ...** , and **ResOct = ...** , for example:

$$\begin{aligned} \text{ResVal} &= \text{kVal} * (\text{FirstParameter} / \text{Your3rdParameter}) * \text{cKey} * \text{cHz} \\ \text{ResOct} &= \text{floor}(\log_2(\text{ResVal})) \end{aligned}$$

Algorithm code may employ Functions and Constants using the popup menus in the editor. Algorithm parameters, statics and constants may be passed to Functions within the algorithm code. See the Function and Constant Editors for more information.

Algorithm code may also reference imported scales, using **scX(sc, n)** or **scS(sc, n)**. The Sign icon gives a list of scales in the editor. This should be used with caution, because if the algorithm is called but the scale referenced has not been imported, an error will occur.

When working with tuning entries formatted as Hz values, algorithms the return type of the algorithm must be a string, and the string must begin with the characters f= ... What follows after the equals is a Hz value. Hz frequency entries require that the decimal value and C4 octave number be calculated from the frequency. The tuning methods [HzEntry\(x\)](#) and [HzOctave\(x\)](#) should be used for this purpose. For example, the following code formats a result value as a correct frequency entry, and returns the correct C4 octave number, where **x** is a Hz value:

```
ResVal = "f=" + str(HzEntry(x))  
ResOct = HzOctave(x)
```

Algorithms may be tested (outside of the editor) using Audition, which returns results in a list. The results may be copied to the Scratchpad or Clipboard, and pasted into your tuning. It is possible that a coding error in your algorithm may cause the computer to hang or crash, but most errors are caught and displayed at runtime, to help you debug your code.

Algorithm files are stored in the H-Pi Instruments folder inside the Preferences folder.

Credits

All versions of TPXE are designed and programmed by Aaron Andrew Hunt, using [Xojo](#) and [MBS Plugins](#) on a [Mac](#).

This documentation was written by Aaron Andrew Hunt, using Apple [Pages](#).

Thank you for supporting H-Pi Instruments.

©2015 H-Pi Instruments · FOR THE FUTURE OF MUSIC

APPENDIX

Language Reference

The following relates to user programmable Functions, Algorithms, and entry Code Snippets. Below, an **n** represents an integer (whole number), an **x** represents a double precision (decimal) number, a **b** represents a Boolean (true or false) value, an **s** represents a string (text), and a **v** represents any of these types.

Operators:

| | |
|------------------|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| n1 Mod n2 | Modular arithmetic, the remainder of the division of n1 by n2 . |
| < | Less than |
| = | Equals |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| <> | Not equal to |
| And | Boolean AND |
| Not | Boolean NOT |
| Or | Boolean OR |

Comments:

'
//
REM

Data Types:

| | |
|-----------------------------------|--|
| Dim YourVariableName ... | Dimension a variable, including arrays |
| Redim YourVariableName ... | Redimension a variable, including arrays |
| as Integer | YourVariable is a whole number = n |
| as Single | YourVariable is a single precision decimal number = x |
| as Double | YourVariable is a double precision decimal number = x |
| as Boolean | YourVariable is a boolean (true or false) value = b |

`as String` YourVariable is a string (text) = s

Arrays:

- YourArray(`n`) An array of `n` number of values, dimensioned as any data type. Multidimensional arrays are YourArray(`n1`, `n2` ...)
- YourArray.append(`v`) Places a new element with value `v` at the end of an array. The data type of `v` must agree with YourArray's data type.
- YourArray.insert(`n`, `v`) Places a new element at the index `n` of an array. Previous element `n` becomes element `n+1`. The data type of `v` must agree with YourArray's data type.
- `v` = YourArray.pop Removes last element from YourArray and returns its value. The data type of `v` must agree with YourArray's data type.
- YourArray.remove(`n`) Removes element `n` from YourArray.
- YourArray.Sort Sorts YourArray in ascending order (one dimensional only).
- YourArray.SortWith(Array1, Array2, ...)
Sorts Array1, Array2 ... in the same order as YourArray.
- `n` = YourArray.IndexOf(`v`) Returns the index `n` of the element having the value `v`. The data type of `v` must agree with YourArray's data type.
- `n` = Ubound(YourArray) Returns the number of elements in the array -1.

Control Structures:

`Function...` A Method which returns a value.

```
Function YourMethod(v1 as DataType, v2 as DataType ... ) as DataType
...
Return v
```

`Sub...` A Method which does not return a value.

```
Sub YourMethod(v1 as DataType, v2 as DataType ... )
...
Return
```

`For... Next` A loop. Replace 0 and 127 with any integers in ascending

order. Use [DownTo](#) in place of [To](#) for descending loops.

```
For n = 0 To 127
    ...
Next n
```

[Exit](#) Exits a loop.

[If... Then... End If](#) Execute code blocks conditionally.
Replace [v1 = v2](#) with any condition.

```
If v1 = v2 Then
    ...
End If
```

[If... Then... Else... End If](#) Execute two code blocks conditionally.
Replace [v1 = v2](#) with any condition.

```
If v1 = v2 Then
    ...
Else
    ...
End If
```

[If... Then... Elseif... Then... Else... End If](#)

[If v1 = v2 Then](#) Execute any number of code blocks conditionally.

```
...
Elseif v1 = v3 Then
    ...
Else
    ...
End If
```

Replace [v1 = v2](#) and [v1 = v3](#) with any conditions.

[Do... Loop](#)

Repeatedly execute a code block until a condition is met.

Replace [v1 = v2](#) with any condition.

```
Do Until v1 = v2
    ...
Loop
```

Condition can be checked at the start or the end of the loop.

[Do](#)

...
 Loop Until v1 = v2

Select Case... End Select Execute code blocks based on a variable value.

Select Case v1
 Case v2 When v1 = v2, execute ...
 ...
 Case v3 When v1 = v3, execute ...
 ...
 End Select

While... Wend Repeatedly execute a code block while a condition is met.
 Replace v1 = v2 with any condition.

While v1 = v2
 ...
 Wend

Return Go back to the calling method.

Return v1 Return the value v1 back to the calling method.

able of contents

Numbers:

- x = Abs(x) Absolute value of x, forcing negative to positive.
- x = Acos(x) Arccosine of x. The angle with the cosine value x in radians.
- x = Asin(x) Arcsine of x. The angle with the sine value x in radians.
- x = Atan(x) Arctanget of x. The angle with the tangent value x in radians.
- x = Atan2(x1, x2) Arctanget of the point x1, x2, where x1 = x, and x2 = y.
- x = Ceil(x) Ceiling returns the value x rounded up to the nearest integer.
- x = Cos(x) Cosine of x, in radians.
- x = Exp(x) The constant e raised to the value x.
- n = FindGCD(n1, n2 ...) Greatest Common Divisor of a set of integers.
- n = FindLCM(n1, n2 ...) Lowest Common Multiple of a set of integers.
- x = Floor(x) The value x rounded down to the nearest integer.
- b = isNANorINF(x) Returns true if x is infinity or if x is not a number.
- x = Log(x) The base e logarithm of x.

| | |
|---|---|
| <code>x = Log2(x)</code> | The base 2 logarithm of <code>x</code> . |
| <code>x = Max(x1, x2 ...)</code> | The largest value from a set of numbers. |
| <code>x = Min(x1, x2 ...)</code> | The smallest value from a set of numbers. |
| <code>x = Pow(x1, x2)</code> | The number <code>x1</code> raised to the power of <code>x2</code> . |
| <code>x = Rnd</code> | Random value between 0 and 1. |
| <code>x = Round(x)</code> | The value <code>x</code> rounded to the nearest integer. |
| <code>x = Sin(x)</code> | Sine of <code>x</code> , in radians. |
| <code>x = Sqrt(x)</code> | The square root of <code>x</code> . |
| <code>x = Tan(x)</code> | Tangent of <code>x</code> , in radians. |
| Strings: | |
| <code>n = Asc(s)</code> | The ASCII value of the first character of a string. |
| <code>x = CDbI(s)</code> | Convert a string to a number, using local decimal separator. |
| <code>s = Chr(n)</code> | The character whose ASCII value is <code>n</code> . |
| <code>n = CountFields(s1, s2)</code> | The number of fields in <code>s1</code> using <code>s2</code> as a separator. |
| <code>s = Format(x, s)</code> | The number <code>x</code> formatted according to the string <code>s</code> . |
| <code>s = Hex(n)</code> | Hexadecimal equivalent of <code>n</code> . |
| <code>n = InStr(n, s1, s2)</code> | Begin at character <code>n</code> and look for <code>s2</code> within <code>s1</code> . |
| <code>s = Left(s, n)</code> | The left part of string <code>s</code> , <code>n</code> characters in length. |
| <code>n = Len(s)</code> | The length of a string. |
| <code>s = Lowercase(s)</code> | Force a string to lowercase. |
| <code>s = LTrim(s)</code> | Remove leading spaces from a string. |
| <code>s = Mid(s, n1, n2)</code> | A part of string <code>s</code> , <code>n2</code> characters in length, starting at <code>n1</code> . |
| <code>s = NthField(s1, s2, n)</code> | Field number <code>n</code> in string <code>s1</code> using <code>s2</code> as a separator. |
| <code>s = Oct(n)</code> | The octal equivalent of <code>n</code> . |
| <code>s = Replace(s1, s2, s3)</code> | In the string <code>s1</code> , replace the first occurrence of <code>s2</code> with <code>s3</code> . |
| <code>s = ReplaceAll(s1, s2, s3)</code> | In the string <code>s1</code> , replace all occurrences of <code>s2</code> with <code>s3</code> . |
| <code>s = Right(s, n)</code> | The right part of string <code>s</code> , <code>n</code> characters in length. |
| <code>s = RTrim(s)</code> | Remove trailing spaces from a string. |
| <code>s = Str(x)</code> | Convert a number to a string. |
| <code>n = StrComp(s1, s2, n)</code> | Compare strings <code>n = 0 = case-sensitive, 1 = lexicographic.</code> <code>s1 < s2 = returns -1, s1 = s2 returns 0, s1 > s2 returns 1.</code> |
| <code>s = Titlecase(s)</code> | Force a string to Titlecase (first character uppercase). |
| <code>s = Trim(s)</code> | Remove leading and trailing spaces from a string. |
| <code>s = Uppercase(s)</code> | Force a string to all UPPERCASE characters. |
| <code>x = Val(s)</code> | Convert a string to a number using a decimal point as the separator character. |

Tuning:

| | |
|------------------------------------|---|
| <code>CorrectOctave(s, n)</code> | Fixes n to the correct C4 octave, where s is a tuning entry. |
| <code>CorrectOctave(x, n)</code> | Fixes n to the correct C4 octave, where x is a tuning entry. |
| <code>Justify(x)</code> | Places the decimal value of x within octave boundaries $1 \leq x < 2$ |
| <code>JustifyCents(x)</code> | Places the cent value of x within octave boundaries $0 \leq x < 1199.99$ |
| <code>JustifyRatio(n1, n2)</code> | Places a ratio n1/n2 within octave boundaries $1/1 \leq n1/n2 < 2/1$ |
| <code>x = DecimalToCents(x)</code> | Returns a cent value (where an octave = 1200) from a decimal value (where an octave = 2.0). |
| <code>x = CentsToDecimal(x)</code> | Returns a decimal value from a cent value. |
| <code>x = HzEntry(s)</code> | Returns a valid tuning entry decimal from a Hz value input as a string. NOTE: the string can be a CodeSnippet inside <> |
| <code>x = HzEntry(x)</code> | Returns a valid tuning entry decimal from a Hz value input as a decimal. |
| <code>n = HzOctave(x)</code> | Returns the correct C4 octave number from a Hz value input as a decimal. |
| <code>x = entryToX(s)</code> | This is actually the parsing engine itself, so it allows nested tuning entries (where s is a tuning entry). |
| <code>s = scS(sc, n)</code> | Tone n from imported scale sc in the form of a string value. |
| <code>x = scX(sc, n)</code> | Tone n from imported scale sc in the form of a decimal value. |
| <code>n = scLen(sc)</code> | The number of tones in an imported scale sc. |