Using the AC Toolbox

A tutorial

Paul Berg

Version: September 2, 2011/March 23, 2015

Comments: pauberg@actoolbox.net

Software: http://www.actoolbox.net

Contents

Introduction

Computer-aided algorithmic composition is the broad area where computer tools are used to assist in the composition of musical material. The material could be limited to a handful of values useful for one parameter. It could be a framework of time and pitch values to serve as a basis for realizing a composition. It could be data for another program such as a sound synthesis system. The material could be a complete description of a composer's model for a particular composition.

The AC Toolbox is a collection of tools to facilitate various aspects of algorithmic composition. Several models for defining musical events are included. They can be used by defining objects such as sections, shapes, masks, or note structures.

Menu items are available for defining the various types of objects. It is also possible to play, plot, modify, and examine objects in a number of ways. Extensive online help is available.

In addition to Midi input and output, the AC Toolbox can also produce files suitable for use as data with other programs. Score files for Csound and binary OSC files for SuperCollider can be produced. Files for formatting musical notation with FOMUS can be written. Floating-point Midi output via a FireWire interface to a Capybara or Pacarana is also supported.

An important method of creating data in the Toolbox is the use of generators. A number of generators have been included reflecting various approaches to the creation of musical material including tendency masks, stochastic functions, chaotic systems, transition tables, recursive subdivisions, metric indispensabilities, morphological mutations, etc.

The AC Toolbox is implemented in Lisp and input syntax reflects the conventions of this language. It is possible for a user to extend the Toolbox by adding Lisp functions. Additional generators, tools, and transformers can be defined in Lisp to use with the Toolbox.

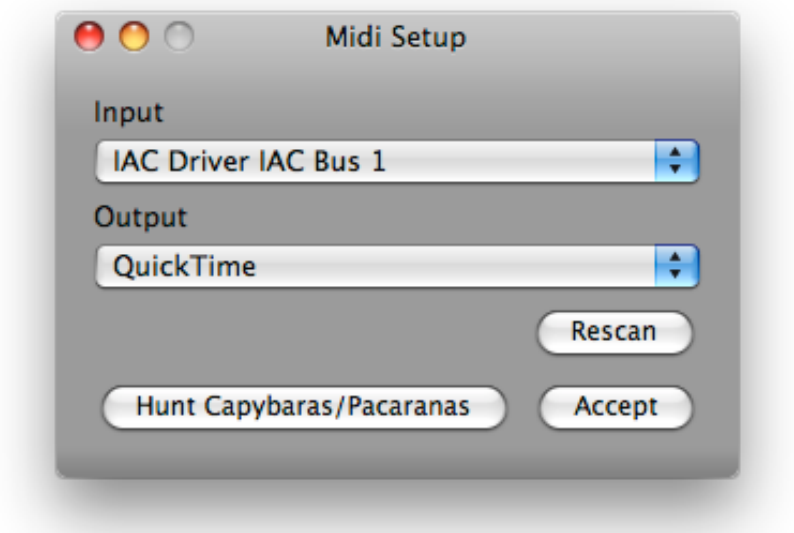
Setup

Computer

The AC Toolbox is an Universal Binary and runs on Apple Macintosh computers with Intel or PPC processors using MacOS 10.3.9 or higher. This tutorial is based on AC Toolbox 4.5.3.

Midi connections

Midi playback within the Toolbox uses either QuickTime Musical Instruments, CoreMidi, or the Symbolic Sound Corporations' Capybara/Pacarana Sound Computation Engine. The Midi setup may be changed and different nodes in the Midi setup may be chosen using the **File** menu item **Midi Setup**. For the changes to take effect, **Accept** should be selected. **Rescan** will look for any changes in the Midi connections and adjust the available nodes in the popup menu accordingly.



Midi playback with QuickTime or a Capybara/Pacarana can use floating-point values for pitch to express microtones. For all other Midi hardware or software, floating-point values will be rounded before being sent.

A Capybara or Pacarana should be connected via a FireWire interface. If *Capybara/Pacarana* does not appear in the list of output destinations, Click on **Hunt Capybaras/Pacaranas** to look for one.

AC Toolbox folder

The folder named *AC Toolbox* should at least contain the following items:
AC Toolbox, Support, Tutorial.

Compatibility

Files produced by the AC Toolbox, using either an Intel or a PPC processor, are interchangeable.

Environment files produced by versions of the AC Toolbox before 4.4 can be read in the current version.

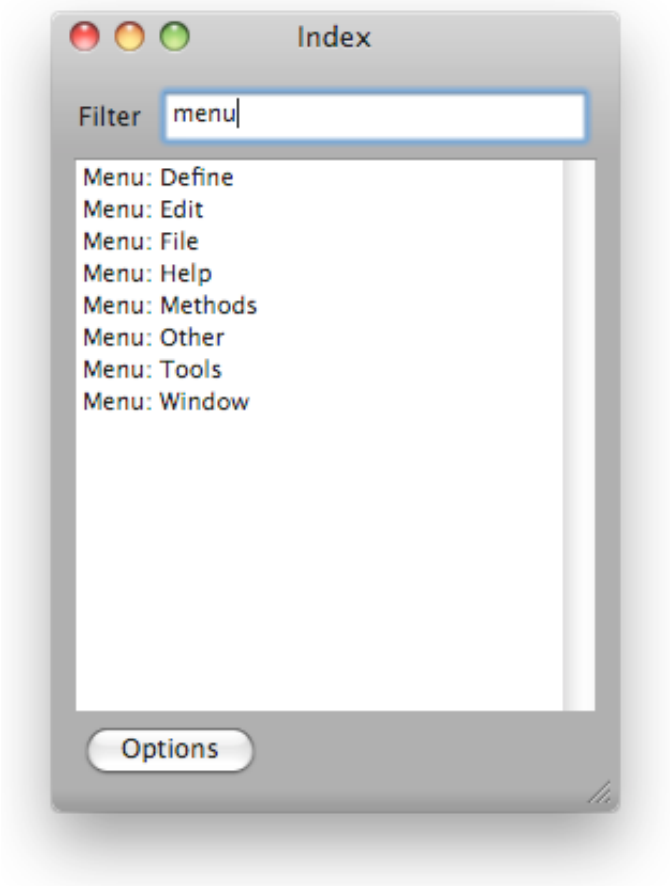
Objects saved in the *new* format (see Preferences) cannot be read in older versions (i.e. before 4.4). To read environment files produced by the current version in older versions of the AC Toolbox, save the objects in the ***old*** format.

Basics

Menu items

The dialog boxes provided with the Toolbox are available through various menus.

Information about menu items can be found by choosing **Index** in the **Help** menu. A table including topics with additional help appears. The table includes all menus.



The **File** menu allows objects to be loaded and saved in various ways. It provides means to import and export Midi files. A connection to music notation software via FOMUS, a music formatting program, is available.

The **Edit** menu provides ways of editing text in dialog box items or in text files. Items for evaluating Lisp expressions are also included.

The **Define** menu allows the various types of Toolbox objects to be defined.

The **Methods** menu allows various methods (functions) to be applied to Toolbox objects. In some cases, methods may also be applied to Csound and OSC files.

The **Tools** menu provides items for inspecting, playing, and removing objects as well as testing generators and other things.

The **Other** menu provides items for using various utilities such as for killing some process, adding comments to object dialogs, and setting various options regarding the AC Toolbox.

The **Window** menu lists all AC Toolbox windows as items. It is also possible to close a selected window or to close all windows (except for the Objects dialog) from this menu.

The **Help** menu offers various ways to get information about parts of the Toolbox.

Input

The dialog boxes must be filled with text.

Numbers may be entered.

Symbols may be used. Symbols may be the names of objects defined within the environment of the AC Toolbox. Symbols may also be one of several predefined symbols for potentially useful quantities such as pitch and dynamics.

The available dynamic symbols are:

```
ppp pp p mp mf f ff fff
```

Dynamics may also be expressed with Midi velocity values (0-127).

Pitches are represented either by Midi note numbers (0 - 127) or symbols containing the pitch class and octave. The octave of middle C is considered to be 4. The symbolic representation for middle C in the Toolbox is *c4*. A half tone higher is *c#4* or *df4*. The lowest available pitch is *c-1* and the highest is *g9*. Note that sharps are *#* and flats are *f* in this representation. In the **Preferences**, the symbolic representation for pitch can be changed to use *c3* for middle C. In that case, octave numbers range from –2 to 8.

Microtones can be specified for pitch values by using floating-point Midi numbers. 60.5 is a quarter-tone higher than Midi note number 60. Playback of microtones is only supported using QuickTime or a Capybara/Pacarana. Playback via other devices will cause these microtones to be rounded to the nearest pitch.

Lists may be used as input. A list is represented as a quote, a left-parenthesis, zero or more elements separated by a space, and a right parenthesis:

```
'(1 2 3)
```

is a list with three elements.

A *function* that produces an appropriate result can also be used as input. In Lisp, a function is a list without a quote. The first element in the list is the name of the function. If there are any other elements in the list, they represent data to which the function is applied.

```
(+ 2 2)
```

+ is the name of the function, *2* and *2* are the data.

```
(random-value 1 100)
```

random-value is the name of the function. *1* and *100* are the data.

Generators

A generator is a function that returns the next value in some series each time it is used. A generator for producing increasing integers would return the value 1 the first time it is used, then 2, then 3, ... A generator for producing Midi note numbers for a major scale starting with 60 (c4) would return 60, then 62, 64, 65, 67, etc.

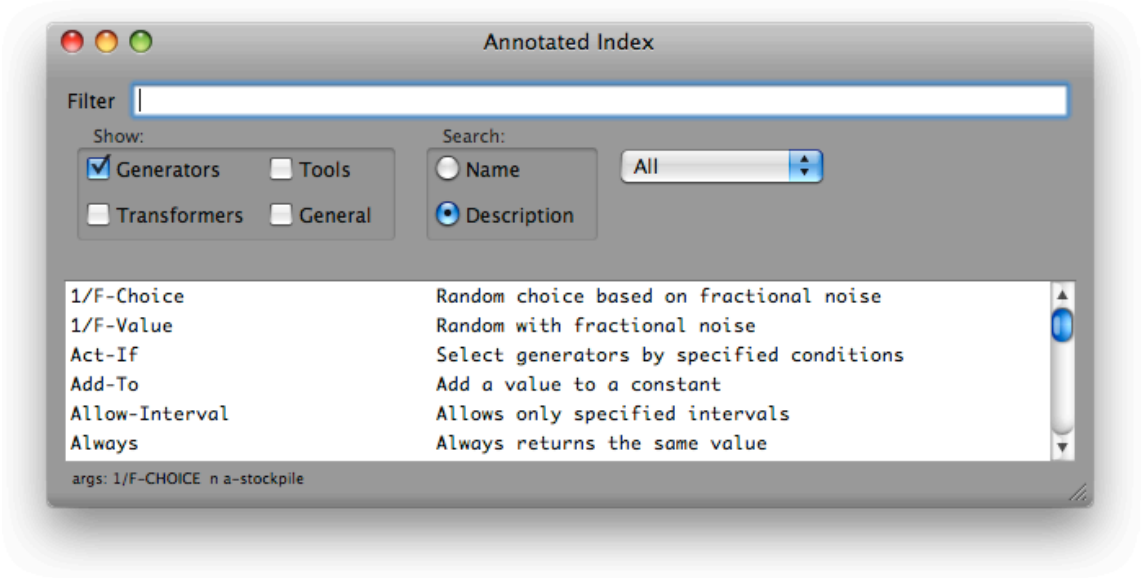
The generators included in the Toolbox have been grouped into various categories such as: Basic, Chaos, Koenig, Noise, Stream, Masks&Shapes, and Verbs. In total, 27 different categories are available.

Chaos refers to nonlinear generators. *Koenig* generators were inspired by (or derived from) the selection principles G.M. Koenig used in his composition programs *PR2* and *SSP*. *Noise* refers to a variety of random or statistical generators. *Stream* refers to generators that are particularly useful when streams (objects that produce values in real-time) are defined. *Masks&Shapes* concerns contours of one or two lines and how they can be mapped to be useful. *Verbs* includes generators such as walk, jump, stutter, and shake.

Generators tend to have one of two suffixes. *Value* means values will be produced between two limits (a bandwidth), e.g. *(random-value 1 100)* will produce a value in the range

between 1 and 100. *Choice* means values will be chosen from an available supply of values, e.g. from a list.
(random-choice '(a b c d e)) chooses a value from the list *'(a b c d e)*.

Individual help windows for each generator are available via the **Annotated Index** menu item in the **Help** menu. Either a table of all generators can be requested or a table of generators from one of the available categories can be selected.



In most cases, a generator will return an integer or real value as appropriate. Usually, if the input to the generator is only integer values, an integer output is produced. Otherwise a real number is returned.

(A generator is often created by a function that will return a specific generator function as its result. Many algorithms for organizing various types of data have been realized in the AC Toolbox as functions that produce generators. In this documentation no further distinction will be made between a function producing a generator or the generator itself. The common use in this documentation is to also call the function that produces the generator a generator.)

Tools

Tools are functions of various kinds to do things that may be useful. Some are essential. Others are only needed in certain special circumstances. Some of the tools can be used as input for the dialog boxes when making objects. Others are more appropriate for users writing Lisp code.

A table listing all available tools and giving access to help windows for them can be found via the **Annotated Index** menu item in the **Help** menu.

When a tool is evaluated, it produces a result such as a list or a number or it performs some action such as writing a file. When a generator is evaluated, it produces a function that still has to be applied before there is a useable result.

Expressed in a different way, you could say a tool is used to do some specific task and a generator is used to generate a series of values.

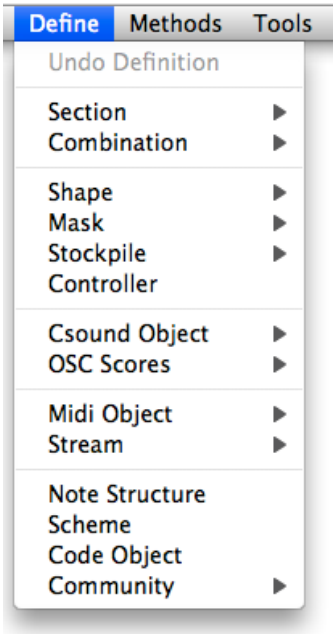
Transformers

Transformers are functions that are used to modify objects or part of an object. Transformers could be used to transpose pitches, stretch tempi, add random deviation to a parameter, etc.

A table listing all available transformers and giving access to help windows for them can be found via the **Annotated Index** menu item in the **Help** menu.

Objects

Various types of objects can be defined in the AC Toolbox. They include sections, shapes, masks, stockpiles, controllers, Csound objects, OSC score objects, Midi objects, streams, note structures, schemes, code objects, and communities. These objects are used to define musical material. To define an object, choose the corresponding menu item in the **Define** menu.



Section
A section can contain one or more notes. In this way it can fulfill many different roles: note, phrase, section, voice, movement, composition, etc.

Different ways of defining sections have been included.
In a *data section*, input for each of the parameters (rhythm, pitch, velocity, and channel) is separate.
In a *note section*, notes are expressed as a combination of those four parameters.
In a *structured section*, an object called a note structure must be entered.
In a *density section*, time is the paramount parameter. Time is filled with either a function or a shape.

Shape
A shape is a curve, reflecting some motion over time. It can be converted to produce a number of values within certain limits.

Mask
A mask is represented by two lines. It is a field that changes over time. A mask can be converted to be in a certain range. Values can be chosen between those boundaries.

Stockpile
A stockpile is a collection of values. A stockpile can be used as the input parameter for generators ending with -choice, e.g. *random-choice*.

Controller
A controller is used to control parameter values on a larger scale. A controller exists outside the boundaries of a section or any other object. It is not to be confused with a Midi controller.

Csound object
A Csound object is a specification of data that can be used to generate a Csound file. It is not the file itself, but the data that could create the file.

OSC score
An OSC score object contains the specification to generate a binary OSC file that can be rendered to an audio file using the SuperCollider Server.

Midi object
A Midi object contains Midi controller values or program changes.

Stream
A stream is an object that produces values in real-time. You turn on the stream and notes or Midi values are produced until the stream is turned off.

Note structure
A note structure is a note, a rest, a delay, or some combination of these structures. Note structures can be combined in sequence or in parallel. Note structures allow the specification of a known group of pitches, chords, and rests.

Scheme
A scheme of variations contains the names of objects that are to be made in a certain order. When the scheme is applied, those objects are made in that order. It is a script to remake several objects.

Code object
A code object contains Lisp code. It is intended for programmers who want to save their code together with the environment. It must be evaluated before it can be used.

Community
A community is a group of sections joined in name only. A community can provide a mechanism for manipulating sections according to some grouping. It is also possible to generate a community containing variants of some section.

Online help

Help can be found in various ways, usually with the **Help** menu.

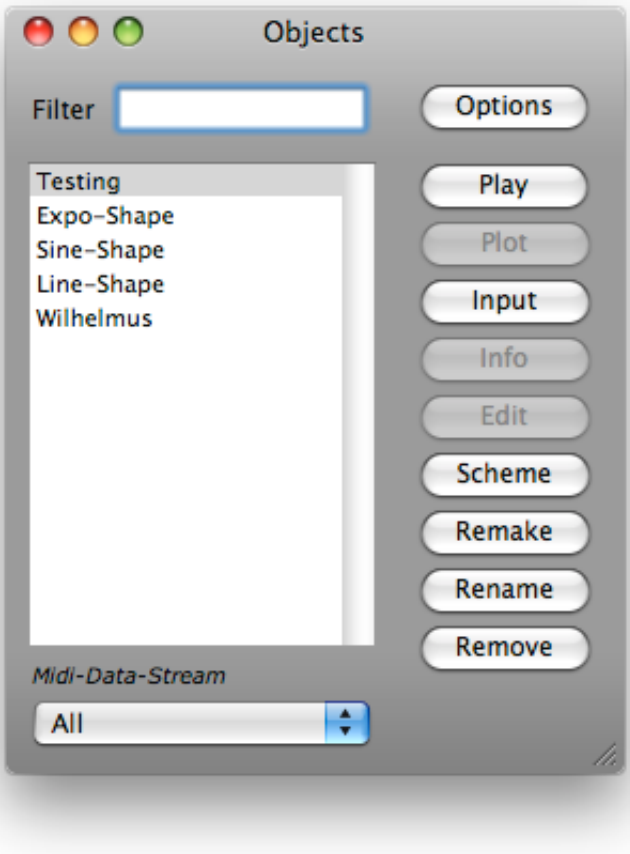
Help Tags
When the mouse is held over a text input item, a button, etc., a help tag will appear after a certain amount of time. Help tags generally do not appear for text items.

?
Many dialogs have a button with a question mark. Clicking this button will produce a window with the relevant help for that dialog. All dialogs that define objects have this help button. A few others such as the Csound File Options dialog also have it.

Generators, tools, transformers
Information about many generators, tools, and transformers is available in the Annotated Index. A brief description appears in the window. Either the names or the descriptions of the generators, etc. can be searched. Selecting a generator etc. will open a more extensive help window.

Alternatively, the name of a generator, etc. can be selected in another dialog. Help can then be found by typing the key combination Command-=. (That is holding the command key while pressing the = key).

Objects
Information about objects that have been defined by a user in the Toolbox can be found using the **Tools>Show Objects** menu item. A table containing all defined objects will appear. In the Options for this dialog, the order for listing the objects can be specified (alphabetical, most recent on top, most recent on the bottom). The input specification for an object can be requested. Various other actions can be performed. Objects can also be removed from the environment by choosing the Remove option.

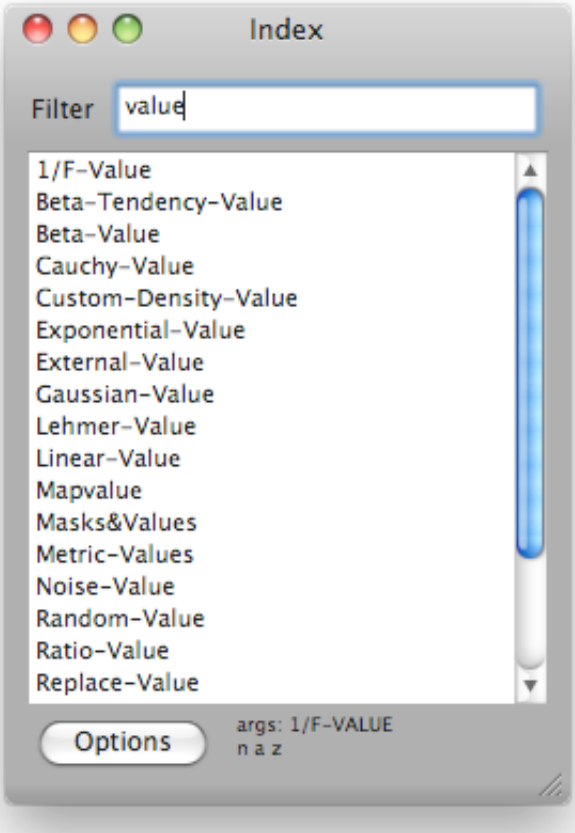


Using a popup menu, a table of objects of a particular type can be selected. Additional buttons may appear that are relevant for that object type.

The table of objects can be filtered to only include objects containing some group of letters. If the previously selected object is not part of this new list of objects, the first object in the table will become the selected object.

Dialog examples
Examples of dialogs used to define various objects can be chosen via the **Help>Dialog Examples** menu item. Each dialog will include an example of appropriate input for that dialog and a brief explanation.

Index
All online help entries for generators, transformers, tools, and general information can be found in the index table. This table can be filtered to only include entries containing some group of letters or words. Index provides a quick way to search for things based on their name. **Index** is found in the **Help** menu.



An option allows limiting the entries in the Index to just the general information about the Toolbox or to everything except the general information. The general information includes topics such as *Writing FOMUS Files*, *Writing Sound Files*, and *Selecting Objects with Generators*. Information about all of the menu items can also be found among the general information. General information is basically information that would otherwise 'fall between the cracks'.

The arguments for the top-most item in the list (or for the most recently selected item) appear in the lower right-hand corner of the Index dialog.

Annotated Index
Online help entries for generators, transformers, and tools can be found in the annotated index table together with a brief description of what the generator, etc. does. Check boxes

allow choosing among the possible types of items that can be displayed. A filter allows searching either the item names or the item descriptions.

A popup menu allows limiting the displayed items to those belonging to a category such as 'Chaos' or 'Rhythm'. A category 'Shortcuts' also exists. It lists all available abbreviations for function names. For example, *rv* is a shortcut for the generator *random-value*. It can be used anywhere that *random-value* could be used.

Arguments for the top-most item in the list (or the most recently selected item) appear at the bottom of the dialog.

Annotated Index can also include the general information available in the Index dialog. This information can be filtered. It does not however include a brief description (since the general information itself is often a brief description). The general information does not get sorted according to categories such as Pitch.

Annotated Index provides a more varied search mechanism than Index. **Annotated Index** is an item in the **Help** menu.

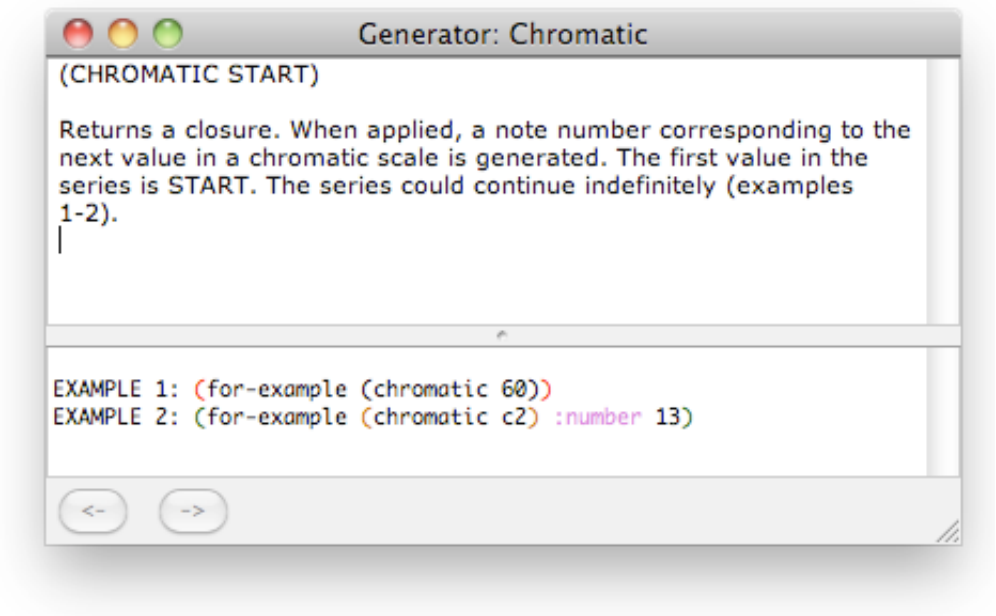
Lisp
This menu item in the **Help** menu contains a submenu with different topics for people who want to know how to use Toolbox objects when writing Lisp code or who want to write their own generators, transformers, or filters in Lisp.

Help->File
A tool with the name 'help->file' can be used to write all or some online help files to a file. This file could be printed, searched, and consulted without using the AC Toolbox. The tool allows an optional type parameter to limit the information being gathered in the window. See the help for this tool for details. To use this tool, evaluate the following expression in the Listener (available via **Other > Listener**):

```
(help->file)
```

About help windows

Help windows for generators, tools, and transformers have a common format and assume that the user is familiar with a few assumptions. Generally, the window is divided into a text pane and a pane with examples that can be evaluated. The following is the help window for the generator *chromatic*:



Text pane
The text pane for a help window for a generator always starts with a template for using that generator. This template includes the name and parameters of the generator, e.g. *(chromatic start)*.

The name of the generator is *chromatic*. The parameter is *start.* The role of a parameter is always explained in the help window.

The help window mentions a closure. It is a type of function that is later applied to get a value. For most purposes, it is not necessary for a user to understand the difference between a closure and a 'regular' function

A generator such as *walk* contains the word *&optional* in its template. This means that all parameters after this word are optional, i.e. they do not have to be used. The user never uses the word *&optional* in an expression.

The template for *walk* is:

```
(walk starting-point next-step &optional lower upper)
```

Valid expressions containing *walk* include:

```
(walk 60 1)
(walk 60 1 40)
(walk 60 1 40 80)
```

A generator such as *random-value* includes the word *&key* in its template. The parameters that follow are keywords and may be ignored. If they are to be used, the keyword is preceded by a colon. After the keyword, the user can enter a value for that keyword. The word &key is never used in an expression. The template for *random-value* is:

```
(random-value low high &key round)
```

Many generators use *round* as a keyword. It will round the result using a quantization unit. The help files for generators with keywords show examples of how to use the keywords in an expression.

Valid expressions using random-value include:

```
(random-value 60 72)
(random-value c4 c5)
(random-value 60.0 72 :round 0.5)
```

The help windows for transformers and tools begin with similar templates.

Example pane
The Help windows for generators, transformers, and tools usually include examples containing Lisp expressions that can be evaluated to produce some sample results displayed in a window. To produce these results, place the cursor at the end of the expression and press the ENTER key (that is different than the return key). The examples are described in the text pane.

The examples can also be evaluated by selecting the Lisp part of one of them and using the menu item **Edit>Evaluate Selection**.

Examples often use tools such as *for-example*, *print-result*, *plot*, *show-transformation*, *fill-template*, *open-url* and *show-help*.

For-example
The function *for-example* applies a generator several times and prints the result in a window. Usually *for-example* will produce 20 values. An expression such as

```
EXAMPLE 1: (for-example (random-value 1 100))
```

means that *for-example* will use the generator *random-value* to produce some sample results.

For-example also has keywords available to adapt the output. Some expressions make use of these keywords. The expression

```
EXAMPLE 1: (for-example (produce-pulse '(5 0 2 4 1 3) 2)
                        :number 12 :number-per-line 6)
```

uses the generator *produce-pulse* to produce 12 values that will be printed six values to a line.

If the input to *for-example* is not a generator, the result of evaluating the input will be printed once to a window.

There is an option to open a new window for each result. This could be useful if various outputs should be compared.

Print-result

This tool evaluates an expression and prints the result in a window. The result is one value and may be a number, list, etc. depending on what is appropriate.

```
EXAMPLE 1: (print-result (average '(1 2 3 4)))
```

Plot

Plot produces a graph related to its input expression. If the input is a generator, that generator is applied a number of times and the values are plotted. When the following expression is evaluated,

```
EXAMPLE 1: (plot (random-value 1 100))
```

100 values produced by the generator *random-value* are plotted.

If the input is a list, the values in the list are displayed.

Plot has keywords available to adapt its output. *Number* allows a generator to be applied a specified number of times instead of the default 100 times. *Min* and *max* allow a minimum and maximum value to be specified for plotting the results. If these values are not specified, plot will use the actual minimum and maximum value derived from the results.

In the following expression

```
EXAMPLE 1: (plot (random-value 50 60) :number 200 :min 1 :max 100)
```

the generator *random-value* is applied 200 times. Even though this generator produces values in the range 50-60, the plot will use a minimum value of 1 and a maximum value of 100.

Show-Transformation

Help windows for transformers use the function *show-transformation* in the expressions that produce sample results. An example is:

```
EXAMPLE 1: (show-transformation (transpose 12)
                                '(60 62 64 65))
```

Show-Transformation prints the result of applying the transformer, in this case *(transpose 12)*, to each value in the list.

```
Applying transformer
(transpose 12)
gives the following results:

Value:   60    Transformation:   72
Value:   62    Transformation:   74
Value:   64    Transformation:   76
Value:   65    Transformation:   77
```

Fill-Template

FILL-TEMPLATE creates a dialog box used for defining an object. The expression may seem confusing but you do not need to understand the expression. What is important is that when you evaluate the expression, a familiar looking dialog should appear.

```
EXAMPLE 1: (fill-template specify-stockpile stock1 c3 c#3 64 66 69 b3)
```

Open-Url
OPEN-URL will open an url supplied as a string if there is an internet connection.

```
EXAMPLE 1: (open-url "http://www.actoolbox.net")
```

The default browser is used. A few help items include an url for getting more information.

Show-Help
Some help items refer to other generators or tools. One way to open the other generator or tool is by evaluating an expression that contains *show-help*. The argument is the symbol for the item to be opened.

```
EXAMPLE 1: (show-help 'random-value)
```

Errors and emergencies

When the AC Toolbox catches errors, a message dialog containing the error message is placed on the screen. These errors include things such as not filling all the boxes of a dialog, using undefined objects, etc. The message should help you fix the problem.

Errors missed by the Toolbox but noticed by the underlying Lisp environment are also printed in a dialog. These messages tend to be about obscure things such as divide by zero. The message may or may not help you fix the problem.

In very exceptional cases, an underlying Lisp error could cause a message to be printed in a window of the Terminal application.

In that case, type :a in the Terminal window.

If that doesn't help, type :top.

If that does not help, try :top or :a again.

If the Terminal window presents options, type :c 1 if you want to do option 1, :c 2 if you want to do option 2, etc.

If a message appears in the Terminal concerning stack overflow, choose :c 2 to extend the stack. This may need to be done a couple of times.

If an object with many events is to be plotted and the pizza wheel of death spins for a long time and a Terminal window opens, try typing :c to force a list of options, then choose :c 2 to extend the stack.

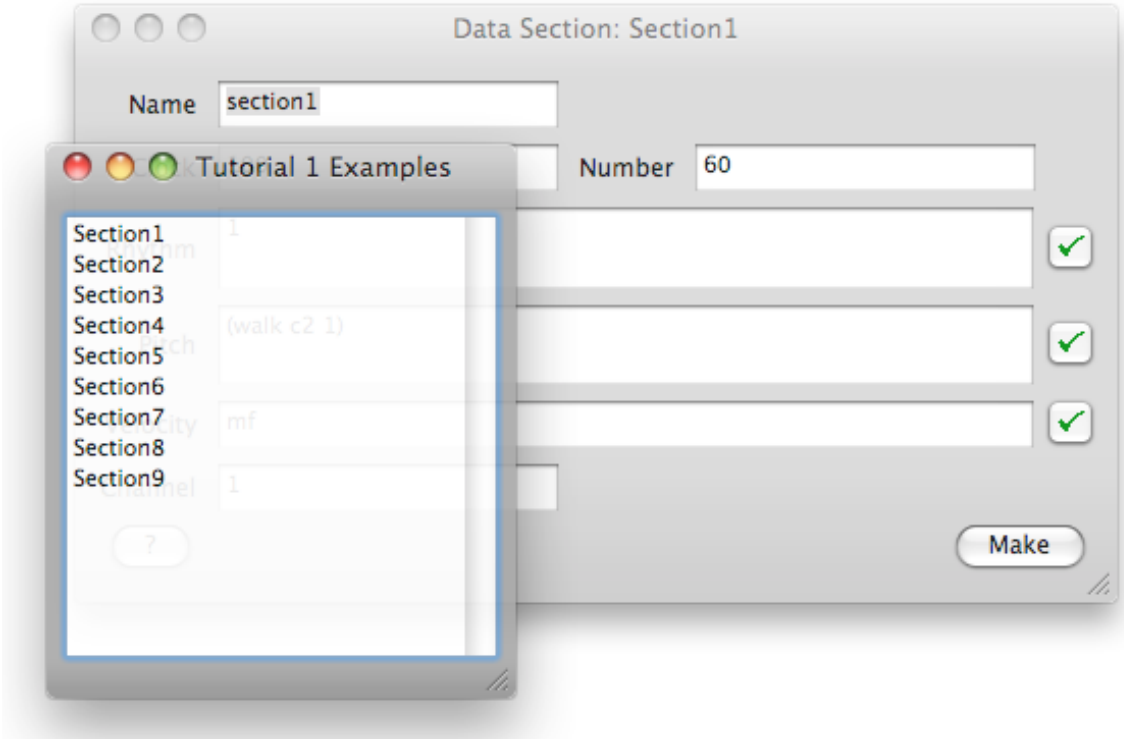
There have been reports that the printing in the Terminal window may be followed by a message from the system that the AC Toolbox crashed. Close the Terminal Window, click on Reopen in the crash message, and you may be able to continue working where you left off in the AC Toolbox.

Emergencies
If for some reason, the AC Toolbox spins out of control (e.g. some process never stops), choose **Kill and Reset** from the **Other** menu or type its keyboard equivalent *Command-K*. This keyboard shortcut can also be used as a brute force method to stop Midi playback of any object or to interrupt Csound or OSC rendering. It will also restore the cursor if it gets lost for some reason. There are cases however where this menu item is not able to save the day. In that case, force quit is the only option.

About this documentation

This documentation includes a number of *tutorials* explaining various aspects of the mechanics of the AC Toolbox. Sections, shapes, masks, stockpiles, etc. are explained. This documentation is not a general tutorial on algorithmic composition.

Accompanying each of the tutorials is an example file containing the objects discussed in the text. These files can be found in the folder *Tutorial/Tutorial Examples.* The file should be loaded by choosing **Load Examples** in the **File** menu. After the objects in a file have been loaded, a table appears containing the names of the objects in the example file. When an object is selected from this table, the dialog box with the object definition appears. Click on *Make* to make each object. The objects should be made in the order in which they occur in the table, from top to bottom.



This tutorial often includes requests for the reader to perform certain actions within the AC Toolbox. A request such as "make section1" assumes that the reader has loaded the example file for that particular tutorial. The reader should then make the example object named *section1*. These requests are preceded by a • symbol, e.g.

- Click on the *Make* button to make the section.

Tutorial 1

Making data sections, combining sections

Data section
A *section* is a collection of one or more notes. Section is the neutral term used in the AC Toolbox for a note, a motive, a phrase, a lick, a section, a part, and a composition. Almost any group of time and pitch points is called a section. This avoids having to make *a priori* value judgments about the material being produced.

A data section is a section where the values for each parameter are specified separately. Rhythm, pitch, dynamics (velocity), etc. are calculated without necessarily referring to each other.

A *clock unit* is specified. This is the basic beat of a section. It is expressed in milliseconds. It can also be calculated with the tool *bpm*. *(bpm 160)* translates 160 beats per minute to the corresponding value in milliseconds. *Mm* will convert a metronome indication to a value in milliseconds, e.g. *(mm 120)*. *Rhythm* is expressed as a multiple of this clock unit. One times the beat, 2 times the beat, 3.5 times the beat, etc.

Pitch is expressed as Midi note numbers (0-127) or as symbols such as *c4* or *c#4*. *C4* is middle c (Midi note number 60). *Ef4* is e-flat in the same octave as middle c. The octave number for middle c can be changed in the Preferences. Pitch can also be expressed as floating-point Midi note numbers. 60.5 is a quarter tone higher than 60. Playback of floating-point pitch values is only possible using QuickTime or a Capybara/Pacarana.

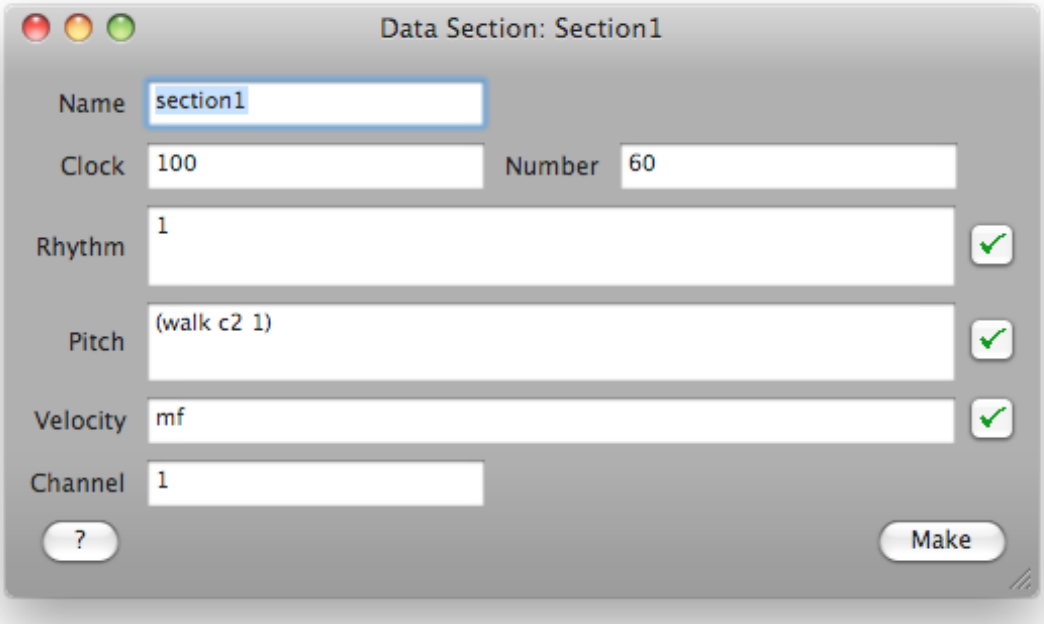
When the menu item **Define>Section>Data section** is chosen, a dialog box appears. Several boxes for text input must be filled with values for the various parameters. The values entered must be either

a number such as *1* or *1.23*
a list such as *'(1 2 3)*
a symbol such as *mf* or *c4*
a generator such as *(random-value 1 10)*
a tool (function) such as *(extract 'pitch section1)*.

Note that lists containing data always have a quote to indicate that they should be used *as is*. If the quote is absent, the first element in the list would be considered a function and an attempt would be made to call a function by that name. This usually results in an error.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 1 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box or choose **Make (Default Button)** from the **Tools** menu.

Defining a section
• Load the file *Tutorial 1 Examples* and select *section1* in the table. You should see the following dialog box:

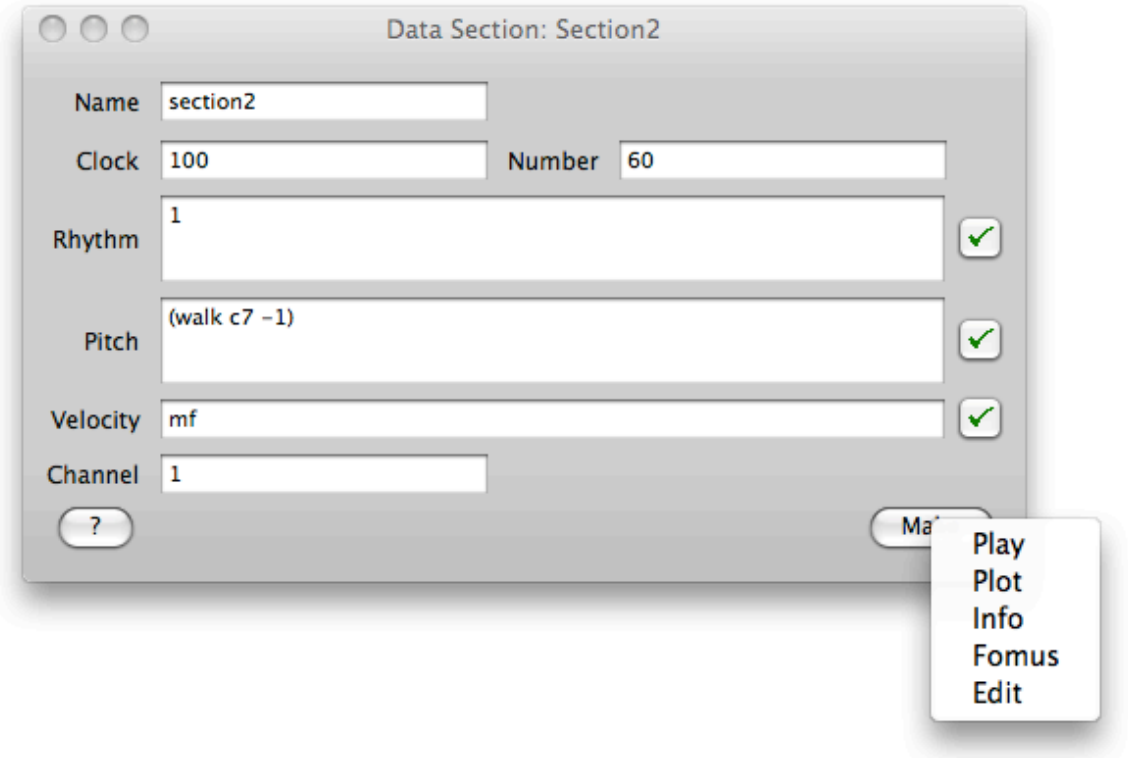


The names of AC Toolbox objects must be contiguous, i.e. without any spaces.
The clock unit is 100 milliseconds (1/10 of a second) .
Sixty notes will be made.
The rhythm unit is 1 that means each note lasts 1 times 100 milliseconds.
Pitch is determined by a generator. Walk starts at *c2*. Each successive note adds one to the pitch value.
Instead of c2, a Midi note number such as 36 could have been specified.
Velocity is *mf*. A number between 0-127 could have been used instead.
Midi channel 1 is used.

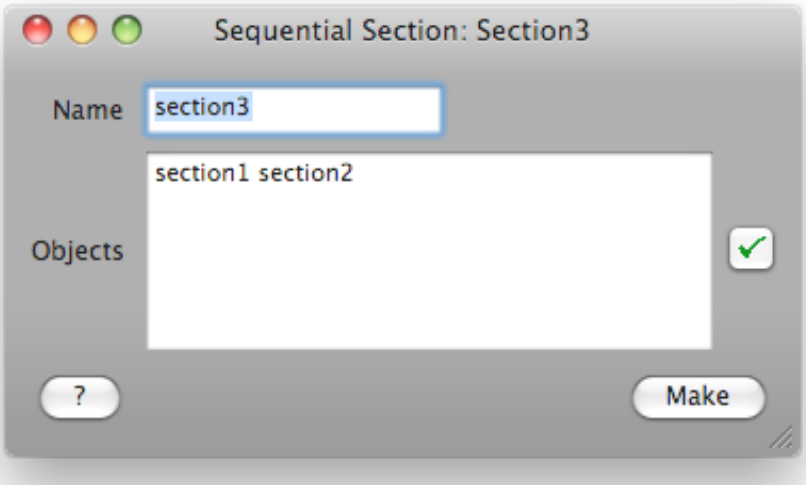
• Click on the *Make* button to make the section. Alternatively, use the keyboard shortcut Cmd-B (from the menu item **Tools>Make Object (Or Do Action)**).
• Choose the *Play* button in the Objects dialog to play the section. Alternatively, use the keyboard shortcut Cmd-P (from the **Tools>Play** menu item).
• Plot the piano roll representation of this section using the *Plot* button in the Objects dialog. (Piano roll notation represents pitch over time; unlike a piano, the high pitches are on the top of the graph and the low pitches at the bottom).

To plot just the pitches, rhythms, or velocity values, choose menu item **Tools>Plot>Section**. Enter the name of the section to plot in the dialog.
• To make *section2*, change the name of the section to section2. Change the pitch specification to:
(walk c7 -1).

It is also possible to play, plot, and perform a few other activities with a section by clicking on the *Make* button with the right mouse button. The combination CTRL-Click could also be used. This opens a popup menu near the *Make* button to examine data from the section. This option plays, plots, etc. the most recently made object. You should click on *Make* before using this feature.



Combining sections
To combine the two sections in sequence (one after another), choose the menu item **Define >Combination>Sequential Section**.



• Play *section3*.

Object names can be typed in the appropriate box. There is no syntactic limit to the number of sections that can be combined in sequence.

Another way to enter sections in a sequential section dialog, is to drag the name from the Objects dialog.

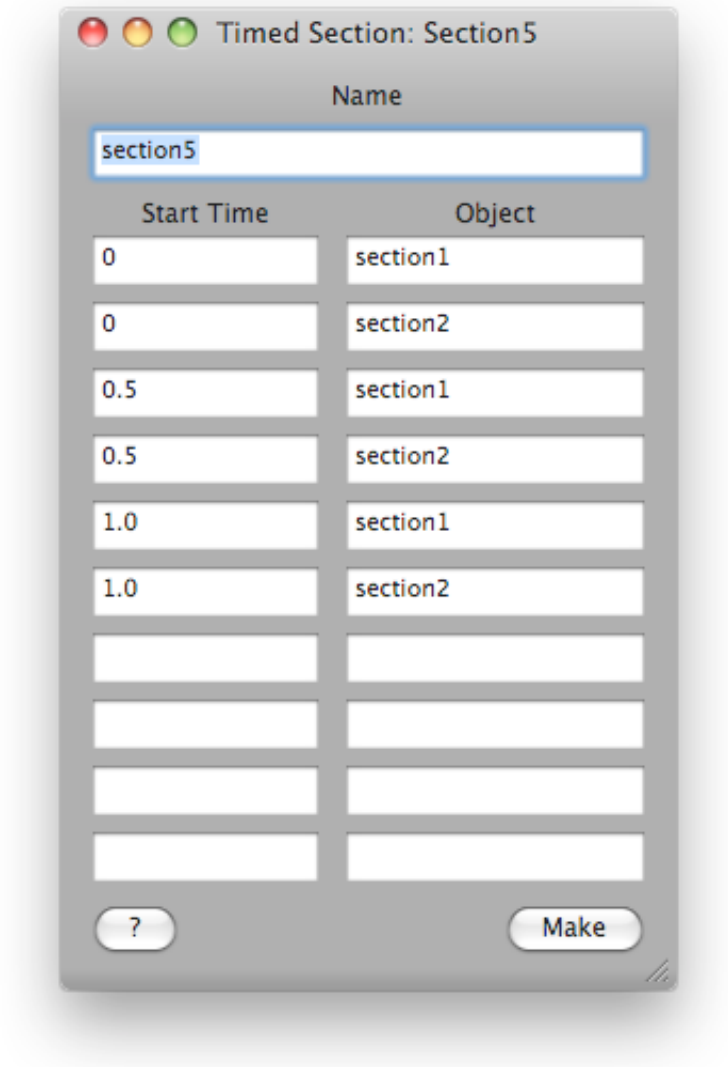
When defining a sequential section, a number can appear between the section names. This number represents a delay in seconds between the sections. For example, *section1 3 section2*.

To combine the two sections in parallel (at the same time), choose the menu item **Define >Combination>Parallel Section**.

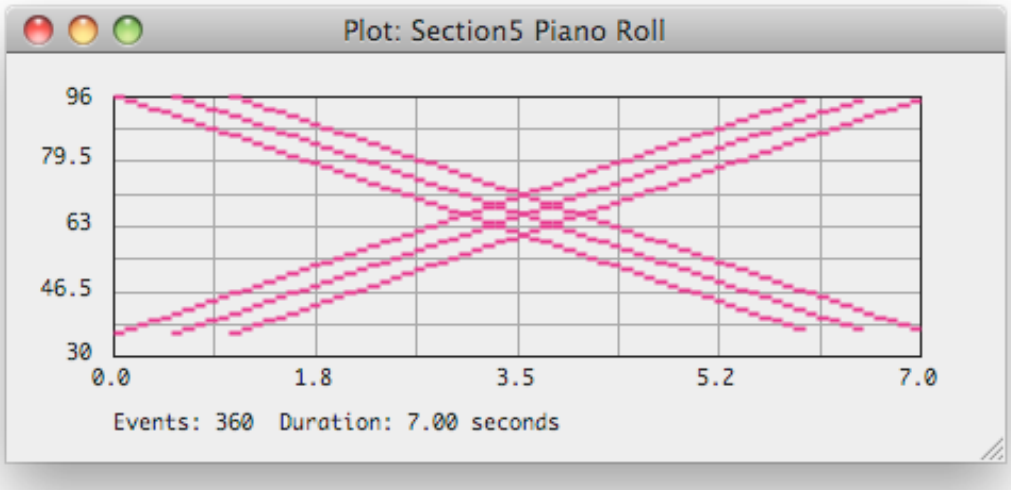
• Plot and play *section4*.

Start times in a parallel section can be staggered by inserting a number representing a time in seconds between the sections. This number is a time from the beginning of the parallel section. For example, *section1 .1 section2 .2 section3* would indicate that section2 starts .1 seconds and section3 begins .2 seconds after the beginning of section1.

To combine sections according to a list of start times, choose the menu item **Define >Combination>Timed Section**. Start times are expressed in seconds. Start time refers to the time elapsed since the beginning of the section. It is not the time between objects.



The piano roll representation of *section5* is:



- Play *section5*.

Generating several parameters
It is possible to use generators for several parameters. In *section6*, rhythm, pitch, and velocity use generators. Check the help for *walk* for an extended discussion of the meaning of each argument to that function. A brief description follows:
Looking at the pitch parameter, walk starts with c4. The next pitch will add a random value in the range of -5 notes to +5 notes. If the new pitch is below the minimum value c2 or above the maximum value c6, the sign of the value being added is reversed. A text description of the input for *section6*:

```
Name            section6
Clock unit      100
Number          200
Rhythm          (walk 1 (random-value -0.5 0.5) 0.1 5)
Pitch           (walk c4 (random-value -5 5) c2 c6)
Velocity        (walk mf (random-value -5 5) pp ff)
Channel         1
```

• Plot and play *section6*.

Random-value produces an uniformly distributed random value between two limits. If the two limits are integers, the result is an integer. If one or more of the limits is a real number, the result is a real number. Random-value can be abbreviated as *rv*.

Section7 uses *random-value* with real and integer values.

```
Name            section7
Clock unit      100
Number          100
Rhythm          (random-value  1.0 2)
Pitch           (random-value  c2 c5)
Velocity        (random-value  30 90)
Channel         1
```

The specification for rhythm includes a real number, therefore rhythm will be random values between 1.0 and 2, e.g. *1.1*or *1.56*. Pitch will be an integer (Midi note) number between c2 (36) and c5 (72), e.g. *38* or *60*.

To see the value of predefined symbols such as *c2* or *mf*, enter them in the Listener (available via **Other>Listener**). `CL-USER 1 >` is a prompt. After it, an expression can be entered. The evaluation of that expression is printed on the next line.

```
CL-USER 1 > c2
36
CL-USER 2 > mf
64
```

Dragging from the Index
If a tool or generator is dragged from the Index or the Annotated Index to an edit box of another dialog, an expression with the generator or tool and its argument names is entered in the edit box. The edit box should be selected before dragging something to it.

If random-value is dragged from the Index to the edit box for pitch in a Data Section dialog, the expression *(random-value low high &key round)* is entered. The parameters *low* and *high* should be replaced with their desired values. *&key round* can be erased.

Channels can also be changed
The channel number for any section does not have to be constant. It can be changed for each note. If the specification for channel is a list, the first note gets the first channel number, the second note the second, and so on. At the end of the list, it cycles back to the beginning. If the specification for list is *'(1 2 3)*, the first note will be assigned to Midi channel 1, the second to channel 2, the third to channel 3, the fourth to channel 1, etc.

```
Channel         '(1 2 3)
```

• Try making a section with a list for the channel assignment. Instruments can be assigned to different channels using menu item **Other>General Midi**.

The channel number for each note can also be generated. If a generator is specified for the channel, the generator is applied for each note to determine the channel for each note.

```
Channel         (random-value 1 16)
```

Each note will be assigned a channel at random, varying between channels 1 to 16.

Making notes until some amount of time has been filled
Instead of specifying the number of notes to be created for a data section, the number specification can also use the expression *(until-time 10)*. 10 represents 10 seconds. Notes will be generated for the data section until the specified amount of time has been filled. The resulting section always allows the last note to sound for its full duration. This could mean that the total length of the section is longer than the specified number of seconds.

Section8 is an example of using *until-time*.

```
Name            section8
Clock unit      100
Number          (until-time  10)
Rhythm          (random-value  1.0 2)
Pitch           (random-value  c3 c5)
Velocity        (random-value  30 90)
Channel         1
```

• Make *section8*. Plot the piano roll representation of it and notice the number of notes that were actually produced.

Until-time can be abbreviated as *ut*.

Using floating-point values for pitch
For most generators, if one or more of the input parameters is a floating-point number, the result will be a floating-point number. Otherwise, the result is an integer. If floating-point values are used for pitch, microtones result. While floating-point pitch values can be specified in all sections, playback of microtones is only possible using QuickTime or a Capybara/Pacarana. In all other cases, pitch values will be rounded on playback.

Midi note number 60 is c4. 61 is one semitone higher: c#4. 60.5 is a quarter-tone higher than 60. Many generators, including *random-value*, have a keyword *round*. This keyword

can be bound to a quantization unit such as 0.5. If *random-value* is used for pitch and either *low* or *high* are floating-point numbers and *round* is 0.5, the output of *random-value* will be quantized to units of quarter-tones.

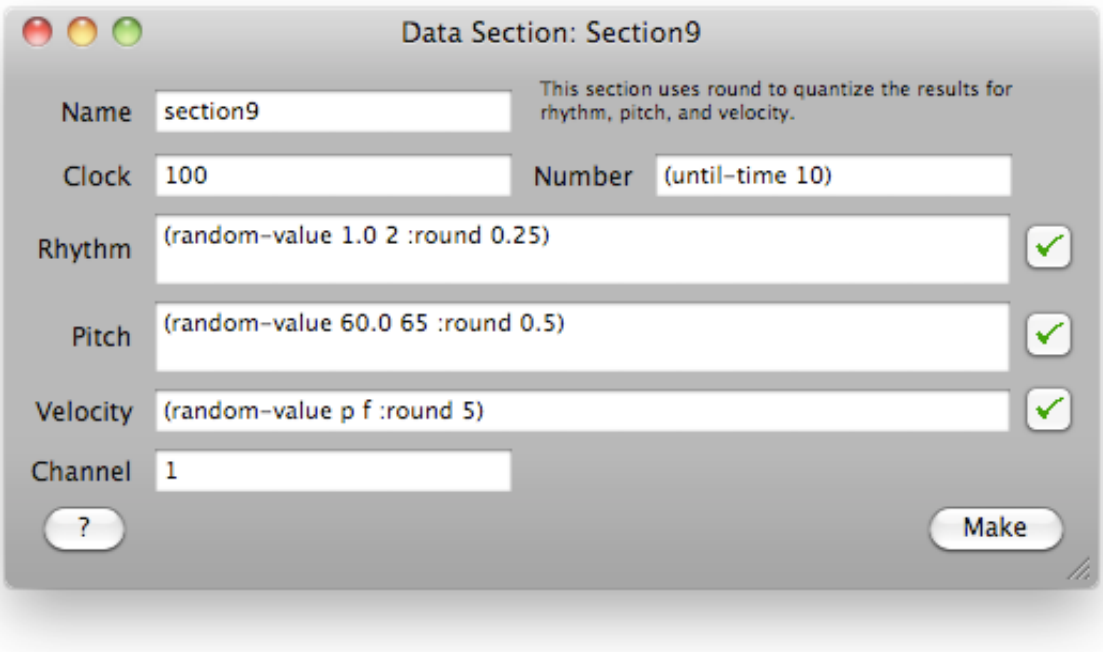
Section9 is an example using fractional pitch values rounded to quarter-tones. Rhythm is rounded to units of 0.25. Velocity is rounded to steps of 5.

```
Name            section9
Clock unit      100
Number          (until-time  10)
Rhythm          (random-value  1.0 2 :round 0.25)
Pitch           (random-value  60.0 65 :round 0.5)
Velocity        (random-value  p f :round 5)
Channel         1
```

More information about using microtones in a section can be found in Tutorial 18.

Adding comments
Comments can be added to sections and other objects. These comments are saved with the object.

To add a comment, select the dialog with the object definition, then choose menu item **Other>Add Comment**. An edit box will be appear to enter the comment. To save the comment, the object needs to be made (again). The dialog for *section9* has had a comment added:



Summary

Menu items
load examples, data section, sequential section, parallel section, timed section, play, make object, plot, add comment

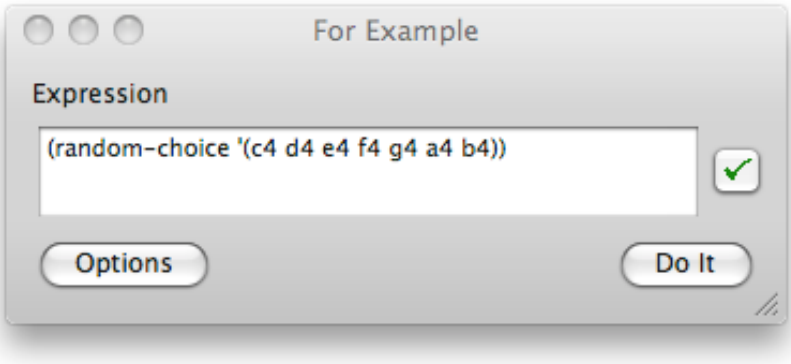
Generators
walk, random-value

Tools
bpm, mm, until-time

Tutorial 2

Using stockpiles, making choices

Generators that choose
Several generators available in the AC Toolbox have the suffix *choice*, such as *random-choice* and *series-choice*. These generators require a collection of values from which they can make their choice. This collection of values could be a list. Note that the list must be preceded by a quote.



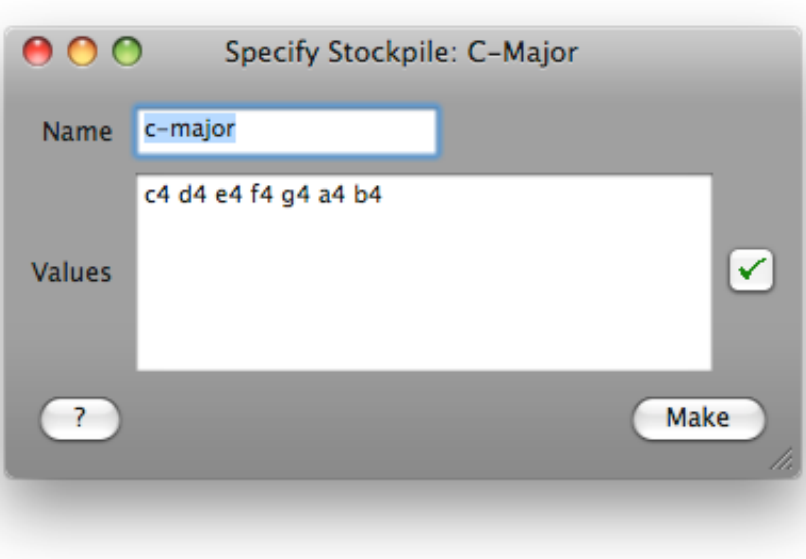

```
20 values from:
(random-choice '(c4 d4 e4 f4 g4 a4 b4))

        a4        a4        g4        c4        a4
        b4        b4        g4        b4        d4
        d4        d4        f4        g4        c4
        d4        e4        f4        f4        g4
```

If frequent choices are made from the same collection of values, it may be more convenient to specify that collection separately and to give it a name. Such a collection of values in the Toolbox is called a *stockpile*.

Stockpile
A stockpile is an AC Toolbox object that contains one or more values. These values can be numbers or symbols. The idea behind a stockpile is that a composer may wish to first gather or prepare material before specifying its use or organization. A stockpile can be specified (each value is entered one by one), generated (using a generator), or constructed (using a function that will produce a list of values). The rule stockpile will be skipped for now. Its primary purpose is in specifying a grammar. An example can be found in the help file for the tool *rewrite*.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 2 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Specifying a stockpile
To specify a stockpile, enter each value (number or symbol) in the available dialog box. Notice that there are no quotes or parentheses. Values are separated by spaces.



Once made, this stockpile can be used as a parameter of a data section

```
Name            section1
Clock unit      150
Number          15
Rhythm          1
Pitch           c-major
Velocity        mf
Channel         1
```

or as the argument to one of the *choice* generators.

```
Name            section2
Clock unit      150
Number          50
Rhythm          (random-choice '(1 2 3))
Pitch           (random-choice c-major)
Velocity        mf
Channel         1
```

Random-choice can be abbreviated as *rc*.

The generator *on-the-fly* can also do calculations with values from a stockpile. Constants or values from lists, stockpiles, or generators can be added, subtracted, etc. from a stockpile.

On-the-fly can transpose a stockpile by adding to or subtracting from each value in the stockpile. In *section3*, the stockpile values are transposed down an octave by subtracting 12. Note that the original stockpile is not changed. The values taken from the stockpile are altered.

```
Name            section3
Clock unit      150
Number          50
Rhythm          (random-choice '(1 2 3))
Pitch           (on-the-fly c-major - 12)
Velocity        mf
Channel         1
```

The value being added can also come from a generator. In *section4*, a value produced by the generator *random-choice* will add –12, 0, or 12. This shifts values up or down an octave.

```
Name            section4
Clock unit      150
Number          50
Rhythm          (random-choice '(1 2 3))
Pitch           (on-the-fly (random-choice c-major)
                            + (random-choice '(-12 0 12))
Velocity        mf
Channel         1
```

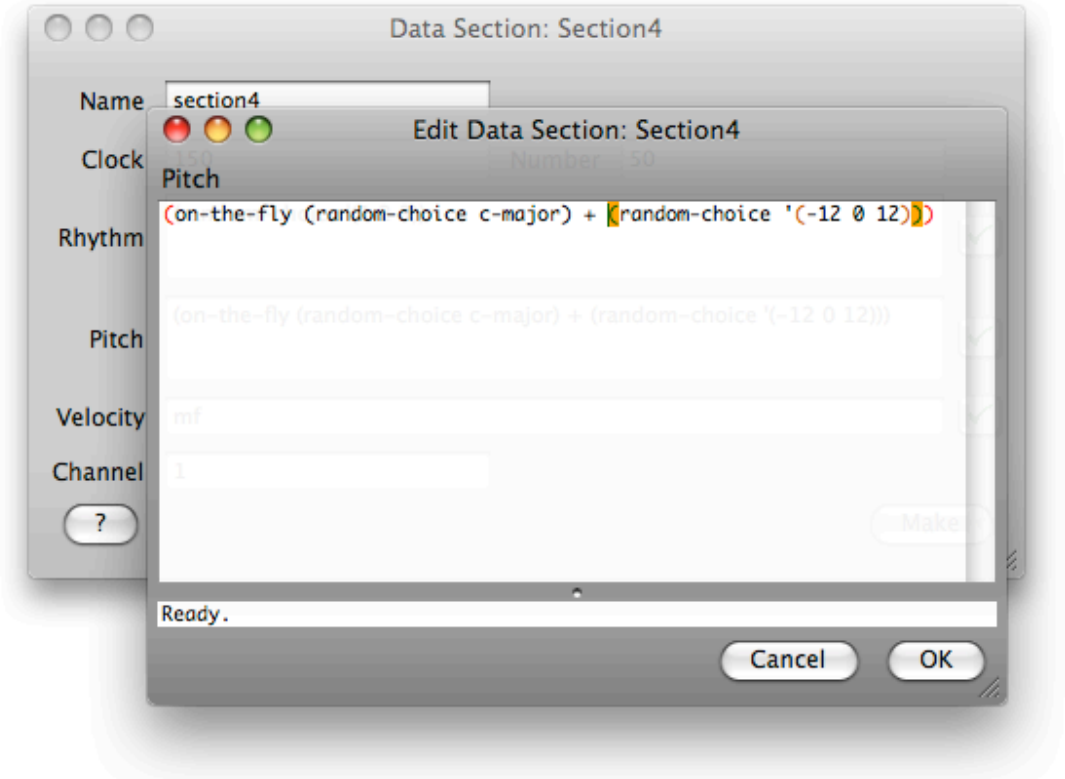
On-the-fly can be abbreviated as *otf*.

When entering a text expression in a dialog box, you can move the cursor to the next line with **Edit>Insert Newline** (command-return).

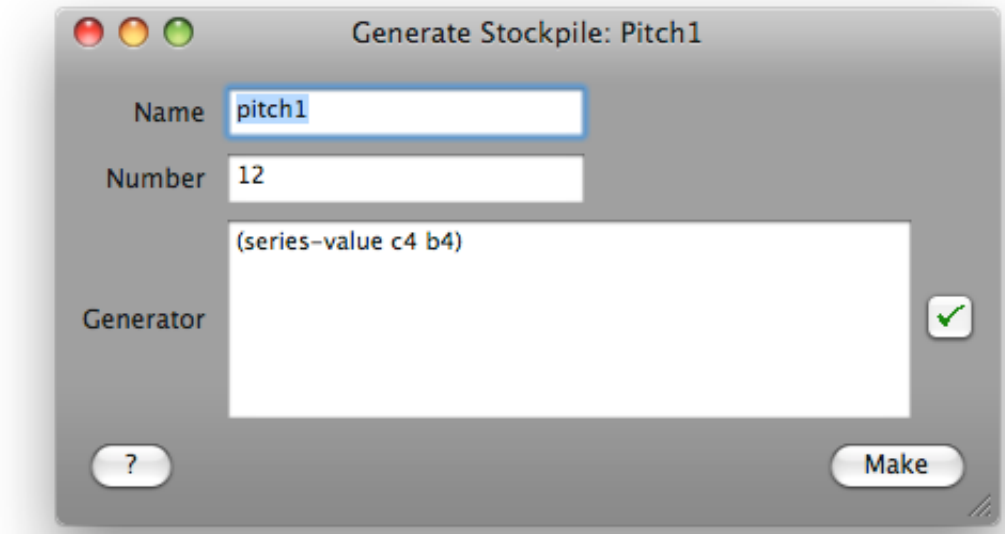
The pitch expression in *section4* may be somewhat confusing if you are not used to all of the parentheses. One way to check the limits of an expression is to place the cursor somewhere in or immediately after an expression. With the menu item **Edit>Select ()**, you can select the expression. The keyboard shortcut for this menu item is CMD-(. When the cursor is placed at the end of the expression, the entire expression should be selected if there are the correct number and kind of parentheses.

Another way to check the parentheses is with **Edit>Balance ()**. If the cursor is after a) or before a (, this command will move the cursor to a corresponding parenthesis, if one is found. Repeating the command will move it back. The keyboard shortcut for this menu item is CMD-0. That is a zero, not an Oh.

An alternative is to click on the check box  that opens a mini-editor where you can check if parentheses are balanced and perhaps use more space to enter the expression. Corresponding parentheses are highlighted in the mini-editor. Click on *OK* to copy the (edited) expression back to the dialog.



Generating a stockpile
A stockpile can also be produced with a generator. The generator is applied a *number* of times to produce the values for the stockpile.



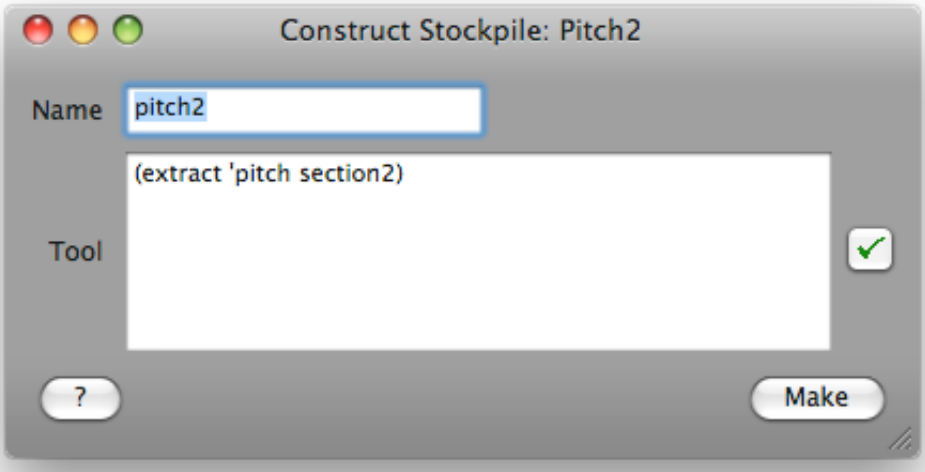
A stockpile with 12 values is produced. *Series-value* will pick note numbers between c4 and b4 randomly without any repetition. Because 12 values are chosen, each pitch will occur once. *Series-value* can be abbreviated as *pick*.

To see the actual values that were generated for stockpile *pitch1*, click *Make* with the right mouse button (or use CTRL-Click) and choose *Values*.

Another way to see the values is to type the stockpile name in a *For Example* dialog (**Tools>For Example**).

• Generate stockpiles using other generators. Examine the values produced. Make data sections using these stockpiles.

Constructing a stockpile
To *construct* a stockpile, you must use a function that will produce an entire list of values for the stockpile. To construct a stockpile, you normally will use a tool. An example of such a tool is *extract* that will return a list of values extracted from an object such as a section. The possible parameters that can be extracted include *pitch, rhythm,* and *velocity. Pitch2* constructs a stockpile containing the pitches used in some previously defined section:



Other tools that could be useful for constructing stockpiles include *convert*, *derive-transition-table*, *from, gather-until*, *generate-range*, *make-conditional-table*, *make-lookup-table*, *make-unconditional-table*, *multiple-bandwidths*, and *pulse-interpolation.*

The expression

```
(multiple-bandwidths 1 10 20 25)
```

returns a list with all integers between successive pairs of low and high values:

```
(1 2 3 4 5 6 7 8 9 10 20 21 22 23 24 25).
```

All integers between (and including) 1 and 10 and between (and including) 20 and 25 will be in the list. Any number of low/high pairs can be specified.

(generate-range (major 36) 72) returns a list of values created by applying the generator until a certain limit has been reached. The above expression returns:

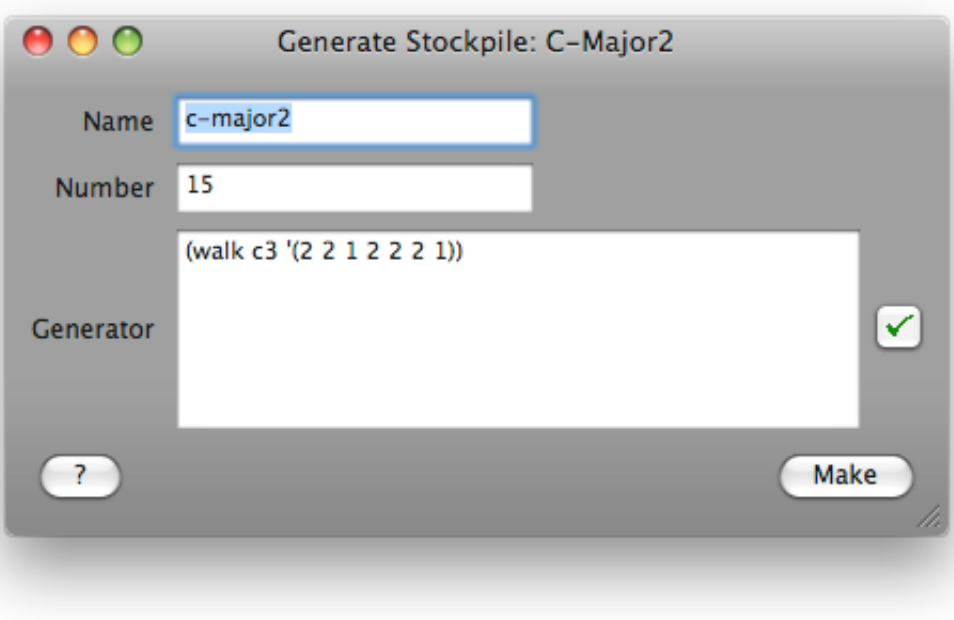
```
(36 38 40 41 43 45 47 48 50 52 53 55 57 59 60 62 64 65 67 69 71 72).
```

The generator should return an ascending series of values.

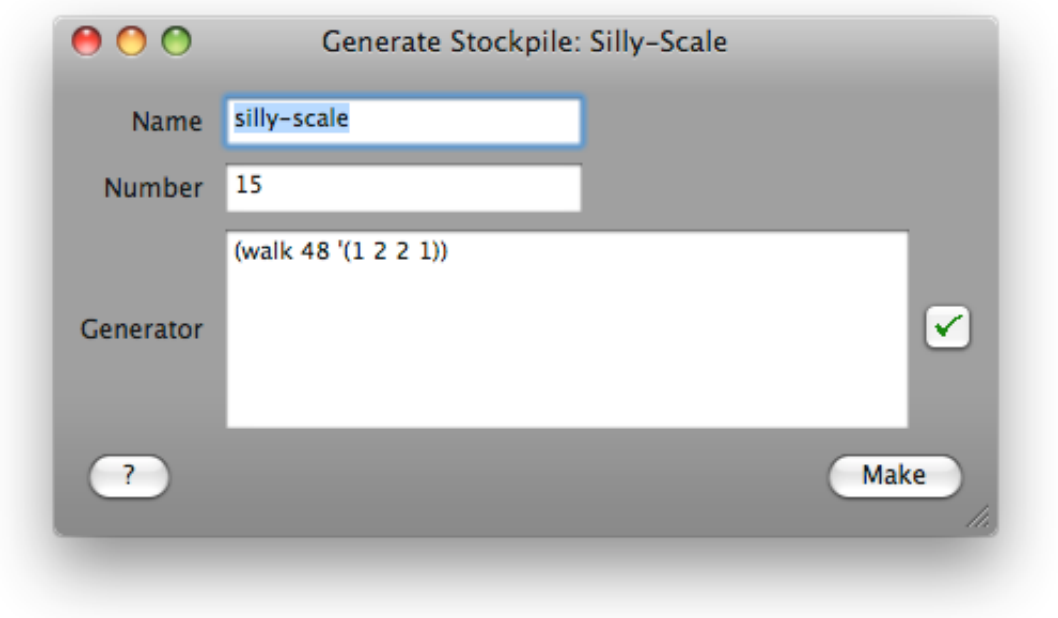
The **Annotated Index** contains a category entitled **Stockpiles** that lists generators and tools that are particularly useful with stockpiles.

Defining scales in a stockpile

In addition to the scale generators that are included in the Toolbox (*major*, *harmonic-minor*, *melodic-minor*, *pentatonic*, *whole-tone*), a stockpile can be defined to contain values representing other scales. This can be done by specifying the interval succession of the scale as an argument for walk. A major scale defined using walk:



Silly-scale is a stockpile with different scale values using the interval pattern *1 2 2 1*:



Choosing among stockpiles
A few stockpiles have been defined during this chapter. Perhaps you want to make a section that uses one of these stockpiles for the first 25 notes, then chooses another stockpile for the next 25 notes, etc. This can be done with generator *select-stockpiles*. The syntax for this generator is a bit more complicated that the generators used so far in the tutorial.

A list of stockpiles can be expressed in this way:

```
(list pitch1 c-major2 silly-scale)
```

A choice can be made from this list with *series-choice*. No stockpile will be repeated until all have been used once.

```
(series-choice (list pitch1 c-major2 silly-scale))
```

Once a stockpile has been chosen, *random-choice* will be used to pick values from that stockpile. After 25 values have been chosen from the stockpile, a new stockpile is chosen.

```
(select-stockpiles 25 random-choice (series-choice (list pitch1 c-
major2 silly-scale)))
```

Additional features of *select-stockpiles* can be found in the help item in the application.

Section5 uses select-stockpiles to make random choices from three of the stockpiles discussed in this tutorial.

```
Name            section5
Clock unit      150
Number          100
Rhythm          (random-choice '(1 2 3))
Pitch           (select-stockpiles 25 random-choice
                   (series-choice (list pitch1 c-major2 silly-scale)))
Velocity        mf
Channel         1
```

Summary

Menu items
specify (stockpile), generate (stockpile), construct (stockpile), select ()

Generators
random-choice, major, on-the-fly, select-stockpiles, series-choice, series-value

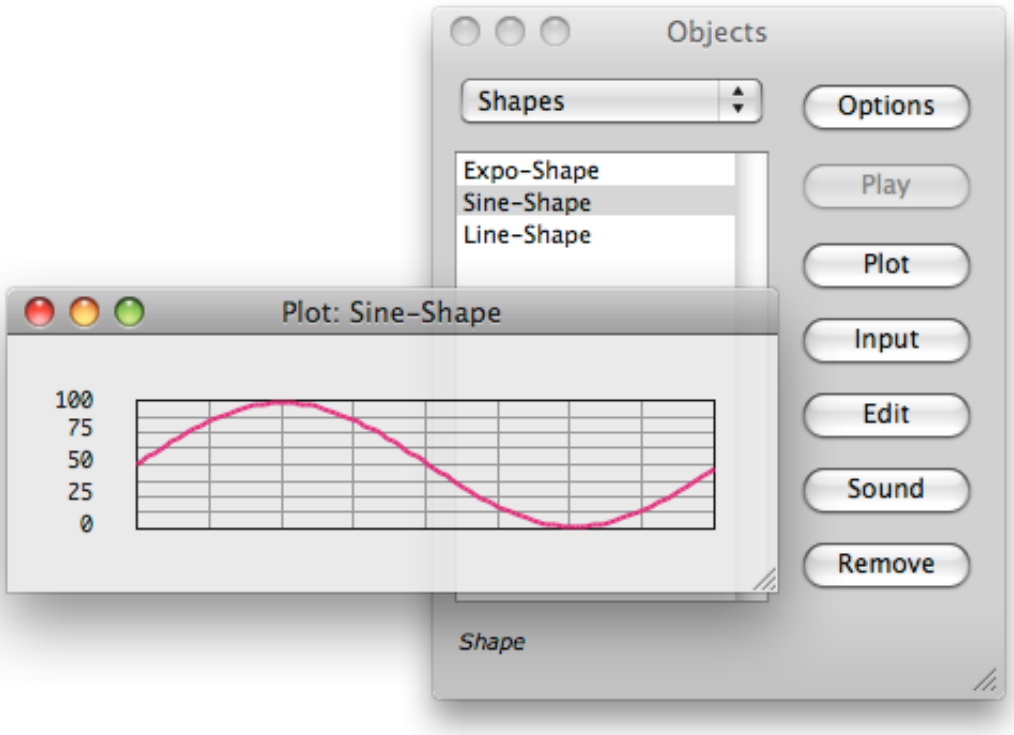
Tools
extract, multiple-bandwidths, generate-range

Tutorial 3

Shapes

A shape is an abstraction reflecting some motion over time. A shape can be drawn with a mouse, expressed as a series of evenly spaced values, or created with a generator. A shape is one way of reflecting movement in music. It can reflect trivial gestures such as 'going up' or more complicated movements. A shape as an AC Toolbox object does not have any particular meaning. It must first be converted to represent a certain kind of data within a certain range before it has a specific significance. Until then it is a gesture, waiting for an interpretation.

Help for shapes
Shapes is available as a popup menu item in the **Objects** table. A list of all defined shapes will appear. Some default shapes have been included in the Toolbox.



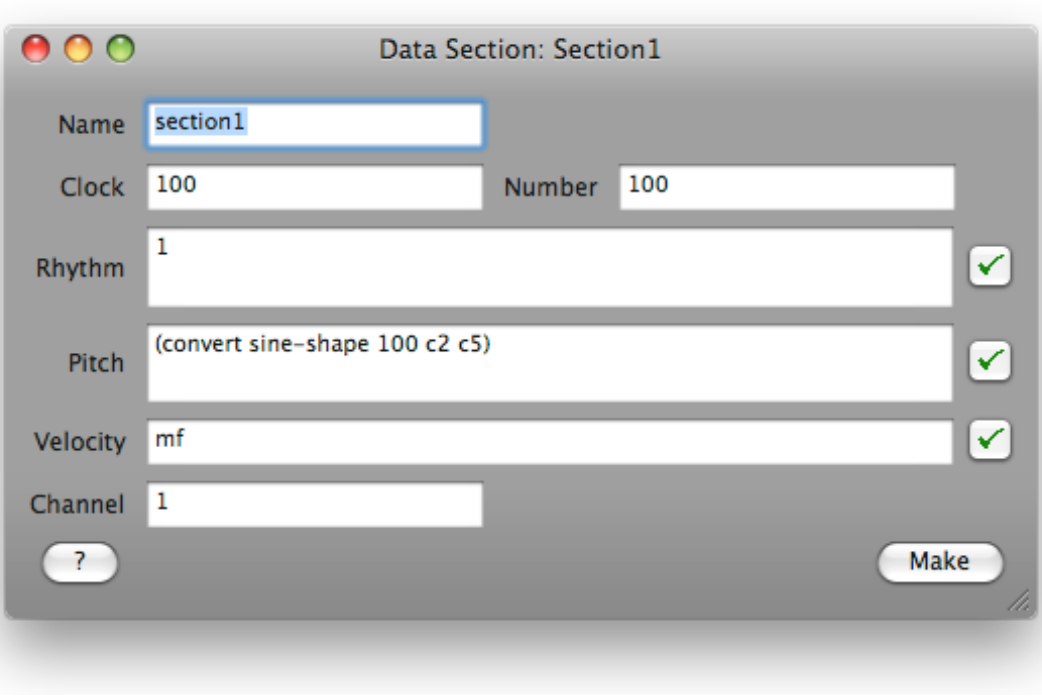
Converting a shape
A shape can be interpreted with *convert*, a tool that can convert a shape object to a list with a certain number of values within a certain range. The expression

```
(convert sine-shape  10 40 80)
```

will produce a list of 10 numbers reflecting a sine-shape between 40 80:

```
(60 73 80 78 68 54 43 40 46 59).
```

Convert can be used to translate *sine-shape* into a list of 100 pitches between c2 and c5.



The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 3 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

• Plot and play *section1* to verify that the pitches reflect the form of a sine.
• Plot the other default shapes.

To convert a shape, it is necessary to specify how many values should be produced using the shape. If a section of 100 values is made but convert produces 50 values from a shape, the shape is repeated twice in the section.

To automatically have *convert* produce the same number of values as the number of notes in a data section, the expression *(from-number)* can be used for the number parameter in convert. This tool determines how many values have been specified (or created) for the number parameter of the section and makes that number available for a tool, such as convert, that needs it. If the number parameter is changed, (from-number) will return the new value.

• Make *section2*. It is an example of using *from-number*.

```
Name            section2
Clock unit      100
Number          100
Rhythm          1
Pitch           (convert  sine-shape  (from-number)  c2  c5)
Velocity        mf
Channel         1
```

When a section uses an *until-time* expression for *number*, *(from- number)* will return the number actually needed to fill the specified amount of time.

• Make *section3* . It uses both *until-time* and *from-number*.

```
Name            section3
Clock unit      100
Number          (until-time 5)
Rhythm          1
Pitch           (convert sine-shape (from-number) c2 c5)
Velocity        mf
Channel         1
```

From-number can be abbreviated as *fn*.

Using a shape to read from a stockpile
Read-from is a generator that reads from various objects such as a list or stockpile. Various types of input can determine what is read. An argument could be a list of index values specifying which value in the list or stockpile should be read. The first element in a list is at index 1, the second at index 2, and so on.

• Type or drag the following expression in a **For Example** dialog (**Tools>For Example**) and look at the values produced:

```
(read-from  '(a b c) '(2 1 3 3 1 2 1 2 3))
```

Instead of using index numbers to read from a list, a shape could also be used. The name of the shape and the number of values to be read should be specified. The shape is converted to index values for the list or stockpile being read. Reading a list with a sine-shape means that reading will begin in the middle of the list, proceed to the end, then go back to the middle, go back to the beginning, and then proceed to the middle.

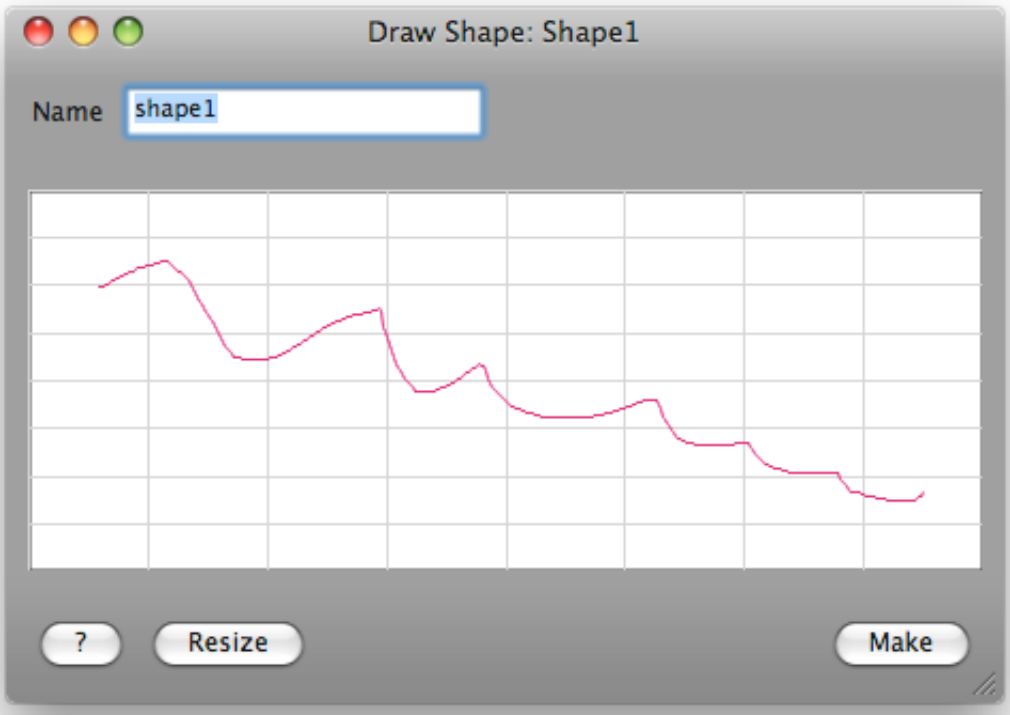
• Type or drag the following expression in a **For Example** dialog and look at the values produced:

```
(read-from  '(a b c d e f g) sine-shape  20)
```

Read-from can also deal with other types of objects as input besides lists and stockpiles. Other examples in later tutorials will demonstrate this. Examples can also be found in the help for read-from.

• Generate a stockpile with 24 values reflecting a major scale. Define a data section using *read-from* to choose the pitches according to a sine-shape.

Drawing a shape
Select **Define>Shape>Draw** to draw a shape. A grid appears where a shape can be drawn from left to right by holding the mouse button down and drawing. The drawing only goes from left to right. To change part of the line already drawn, click in the grid and begin drawing again, from left to right. Once a shape has been drawn, a name should be entered and *Make* clicked. There is no need to worry about where the drawing begins. When a shape is converted, only the shape is considered, not its position on the grid.

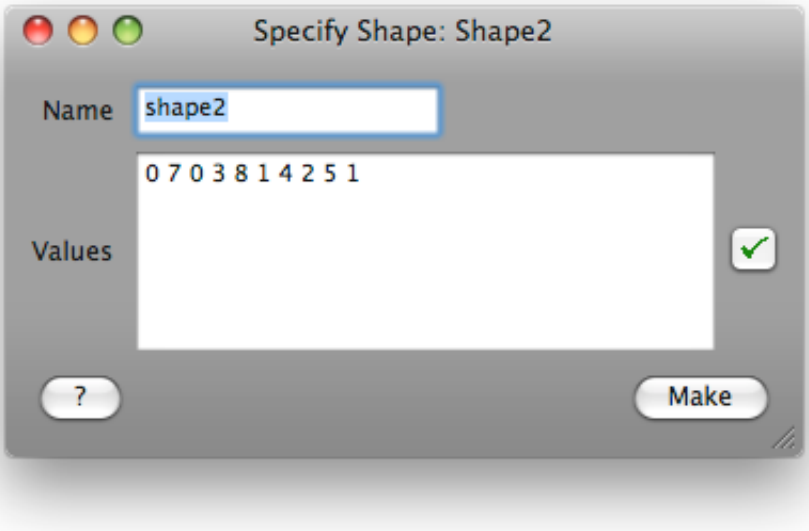


After a shape has been made, it can be edited in the same way as the original drawing, by clicking on some point in the drawing, holding the mouse down, and moving the mouse from left to right. To change the definition of a shape, *Make* must again be clicked.

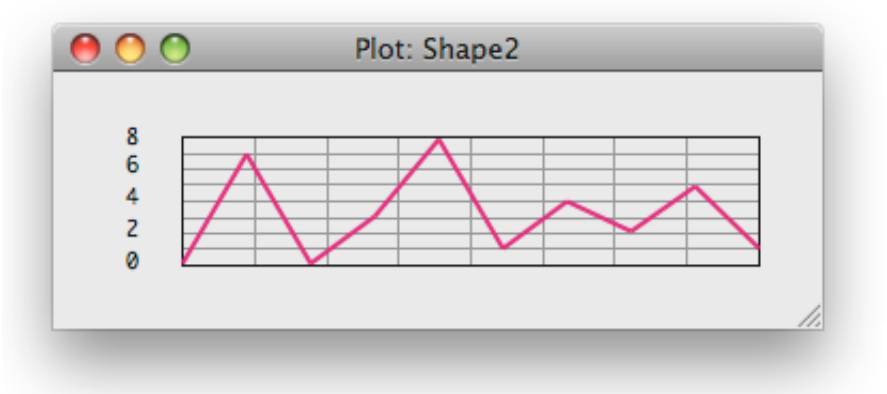
If the size of the drawing window is increased, the shape can continue in the new space. If the size of the drawing window is decreased, some of the shape may not be visible (though it will be saved). To rescale the shape to fit in the available window size, select *Resize*.

To change the color of the pen used for drawing, select another color in the **Preferences**.

Specifying a shape
Enter a series of numbers to specify a shape. These numbers are used to create a shape. When the numbers are entered, any range can be used.

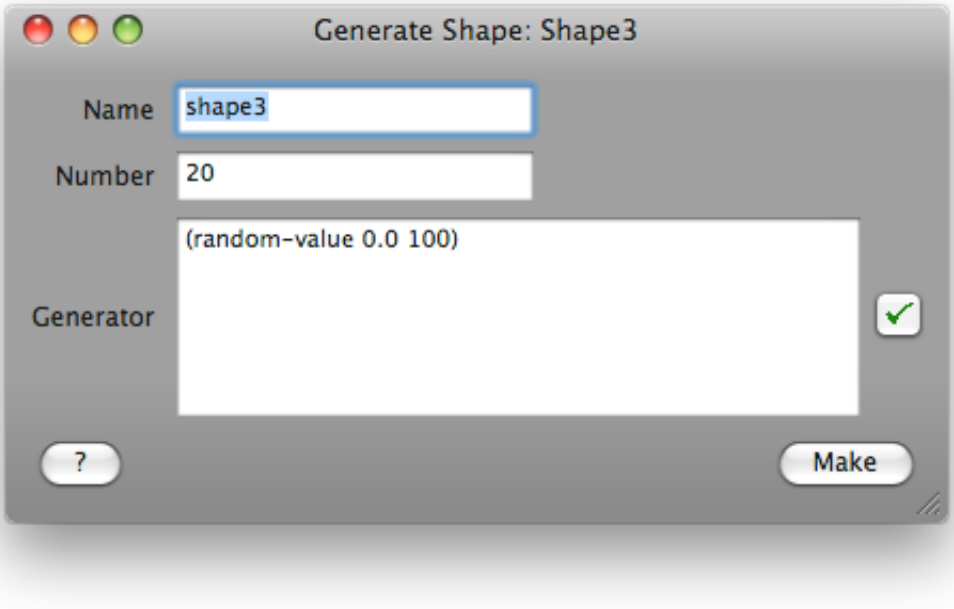


Displaying this shape reveals the following representation:

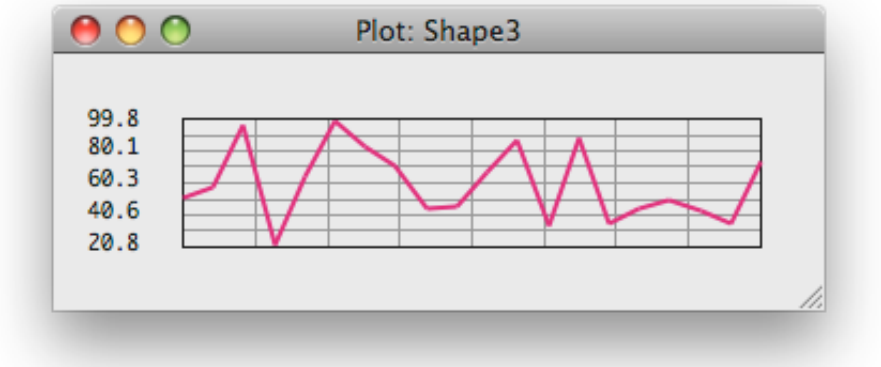


If the **Shapes** popup menu item in the **Objects** dialog is chosen, a button called **Edit** appears. If it is chosen, the shape is drawn on a grid and it can be modified by drawing changes with the mouse in the same way as with **Draw Shape**.

Generating a shape
A generator can be applied a number of times to create values that are considered to be a shape. The numbers produced by the generator can be in any range.



The dialog box above generated the following shape:



To regenerate the shape, (click the *Input* button on the table showing the available shapes and then) select *Make*.

If the *Make* button is clicked with the right mouse button (or CTRL-Click is used), *Edit* can be chosen from the popup menu. The shape will be drawn on a grid. It can be modified by drawing changes with the mouse**.**

Summary

Menu items
draw (shape), specify (shape), generate (shape).

Generators
read-from

Tools
convert, until-time, from-number

Tutorial 4

Masks

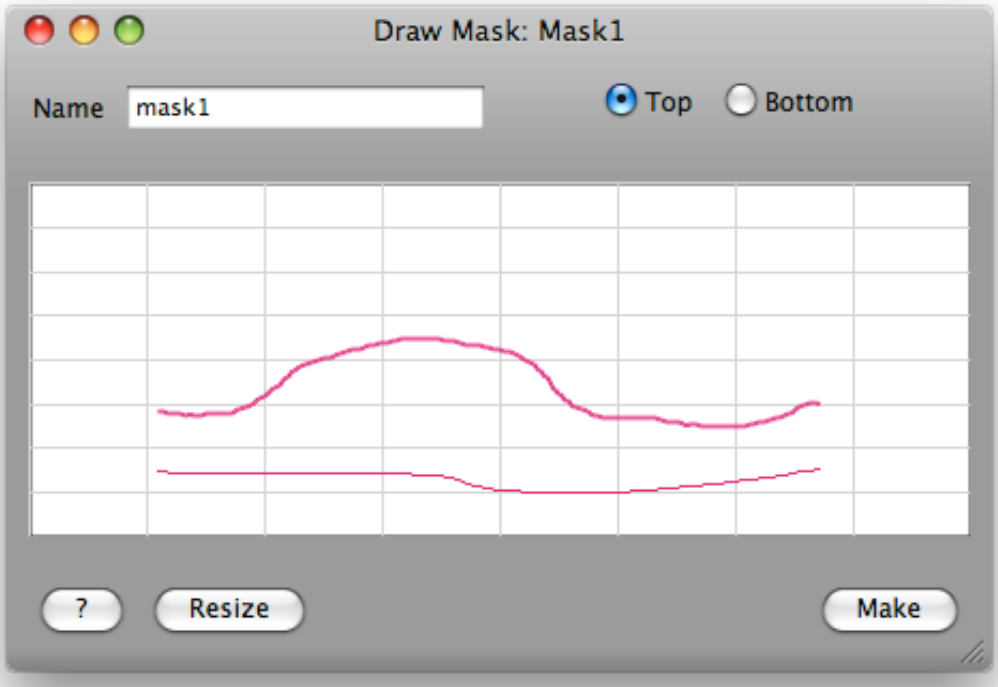
A mask is an abstraction represented by two lines. It is a field that changes over time. Events can happen in the area between the lines. This requires interpretation. A mask, once interpreted, can represent the tendency of certain events to happen.

A mask can be drawn with the mouse, specified as a series of evenly spaced numbers for each line, or created with a generator for each line.

A mask is related to G. M. Koenig's concept of a tendency mask that he formulated for his composition program *Project 2*. His concept has since been adapted and used by several composers including Barry Truax and Jean Piché.

Drawing a mask
To draw a mask, select **Define>Mask>Draw**. A grid appears where two lines can be drawn from left to right by holding the mouse button down and drawing. The first line is considered to be the *top* of the mask. To draw the *bottom* of the mask, click on the bottom button before drawing. To edit a line, choose *Top* or *Bottom* and then redraw (part) of the line from left to right. The selected line (Top or Bottom) is thicker.

Once the drawing has been finished, enter a name for the mask and click on *Make*. There is no need to worry about where in the grid the mask begins. When a mask is converted, only the lines of the mask are considered, not their position on the grid.



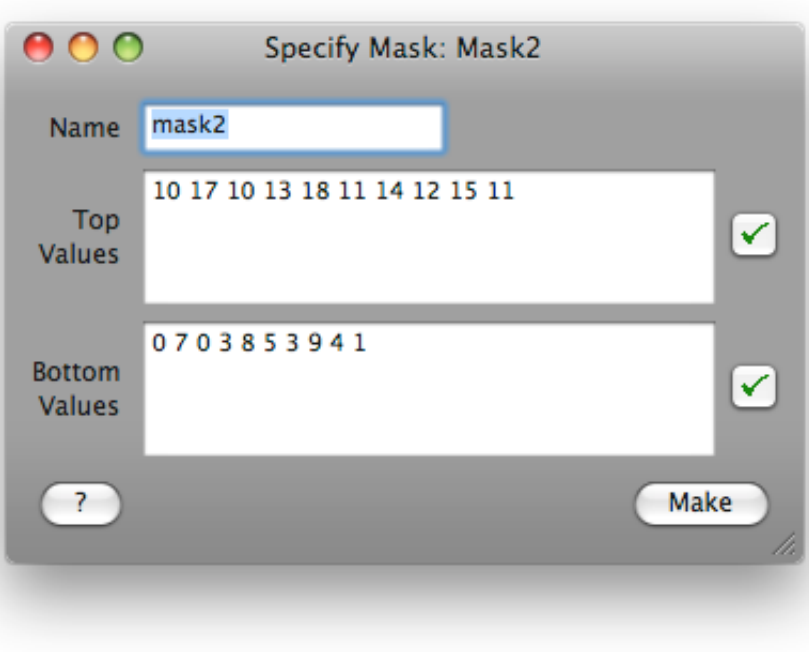
After a mask has been made, it can be edited in the same way as the original drawing by selecting *Top* or *Bottom,* clicking on some point in the drawing, holding the mouse down, and moving the mouse from left to right. To change the definition of a mask, *Make* must again be clicked.

If the size of the drawing window is increased, the mask can continue in the new space. If the size of the drawing window is decreased, some of the mask may not be visible (though it will be saved). To rescale the shape to fit in the available window size, select *Resize*.

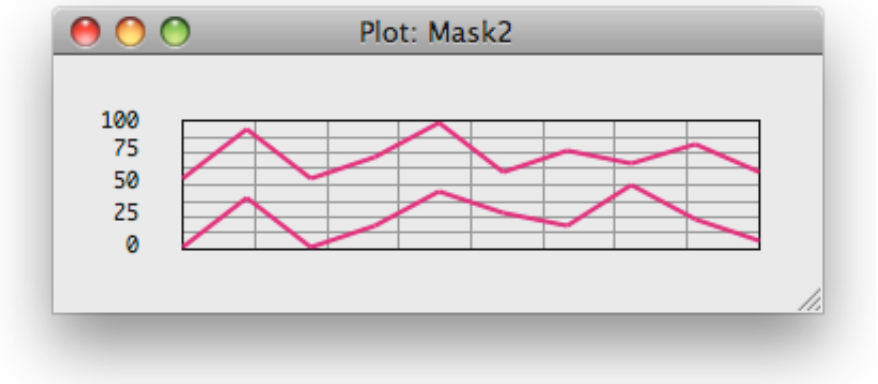
To change the color of the pen used for drawing, select another color in the **Preferences**.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 4 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Specifying a mask
A mask can be specified with a series of numbers in any range. These numbers are used to create the two lines of the mask.

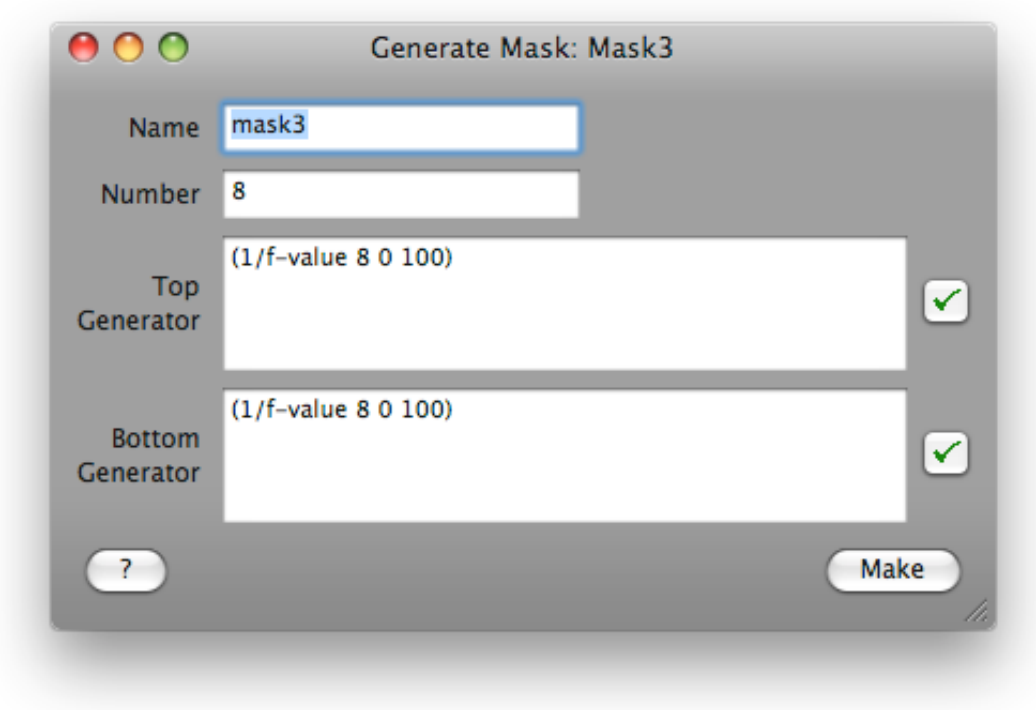


Displaying this mask produces the following representation:

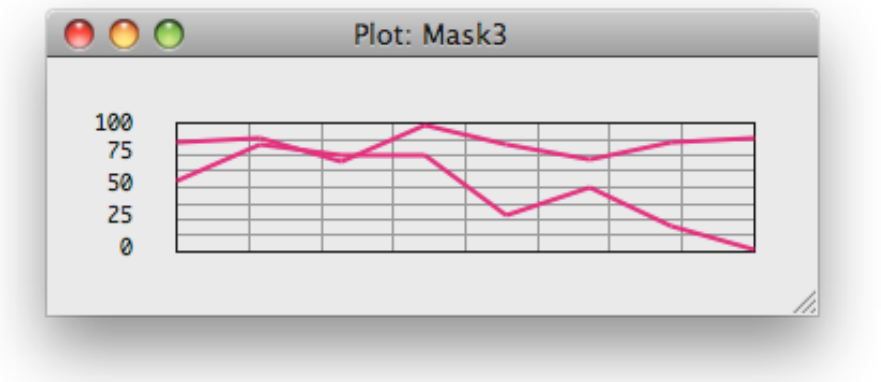


If the **Masks** popup menu item in the **Objects** dialog is chosen, a button called **Edit** appears. If it is chosen, the mask is drawn on a grid and it can be modified by drawing changes with the mouse in the same way as with **Draw Mask**.

Generating a mask
A generator can be applied a number of times to create values that describe the lines of the mask. The values can be in any range.



The following mask was generated with the above description:



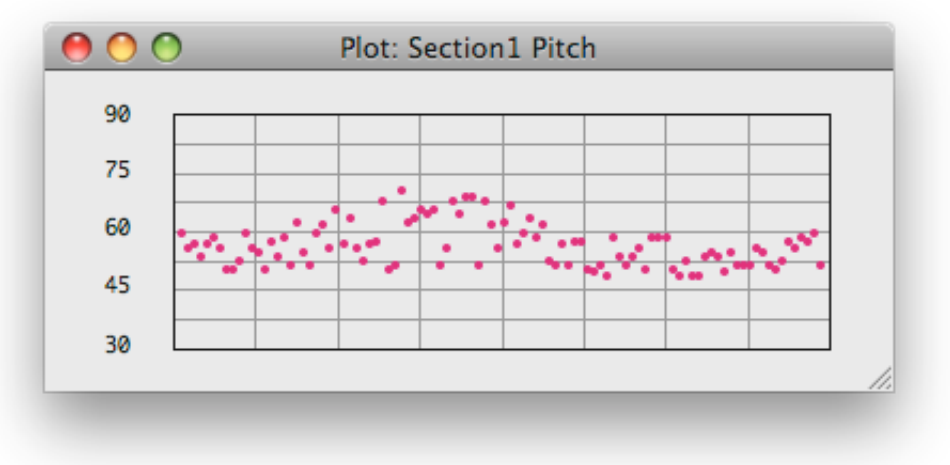
If the *Make* button is clicked with the right mouse button (or CTRL-Click is used), *Edit* can be chosen from the popup menu. The mask will be drawn on a grid and it can be modified by drawing changes with the mouse in the same way as with **Draw Mask**.

Converting masks
A mask can be interpreted with *convert*. The lines of the mask will be translated to some range, e.g. in *section1* the lowest point will be 48 and the highest point will be 72. The lines will determine boundaries between which a random value is chosen. A number of these values will be gathered into a list.

Section1 uses a mask to produce pitches for a data section:

```
Name            section1
Clock unit      100
Number          100
Rhythm          1
Pitch           (convert  mask1 100 48 72)
Velocity        mf
Channel         1
```

Convert produced 100 values between 48 and 72 that will be used as pitches in *section1*. The 100 values were chosen at evenly spaced intervals along the horizontal axis of the mask.



To convert a mask, it is necessary to specify how many values should be produced using the mask. If a section of 100 values is made but convert produces 50 values from a mask, the mask is repeated twice in the section.

To automatically have convert produce the same number of values as the number of notes in a data section, the expression *(from-number)* can be used for the number parameter in convert. This tool determines how many values have been specified (or created) for the number parameter of the section and makes that number available for the tool, such as convert, that needs it.

Section 2 uses *(from-number)*:

```
Name            section2
Clock unit      100
Number          100
Rhythm          1
Pitch           (convert mask1 (from-number) 48 72)
Velocity        mf
Channel         1
```

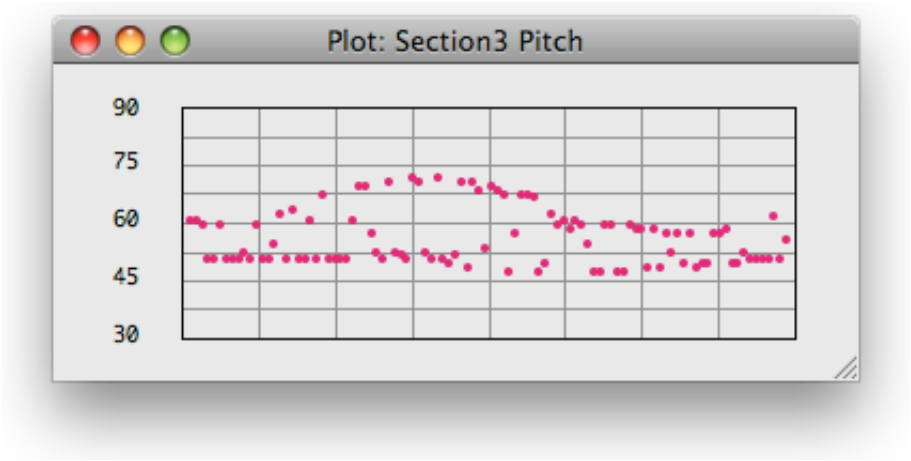
- Make *section2*.
- Define some masks and use them to describe pitch values in a data section.
- Use masks to describe changes in rhythm and velocity.

To use a mask to block some area, see the help for the tool *2-masks* in the application. It will generate values with one mask and block them with another.

With a different generator
By default, values are chosen within the boundaries of the mask using *random-value*. It is possible to use another generator to indicate the kind of choice to be made between the boundaries. This generator should produce values between 0.0 and 100, reflecting a position between the lower boundary (0) and the upper boundary (100).

```
Name            section3
Clock unit      100
Number          100
Rhythm          1
Pitch           (convert  mask1 100 48 72
                   (beta-value 0.0 100 0.1 0.1))
Velocity        mf
Channel         1
```

The generator *beta-value* was added to the end of convert. This statistical generator produces values between 0.0 and 100. Most of the values will be near 0 or 100. The distribution of pitches then will tend to follow the form of the mask and there will be few values chosen between the boundaries.



Rounding the values
Masks can be converted and the results rounded with tool *convert2.* It is similar to *convert* but it uses keywords instead of optional parameters. It has a keyword *round* that can be bound to a quantization unit.

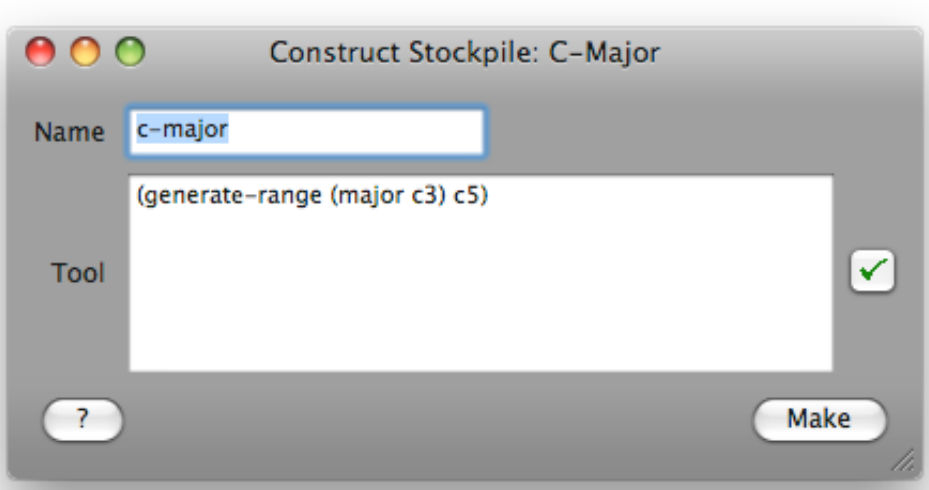
```
List returned from:
(convert2 mask2 20 40.0 80 :round 0.5)

(
    48.500    70.500    72.000    51.000    61.000
    60.000    62.500    65.500    80.000    65.500
    62.500    61.500    69.500    61.000    65.000
    64.500    50.000    68.000    56.500    51.500
)
```

Section4 quantizes rhythm to units of 0.25 and pitch to units of a quarter-tone.

```
Name            section4
Clock unit      100
Number          100
Rhythm          (convert2 mask1 100 1.0 3 :round 0.25)
Pitch           (convert2 mask1 100 48 72.0 :round 0.5)
Velocity        mf
Channel         1
```

Some other possibilities for limiting the intervallic content or the possible values when converting a mask can be found in Tutorial 24.

Masking a stockpile
If the pitches should be limited to some collection, such as those in a C major scale between c3 (48) and c5 (72), a stockpile can be defined with only those pitches.



A mask can control the choice of values read from this stockpile. The values derived from the mask represent indices for reading from the stockpile. The lowest value produced by the mask returns the first value from the stockpile. The highest value produced by the mask returns the last value from the stockpile.

```
Name            section5
Clock unit      100
Number          100
Rhythm          1
Pitch           (read-from  c-major  mask1)
Velocity        mf
Channel         1
```

In *section5,* pitches will follow the shape of the mask, but only pitches that are part of a C major scale will be chosen.

Read-from assumes that a mask will be used to produce 100 values. To produce some other number of values with a mask, an additional argument must be given for that number:

```
(read-from  c-major  mask1 20)
```

To use a different generator than *random-value* when reading with a mask, the generator should be included after the number argument in *read-from*. It should produce a value between 0 and 100.

```
(read-from  c-major  mask1 100 (beta-value 0.0 100 .1 .1))
```

Summary

Menu items
draw (mask), specify (mask), generate (mask)

Generators
beta-value, read-from

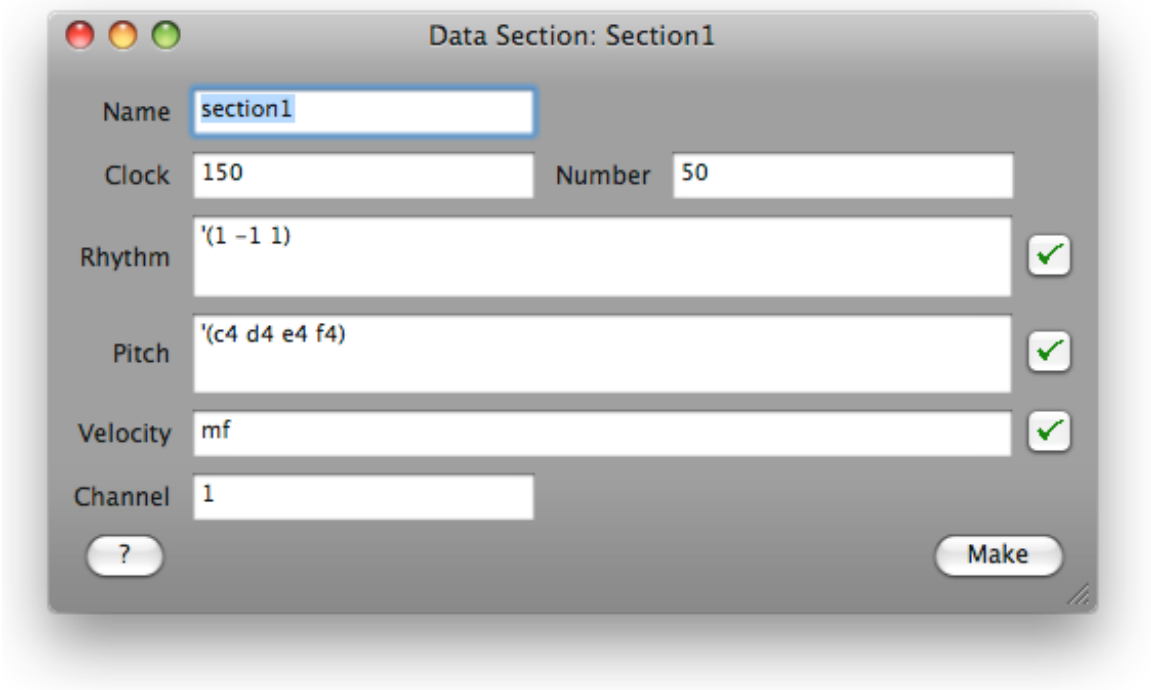
Tools
convert, convert2

Tutorial 5

Chords, rests, and transpositions

Rests

Rests are represented in the AC Toolbox as negative numbers. In a data section, rhythms are multiples of a clock unit. Rests are also considered multiples of the clock unit. The absolute value is the length of the rest. For example, if the clock unit is *100* milliseconds and the rhythmic values are *'(1 -1 2 -3)*, the durations of the notes would be: note for 100 milliseconds, rest for 100 milliseconds, note for 200 milliseconds, rest for 300 milliseconds.



A rest is not inserted between notes. A rest causes a note that is produced not to be heard. That is the reason for the following pattern in the above example:

```
c4  rest  e4  f4  rest  d4  e4  rest  c4  d4  rest  f4  c4  rest ...
```

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 5 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

A note with a rest also counts as one of the number specified in the dialog box. If 50 notes are requested, fewer will sound if some of the notes are rests. The section object only stores the notes that are not rests. In *section1*, only 33 notes are stored.

Rests can be used anywhere that a positive rhythmic value can be used. Zero should not be used as a rhythmic value.

```
Name            section2
Clock unit      150
Number          50
Rhythm          (random-choice  '(1  -1))
Pitch           '(c4  d4  e4 f4)
Velocity        mf
Channel         1
```

Generators for rests

Plus-min is an useful generator for creating rests. Arguments are a generator etc. for producing a positive value and a value between 0 and 1 describing the probability that the outcome should be made negative (i.e. should be a rest). For the probability argument, zero is *never* and one is *always*.

```
20 values from:
(plus-min (random-value 1.0 3) 0.4)
    -2.273     2.358    -1.533     1.774    -1.069
     1.377     1.941     2.468     1.976     2.924
     2.601     2.435    -2.737     1.262    -1.774
    -1.391     2.654     2.629    -1.807    -2.488
```

Section2b uses plus-min to create rests.

```
Name            section2b
Clock unit      150
Number          50
Rhythm          (plus-min (random-value 1.0 3) .4)
Pitch           '(c4  d4  e4 f4)
Velocity        mf
Channel         1
```

To avoid losing pitch values when there is a rest, use the generator *skip-rests*. The duration values can be first made in a stockpile. That stockpile can be used for duration in the section and as input to skip-rests to determine if a value is to be used or not.

Rhythm is a stockpile with a 100 values made with plus-min. 100 values are used so that the stockpile will not have to be repeated to produce enough values for the section.

Skip-rests requires something with rests (such as a stockpile) and a generator etc. to produce the new values if there is no rest. In *section2c*, the something with rests will be the stockpile *rhythm*. The new values will be taken from the list `'(c4 d4 e4 f4)`. Using skip-rests, a new value is only taken when there is no rest. This means no desired pitch values will be 'lost'.

```
Name            section2c
Clock unit      150
Number          50
Rhythm          rhythm
Pitch           (skip-rests rhythm '(c4  d4  e4 f4))
Velocity        mf
Channel         1
```

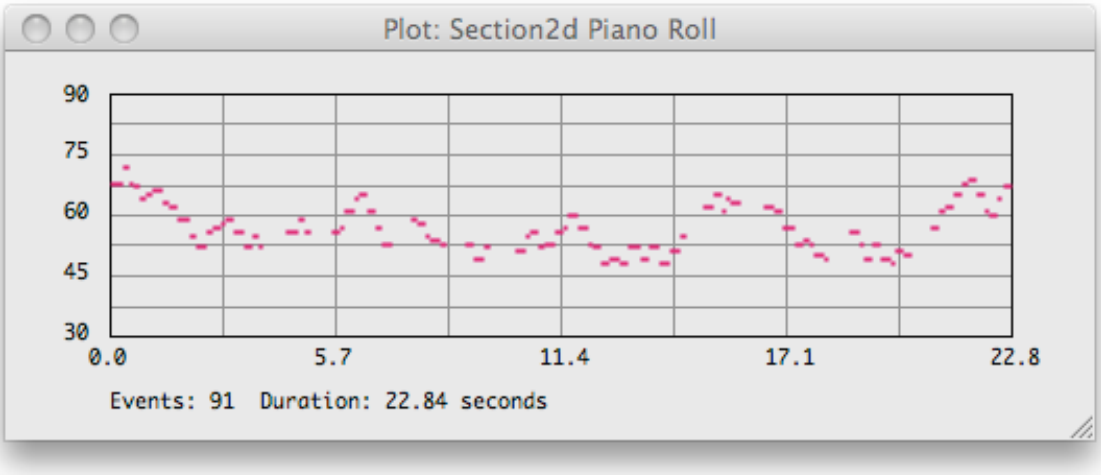
Take can be used to make rhythmic groups. Take will return a number of values from one generator, then some more from another, etc. When all number/generator pairs have been used, the process starts again from the beginning.

```
20 values from:
(take 5 (random-value 60 72) 10 (random-value 40 50))
        67        70        62        68        71
        47        44        49        48        44
        42        48        44        44        49
        72        66        61        66        68
```

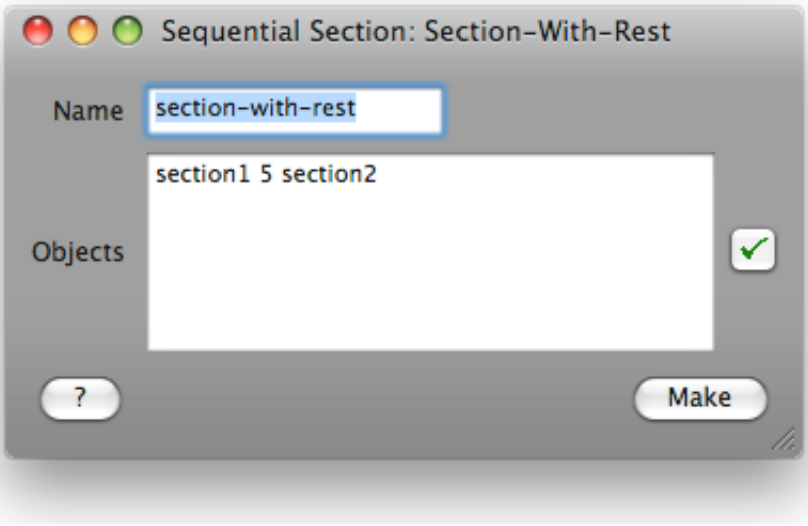
In the above example, 5 values are made with (random-value 60 72), then 10 values are made with (random-value 40 50), and then 5 values with (random-value 60 72).

One use of *take* would be to produce some positive rhythmic values followed by only one negative value. This would produce a group of notes, a rest, another group of notes, another rest, etc. In *section2d*, the number for the positive rhythmic values is chosen with (series-choice '(3 5 8 20)). The positive rhythmic values are chosen with (rv 1.0 3). There is 1 rest. It is chosen with (rv -4.0 -7.0).

```
Name            section2d
Clock unit      100
Number          100
Rhythm          (take (series-choice '(3 5 8 20)) (rv 1.0 3)
                      1 (rv -4.0 -7))
Pitch           (random-intervals c3 c5 1 3 4)
Velocity        mf
Channel         1
```



Rests between sections
Rests can inserted between sections when a sequential section is specified. In the dialog box for a sequential section, a length in seconds between the sections can be indicated. In the section *section-with-rest*, 5 seconds of silence separates the two sections.



Chords

In a data section specification, a chord is a list of pitches instead of one Midi note number or pitch symbol. If the pitch parameter contains *'((c4 e4 g4) (c4 d4 e4 f4) (d4 g4 b4))*, the first event is a chord with the pitches c4, e4, g4. The entire chord has only one duration and one velocity value.

```
Name            section3
Clock unit      150
Number          50
Rhythm          (random-choice  '(1 -1))
Pitch           '((c4 e4 g4) (c4 d4 e4 f4) (d4 g4 b4))
Velocity        mf
Channel         1
Delay           0
```

In *section4*, the chords are chosen at random from a list of three possible chords:

```
Name            section4
Clock unit      150
Number          50
Rhythm          (random-choice  '(1 -1))
Pitch           (random-choice  '((c4 e4 g4) (c4 d4 e4 f4) (d4 g4
b4)))
Velocity        mf
Channel         1
```

Generating chords

Make-chord is a generator for chords. The first argument is a list, stockpile, or generator that can supply the pitch values for a chord. The second argument indicates how many pitches should be in one chord. It can be specified as a constant, list, generator, etc.

The pitches produced by the first arguments are gathered in the order in which they are produced. For example, *(make-chord '(60 62 64 65 67 69 71) 3)* produces the following series of chords:

```
(60 62 64)
(65 67 69)
(71 60 62)
(64 65 67)
(69 71 60)
(62 64 65)
...
```

• Define the *c-major* stockpile object (from the *Tutorial 5 Examples* file). It is used in the definition of *section5*.

The pitch specification for *section5* is:

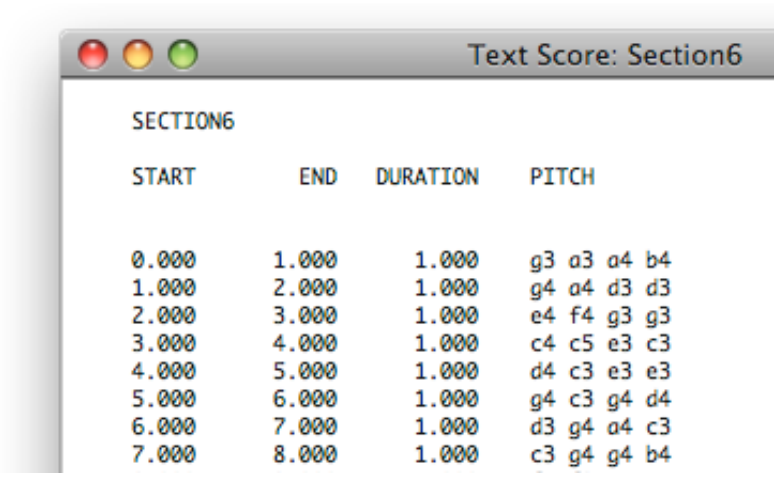
```
(make-chord c-major 2).
```

Two-note chords are gathered from the c-major stockpile, in the order in which they occur in the stockpile.

In *section6*, pitches for the chords are randomly chosen from the pitches available in the c-major stockpile. The pitch specification is:

```
(make-chord (random-choice c-major) 4).
```

In addition to listening and plotting this section to inspect the way that *make-chord* works, a simple, no-nonsense text 'score' of the section can be made via **Tools>Text Score.** An example of part of the output for *section6:*



Another way to examine the data is to select the section name in the Objects dialog, and then click on the *Edit* button.

In *section6b*, a key word is used with *make-chord* to block the repetition of the same note in a chord. To do this, interval 0 should be blocked.

```
(make-chord (random-choice c-major) 4 :block 0)
```

To prevent pitch repetitions in successive chords, the key word *allow-repeats* can be bound to *nil.* This is done in *section6c*.

```
(make-chord (random-choice c-major) 4 :block 0 :allow-repeats nil)
```

See the help text for *make-chord* for more information on blocking intervals or filtering repetitions.

In *section7*, both the pitches and the chord size are randomly generated for each chord.

```
Name            section7
Clock unit      250
Number          50
Rhythm          (random-choice  '(1 -1))
Pitch           (make-chord  (random-value c3 c5)
                             (random-value 1 5)
                             :block 0)
Velocity        mf
Channel         1
```

Mc is the shortcut for *make-chord.*

```
20 values from:
(mc (rv 40 80) 3)
    ( 54 42 70 ) ( 68 61 73 ) ( 79 51 73 ) ( 79 66 74 ) ( 46 67 48 )
    ( 47 53 77 ) ( 47 47 60 ) ( 67 58 66 ) ( 57 67 49 ) ( 73 58 74 )
    ( 65 70 80 ) ( 73 72 55 ) ( 79 51 50 ) ( 50 48 66 ) ( 65 45 42 )
    ( 53 70 71 ) ( 67 58 52 ) ( 78 62 75 ) ( 40 79 62 ) ( 59 76 71 )
```

Transpositions

Pitch classes and octaves

Some people like to talk about pitch classes. A *pitch class* is a pitch without an octave indication. *C* instead of *c4, c#* instead of *c#2*. A pitch class can be represented by a number. 0 for c, 1 for c#, 2 for d, etc.

People organizing pitch classes may generate pitches and their octaves separately. The Toolbox generator that does this is *pitch-and-octave*. The first argument is a list, stockpile, generator, etc. for deriving the pitch class. Regardless of what note numbers are produced by this argument, they will be reduced to the range 0 - 11. The second argument is a list, stockpile, generator, etc. to specify the octave. The octave starting at middle C is octave

4, though this can be changed in the **Preferences** to be 3. For each pitch, *pitch-and-octave* will generate a pitch class value and an octave value and then combine them into a Midi note number.

For example, 10 values from *(pitch-and-octave '(60 61 62 63) '(4 5 1 2 3))*:

```
60    73    26    39    48    61    74    27    36    49.
```

'(60 61 62 63) is the same as '(c4 c#4 d4 d#4). The 10 values produced above are the same as

```
c4    c#5   d1    d#2   c3    c#4   d5    d#1   c2    c#3.
```

In *section8*, the pitches of a C major scale, are randomly assigned octaves between 3 and 5.

```
Name            section8
Clock unit      200
Number          50
Rhythm          1
Pitch           (pitch-and-octave  c-major  (random-value  3 5))
Velocity        mf
Channel         1
```

In a twelve-tone series, each pitch class is used once before any is repeated. A generalization of this idea is to make a random choice from available values but without repeating any of the values until all have been chosen once. *Series-value* is the name of the generator that does this within some bandwidth.

For example, 10 values from *(series-value 0 4)*:

```
1 3 2 0 4 0 1 4 3 2
```

In *section9*, series-value is used to determine the pitch class. The pitch specification is:

```
(pitch-and-octave (series-value 0 11) (random-value 3 5))
```

The shortcut for *pitch-and-octave* is *spread*.

```
20 values from:
(spread (rv 0 11) (rv 3 5))
        70        64        55        80        62
        56        66        82        76        83
        72        74        60        55        74
        51        80        50        59        70
```

Per note

Each value in a list or stockpile can be transposed using *on-the-fly*. If the number being added is a constant, all values in the list or stockpile will be transposed by the same amount.

```
20 values from:
(on-the-fly '(c4 d4 f4) + 12)
        72        74        77        72        74
        77        72        74        77        72
        74        77        72        74        77
        72        74        77        72        74
```

If the number being added is a generator, each note will be transposed by a different amount. This is what happens in *section10.*

```
Name            section10
Clock unit      120
Number          36
Rhythm          1
Pitch           (on-the-fly  '(c4  d4  f4) + (random-value –12 12))
Velocity        mf
Channel         1
```

This form of transposition may diminish the coherence of the original list or stockpile, depending on the chosen transposition values.

Per group
Another way of transposing is with the generator *transform-material*. The entire stockpile is transposed with an interval and then the entire stockpile is transposed by the next interval, etc. The first argument is the material to be transformed. The second argument is a list, stockpile, generator, etc. specifying the intervals of transposition.

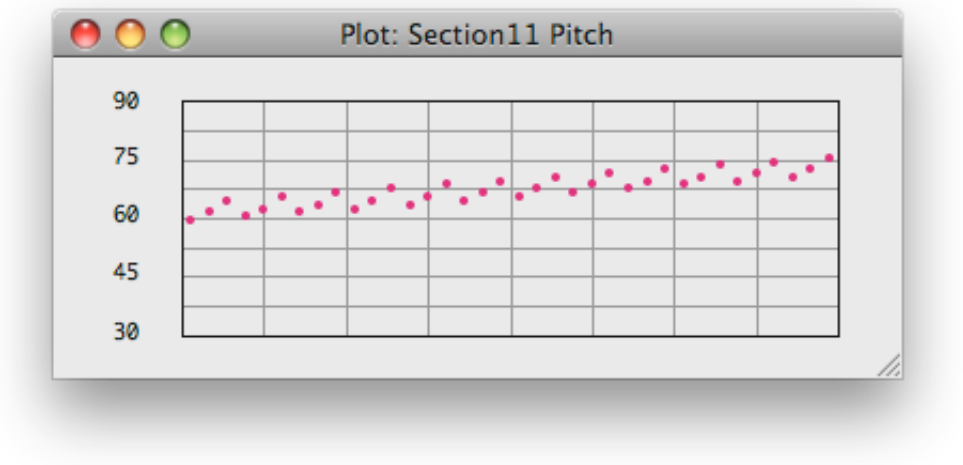
For example, 18 values from *(transform-material '(0 1 4) '(0 1 2 5))*:

```
0       1       4       1       2       5       2       3       6
5       6       9       0       1       4       1       2       5
```

First 0 was added to each value, then 1, 2, 5, and then the process started all over again with 0.

```
Name            section11
Clock unit      120
Number          36
Rhythm          1
Pitch           (transform-material  '(c4  d4  f4) (from  0 11))
Velocity        mf
Channel         1
```

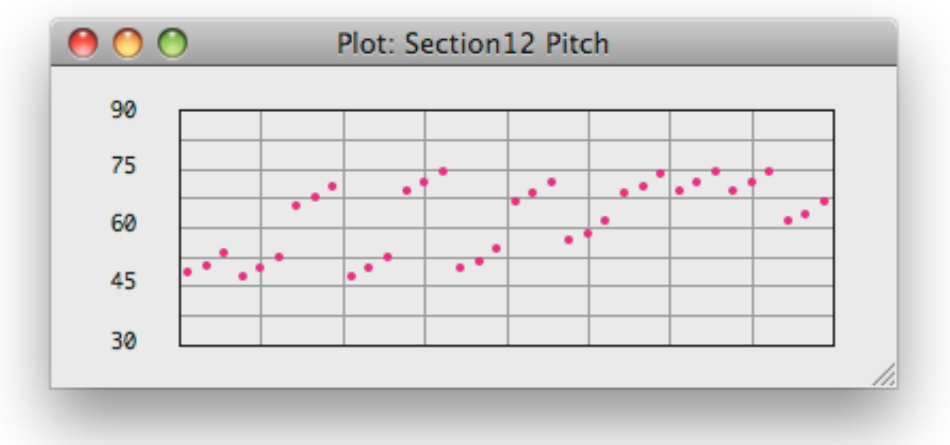
(from 0 11) produces a sequence of values from 0 to 11. The three-note pattern is first presented without transposition (0), and then is transposed up a semi-tone 11 times.



For other possible transformations, see the help for transform-material in the application.

In *section12*, the transposition interval is random within an octave on either side. The pitch specification is:

```
(transform-material '(c4 d4 f4) (random-value -12 12)).
```



• Make the *series1* stockpile object. It is used in the pitch specification for *section13*. Each note from the stockpile (an eleven-note series) is transposed according to another unique series.

```
Name            section13
Clock unit      120
Number          132
Rhythm          1
Pitch           (transform-material  series1
                      (series-value  0 11))
Velocity        mf
Channel         1
```

Other ways to transform material are discussed in Tutorial 7 and in Tutorial 25. Transposing pitch with a matrix is discussed in Tutorial 24.

Summary

Menu items
Text score

Generators
make-chord, pitch-and-octave, transform-material, series-value, plus-min, skip-rests

Tools
from

Miscellaneous
rests, chords, text score

Tutorial 6

Making variants, editing objects

Variants
AC Toolbox objects are declared by specifying rules. The dialog boxes aid in entering the rules. The rules may be trivial or evocative. If the rules involve any indeterminacy, applying the same rules another time could lead to a different result.

Each AC Toolbox object records the rules by which it was specified (input) and the specific results produced by using those rules (output). The general (the input) and the specific (the output) are both maintained. This allows the user to specify objects using more general formulations and to always be able to see how an object was made. It also allows the user to keep a specific instance of the application of those rules. Composition sometimes requires both good design and good luck.

A *variant* of an object is an object of the same type but possibly with different values produced by the same rules. A new object is produced by applying the rules used to create the old object. Any object whose rules include any indeterminacy can be varied. (Objects without indeterminacy can also be varied but this has no point since the result will always be the same).

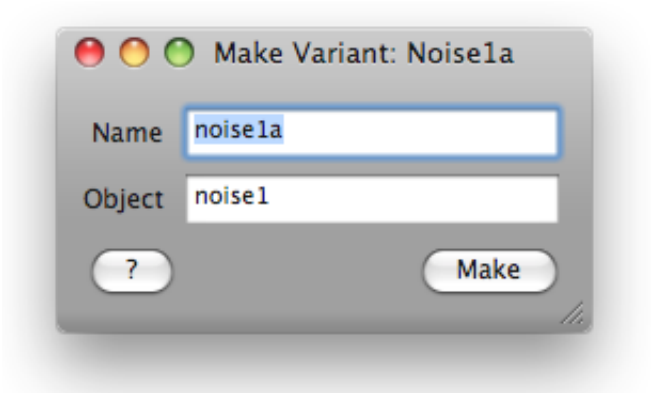
The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 6 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *make* in the dialog box.

The following data section will be varied:

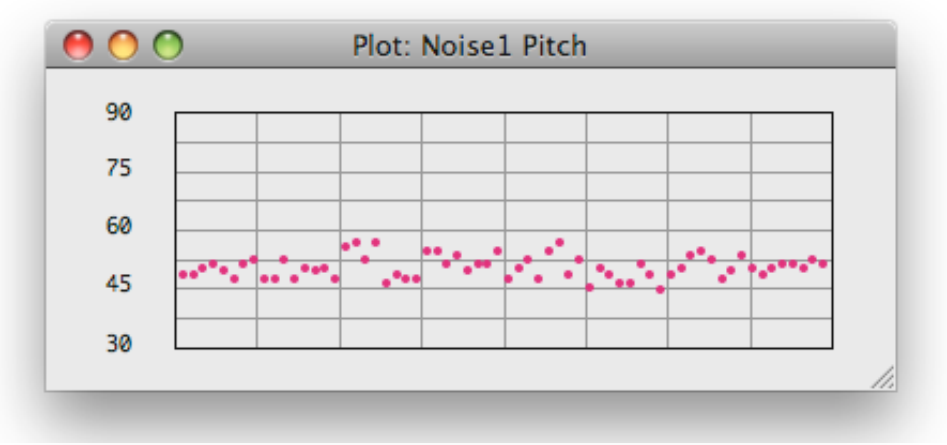
```
Name            noise1
Clock unit      150
Number          64
Rhythm          (1/f-value  64 0.5 2)
Pitch           (1/f-value  64 c2 c5)
Velocity        (1/f-value  64 30 90)
Channel         1
```

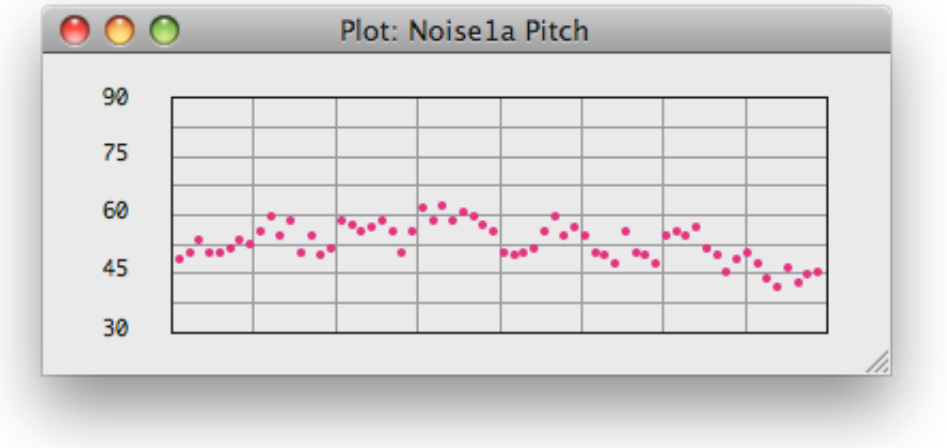
1/f-value is a generator for a type of fractional noise. Its arguments are the number of values to be produced, a lower limit, and an upper limit. The lower and upper limits are seldom reached.

A variant can be specified by choosing **Make Variant** in the **Methods** menu.



The two objects, *noise1* and *noise1a* were made using the same rules. The results of using those rules were different. This can be verified by listening to the two sections. Examining the pitches of both sections shows general similarities and specific differences.

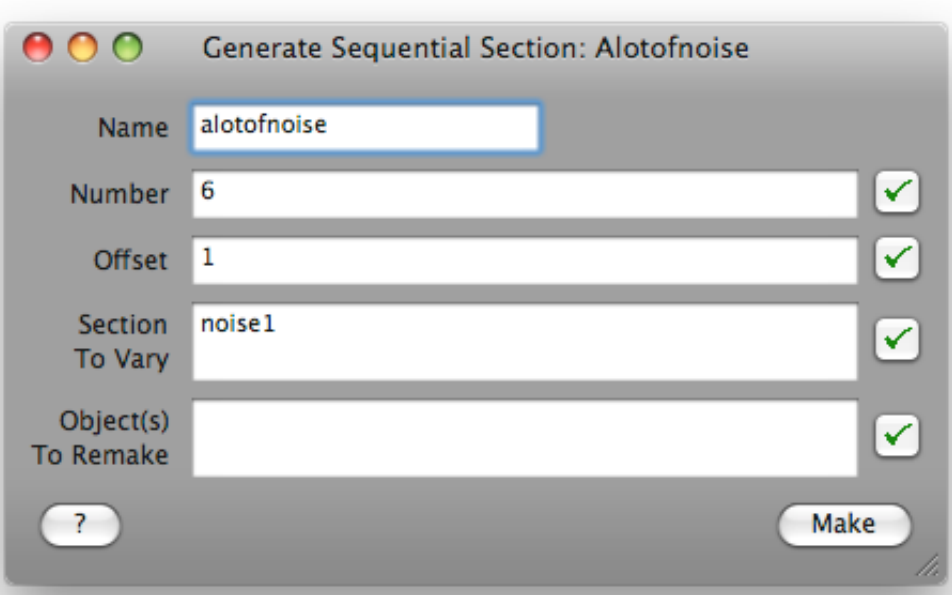




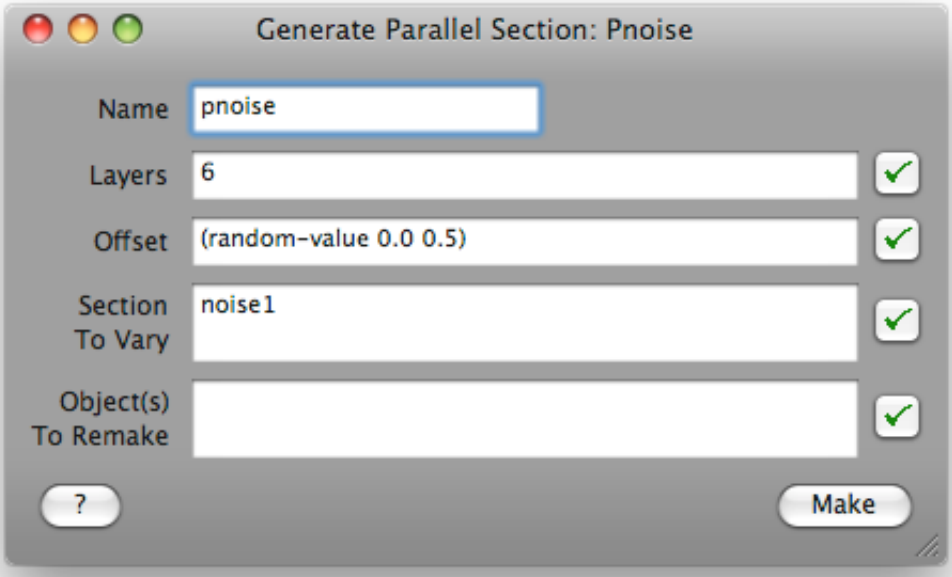
Variants can be made of other types of AC Toolbox objects including shapes, masks, stockpiles, and sections.

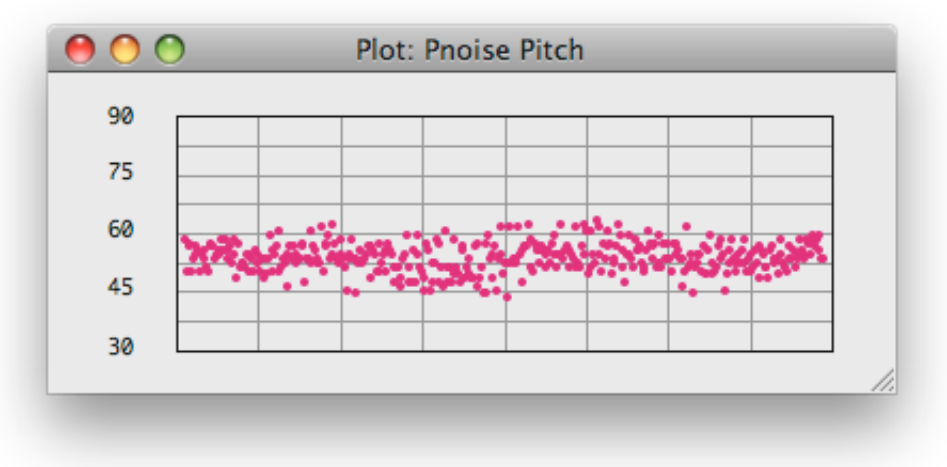
Combining variants in a section
Several variants of one section can be generated and combined into one section using **Define>Combination>Generate Sequential Section**. With this, it is possible to specify one or more sections to vary and then to join the results into one section.

Number is the number of variants to be generated. *Offset* is a time delay in seconds between the variants. This value can be a constant value, generator, etc. *Section to Vary* is the name of a section, a list of sections, a generator that could produce the name of a section, etc. The resulting section(s) will be varied and the results joined into a new section. The remaining parameter will be discussed later.



Several variants can be generated and joined in parallel using **Define>Combination>Generate Parallel Section.** Because the variants will be joined in parallel (that is on top of each other), *Offset* refers not to the time between variants but the time from the start of the resulting section. In section *pnoise,* each variant will start within the first .5 seconds of the new section.





Remaking values per variant
An expression such as *(random-value 1 10)* returns a type of function called a lexical closure that must be applied in order to get the random value between 1 and 10. When specifying input for a Toolbox dialog box, it is often not necessary to worry about this application since you are specifying rules and the Toolbox takes care of the rest. Occasionally a situation may arise when you do want to worry about this.

One way to apply the function returned by an expression such as *(random-value 1 10)* is to use *make*. The following two expressions can be evaluated in the Listener (**Other>Listener**) and the results examined:

```
CL-USER 1 > (random-value 1 10)
#<Closure (RANDOM-VALUE . 1) 200EABDA>
CL-USER 2 > (make (random-value 1 10))
1
```

(random-value 1 10) returned a closure. *(make (random-value 1 10))* returned a value.

Make should only be applied to generators, that is those functions included in the list of generators. *Make* should not be applied to tools.

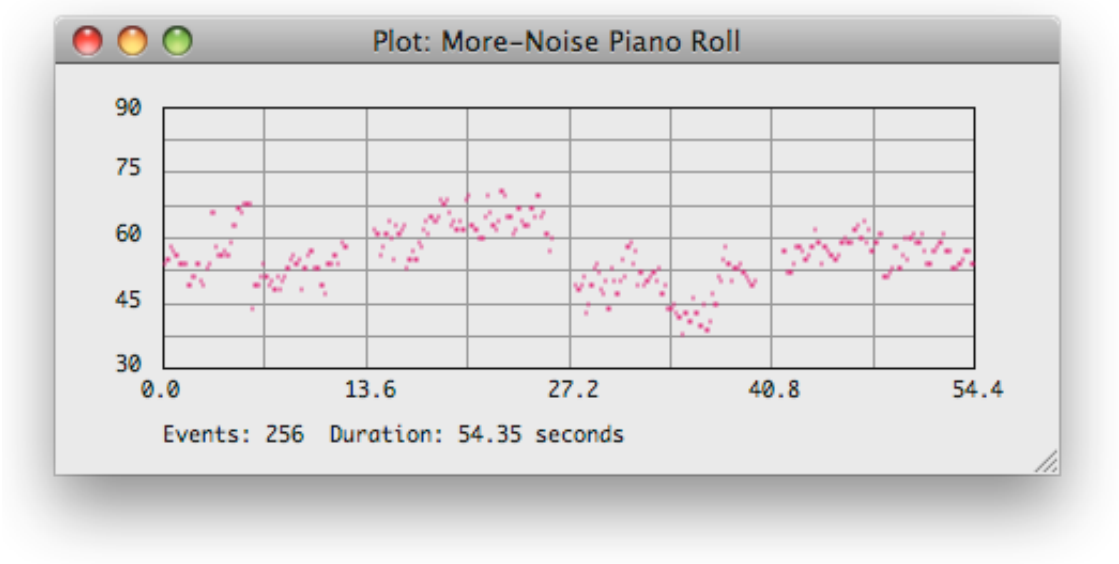
Many generators included in the Toolbox accept lists, generators, etc. as input for parameters. Each time the generator is applied, a new value for the parameter will be produced.

If however you want to use a generator to make *one* value for a parameter that will not change over time, you enclose the generator in *make*. The result is one constant value.

The low and high limits for *1/f-value* for pitch in section *noise2* are produced by making random-values in the following dialog box.

```
Name            noise2
Clock unit      150
Number          64
Rhythm          (1/f-value  64 0.5 2)
Pitch           (1/f-value  64 (make (random-value  24 48))
                               (make (random-value  60 80)))
Velocity        (1/f-value  64 30 90)
Channel         1
```

The lower limit for *1/f-value* will be a random value between 24 and 48. The same limit is used for the entire section. The upper limit will be a random value between 60 and 80. Each time the section is varied, a new lower and upper limit for *1/f-value* will be chosen.



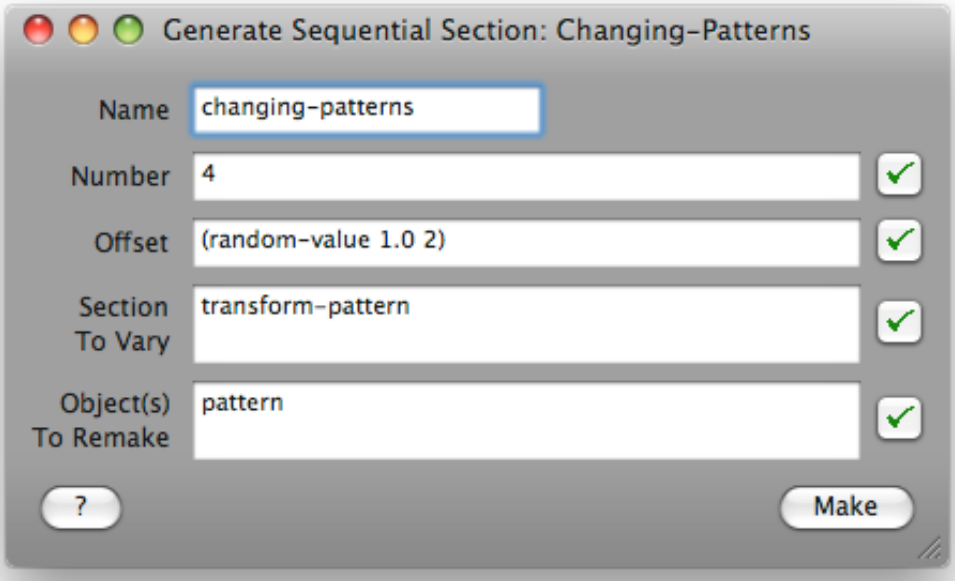
Remaking objects per variant

• Make the stockpile *pattern*. It will be transformed in section *transform-pattern*.

```
Name            transform-pattern
Clock unit      150
Number          90
Rhythm          (1/f-value  90 0.5 2.0)
Pitch           (transform-material  pattern
                          (series-choice '(0 1 4 5 7 -1 -4 -5 -7)))
Velocity        (1/f-value  90 30 90)
Channel         1
```

If this section is varied, the same pattern would always be used since the stockpile pattern has not been remade in the meantime. In order to remake one or more objects and then vary some other object, specify the object(s) to remake in the box labeled *Object(s) to Remake*. Each object in this box will be remade before a variant is made. Remaking the object means that the old output is lost and the input of the object will be used to create a new output.

Section *changing-patterns* remakes the stockpile *pattern* before each variant of *transform-pattern* is produced:



Making several variants with separate names
Several variants can be created and assigned user specified names with the tool *make-many-variants*. An expression with this tool can be evaluated in the Listener or in an editor window. The arguments are:

a list of names to be used for the new variants
an object to vary

The following expression will produce objects *noise3*, *noise4*, and *noise5* which are variants of *noise2*.

```
(make-many-variants '(noise3 noise4 noise5) noise2)
```

Editing an object
Sections, stockpiles, shapes, masks, and Midi objects can be edited. The original object can be replaced but the altered data can also be saved with a new name. In the latter case, this might be considered making a variant by hand.

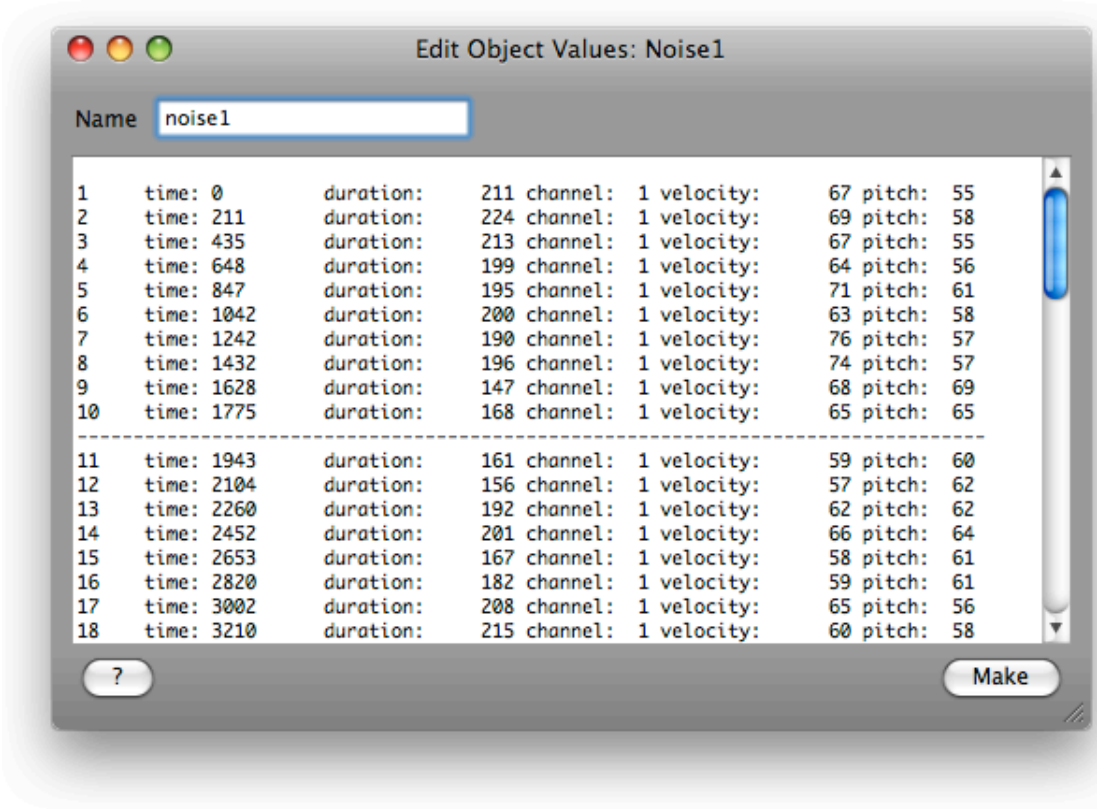
Shapes and masks are edited by drawing. The other objects are edited by changing text values.

To edit a section, select it in the Objects dialog and click on the *Edit* button. A window will open with the values for the section events printed in a fashion somewhat similar to a spreadsheet. Each line of text is an event. The first column is an index value, starting with 1. Time and duration are expressed in milliseconds. Channels are numbers starting with 1. Velocity and pitch are represented as Midi numbers.

Values for the parameters can be changed by replacing the current text value with another one. Pitch and velocity could also have one of the predefined input symbols such as *c4* or *pp* entered. To remove an event, the entire line should be erased. To add a new event, copy an existing line and change the values. It is important that the order of the columns be kept as is.

When all changes have been made, assign a name for the result and click on *Make*. If the same name as the original object is used, the events of that object are replaced. If a new name is given, a new object will be made and the original object is not changed. In both cases, the input description of the original object is used as the input description for the object.

If the original object has a comment, the comment can also be edited.



Midi objects are edited in a similar fashion.

Stockpile values are presented left to right, top to bottom. Values can be replaced or added.

The editor for an object can also be selected from the menu after right-clicking (CTRL-Click) the *Make* button.

Summary

Menu items
make variant, generate sequential section, generate parallel section

Generators
1/f-value

Tools
make, make-many-variants

Tutorial 7

Transform and other methods

Methods are functions that can transform several types of AC Toolbox objects including sections, shapes, masks, and stockpiles. All of these methods can be non-destructive, that is the object being operated on is not changed. A copy of the object is made and that is changed. If the new object has the same name as the old object, the old object is replaced with the transformed version.

Methods can be chosen via the **Methods** menu. The *make variant* method was introduced in Tutorial 6. The remaining methods are *transform, backwards, join, slice,* and *filter.*

Transform

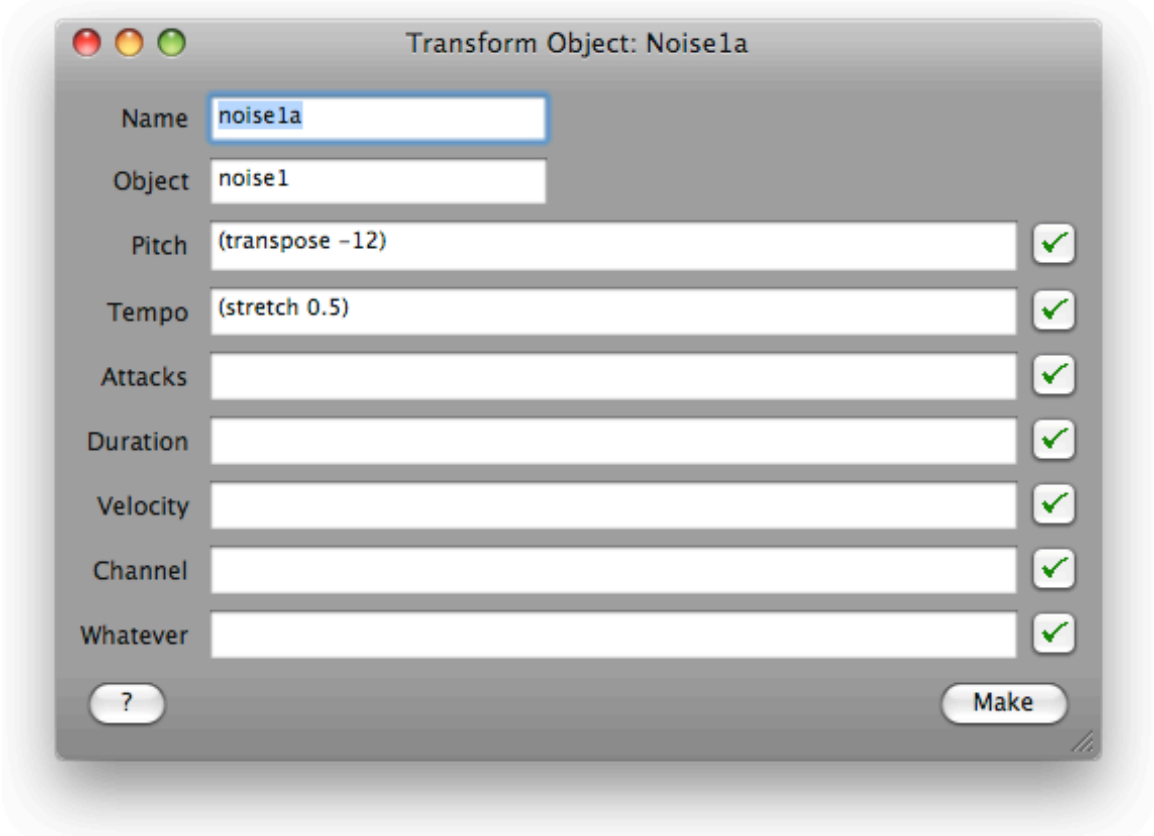
The ability to transform objects that have been previously defined is a powerful capability. Several transformers, such as *transpose*, *stretch*, *mirror*, *replace-if*, *random-deviation*, and *follow* are available and can be applied to several different classes of objects. The Transform dialog box includes separate boxes for many of the parameters for a section such as pitch, tempo, attacks, durations, velocity, and channel. Other objects, such as shapes, can be transformed by using the *whatever* parameter.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 7 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

• Make section *noise1* that has been defined in the *Tutorial 7 Examples* file. *Noise1* will be the basis for a few transformations.

```
Name           noise1
Clock unit     150
Number         64
Rhythm         (1/f-value 64 0.5 2)
Pitch          (1/f-value 64 c2 c5)
Velocity       (1/f-value 64 30 90)
Channel        1
```

Noise1a is a transformed version of *noise1*.



More than one parameter can be transformed at a time. In *noise1a*, pitch is transposed down an octave (12 semitones) and tempo is multiplied by 0.5. Section *noise1* is not affected by the transformations. The transformed object is *noise1a*.

If more than one transformer should be applied to a parameter, *compose* can be use to join the transformers together in a chain.

```
(compose (transpose -12) (compress 50 c3))
```

First the rightmost transformer is applied, then the next is applied to that result. In the above case, the value is first compressed and then transposed.

Help for transformers
A table listing available transformers is found by checking the *Transformers* box in the *Annotated Index.*

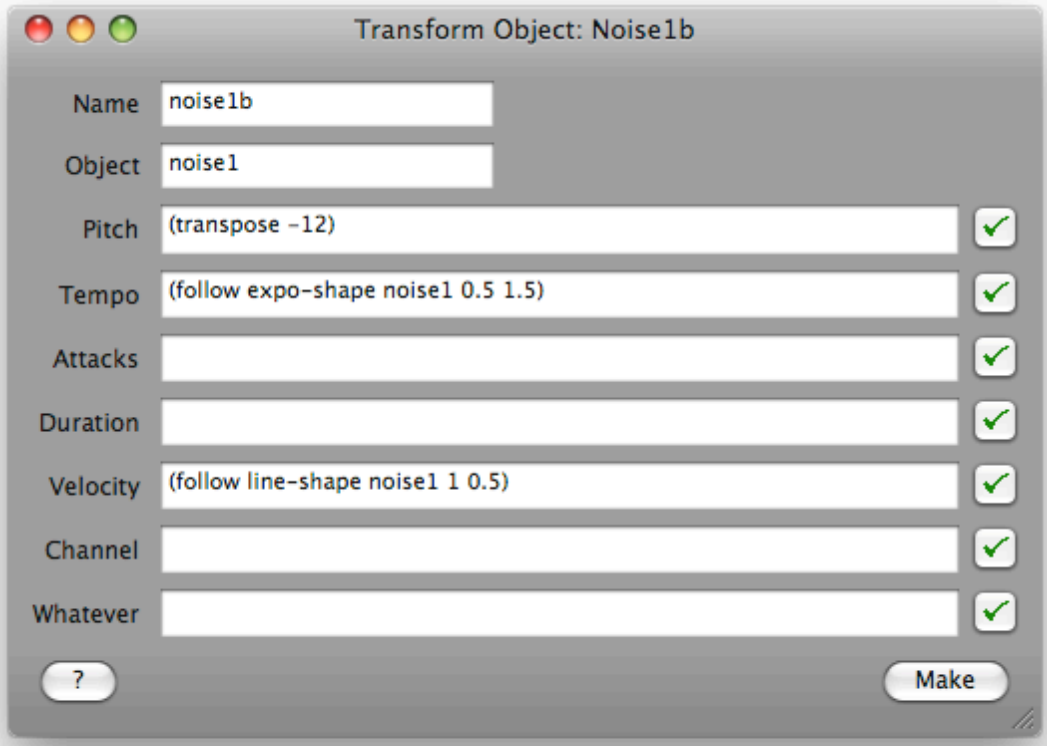
A text window containing information about a transformer appears when a transformer is selected. Many help windows for transformers contain statements such as:

```
EXAMPLE 1: (show-transformation (transpose 12)
                                '(60 62 64 65))
```

When the Lisp expression is evaluated, the transformer is applied to each value in the list that follows. The outcome can be seen.

```
Applying transformer
(transpose 12)
gives the following results:

Value:    60    Transformation:    72
Value:    62    Transformation:    74
Value:    64    Transformation:    76
Value:    65    Transformation:    77
```

Changing transformations over time
Follow is a transformer that multiplies parameter values. The multiplication factor will change over time following the curve of a shape. The shape is mapped to be between specified low and high values. The arguments for *follow* are shape, object, low, and high. *Object* can be the name of the object being transformed. The shape is divided into the same number of values as are present in that object.



Expo-shape and *line-shape* are included in the Toolbox as standard (default) shapes. Tempo will change in *noise1b* according to expo-shape (an exponentially increasing shape), varying from 0.5 times the tempo of *noise1* at the beginning to 1.5 times at the end. Velocity will change linearly from 1 (the velocity as found in *noise1*) to 0.5 of the original velocity values. For velocity the value *low* is higher than the value for *high*, therefore the effect of the shape is reversed.

Other transformers can also change over time. Consult the help windows for the transformers to see if they allow time-variant parameters.

One transformer that allows a parameter to change over time is transpose. In the following example the transposition interval is chosen using the generator random-value. Each time the transformer is applied, a new value for the amount of transposition is calculated with the generator random-value.

```
Applying transformer
(transpose (random-value -5 5))
gives the following results:

Value:    60    Transformation:    63
Value:    60    Transformation:    64
Value:    60    Transformation:    58
Value:    60    Transformation:    58
Value:    60    Transformation:    63
```

Transforming only durations
The dialog for *Transform Object* has an edit box for *Tempo* that controls attack times and durations. When tempo is transformed, both the attack times and durations change.

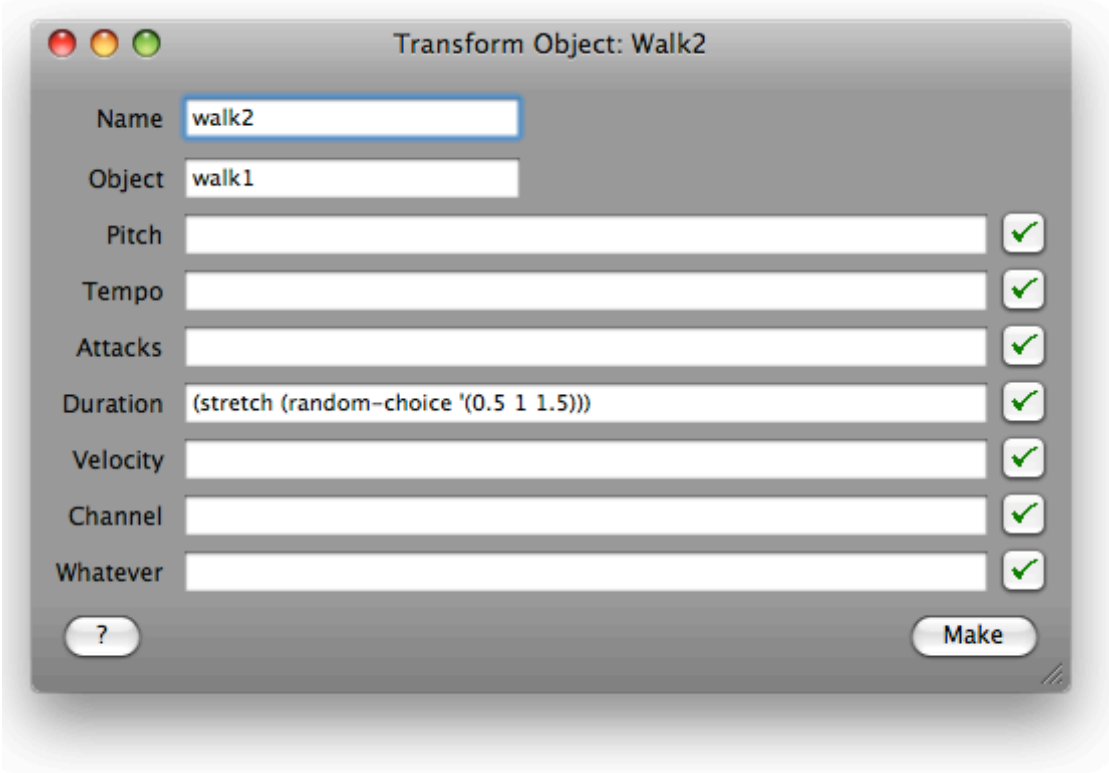
The dialog also has separate edit boxes for *Attacks* and *Duration*. These two parameters can be transformed separately, e.g. the attack times can be stretched while the duration values remain the same.

A form of rhythmic articulation can be obtained by adjusting the duration values without changing the attack times. Depending on the input, this could result in articulations similar to staccato or legato.

Object *walk1* will be transformed. This object uses random walks for rhythm and pitch.

```
Name            walk1
Clock unit      100
Number          100
Rhythm          (walk 2.0 (rv -0.1 0.1) 0.5 3)
Pitch           (walk c4 (rv -5 5) c3 c6)
Velocity        mf
Channel         1
```

Object *walk2* is the transformation of the durations from *walk1*. The durations are multiplied by 0.5, 1, or 1.5.



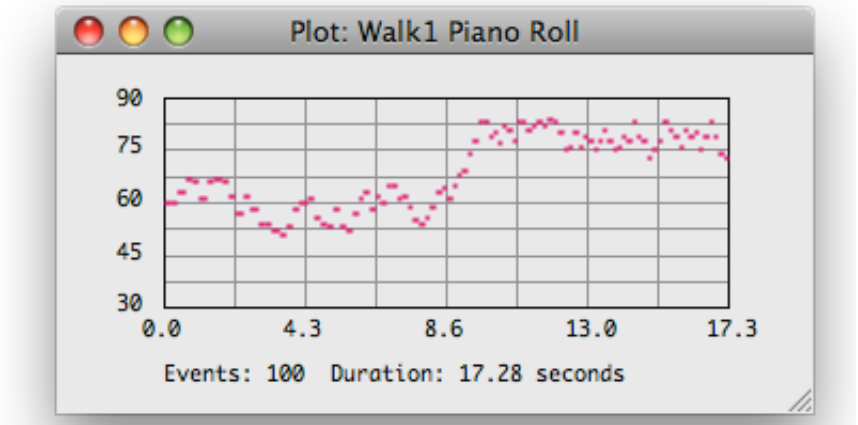
Transforming some of the values

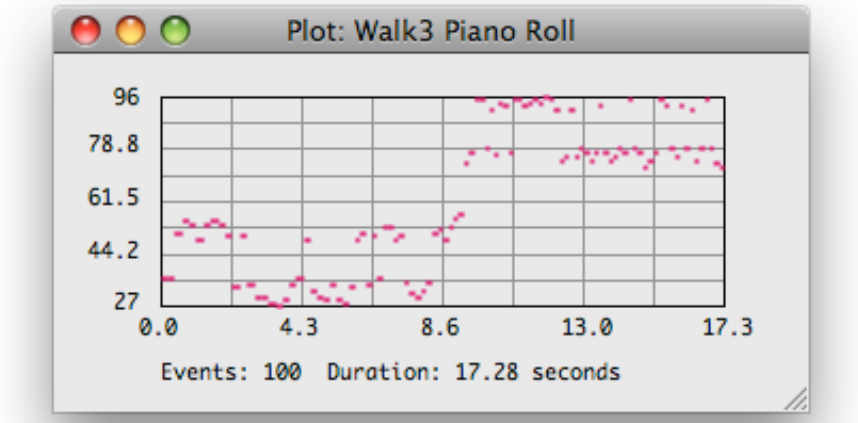
Transformer *transform-some* allows groups of values to be transformed, leaving other values untouched. The idea is that if a parameter value is within a specific range, it will be transformed. *Transform-some* operates on one parameter.

Object *walk3* transforms section *walk1*. The pitch transformation uses *transform-some*:

```
(transform-some transpose (c3 c4) -24 (c4 c5) -12 (80 90) 12)
```

In the above example, the transformer *transpose* will be applied to values within the specified ranges. Values between c3-c4, will be transposed with the argument –24 (down 2 octaves). Values in the range c4-c5, will be transposed down one octave (-12). Values in the range 80-90 will be transposed up an octave (12). Any number of these ranges can be specified. This transformation essentially stretches the contour of the random walk in *walk1*.





Transformations can also be limited to one or more Midi channels using *if-channel*. This is particularly useful if each channel is considered a separate instrument. Channel (instrument) 1 can be transformed without affecting channel (instrument) 2.

Object *walk4* uses random walks for rhythm and pitch. The notes alternate between two Midi channels:

```
Name            walk4
Clock unit      100
Number          100
Rhythm          (walk 2.0 (rv -0.1 0.1) 0.5 3)
Pitch           (walk c5 (rv -5 5) c4 c6)
Velocity        mf
Channel         '(1 2)
```

The pitches in channel 1 can be transposed without affecting those in channel 2. To do this, the *Whatever* parameter should be transformed. This transformer needs to know about both the pitch and channel data.

The input specification for *if-channel* is:

```
(IF-CHANNEL CHANNEL PARAMETER TRANSFORMER)
```

Channel is a channel number or a list of channel numbers to be transformed. *Parameter* can be one of the following:

pitch, velocity, tempo, duration, attack, controller-value, controller-number, controller-channel, program-number, program-channel.

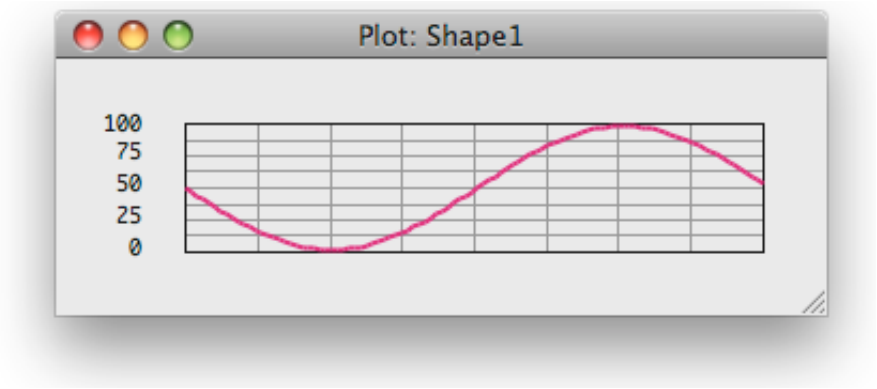
The last five of those parameters concern Midi objects that are discussed in Tutorial 11.

The effect of this transformation is clearer if the two channels are assigned to different Midi instruments. This can be done with **Other>General Midi**.

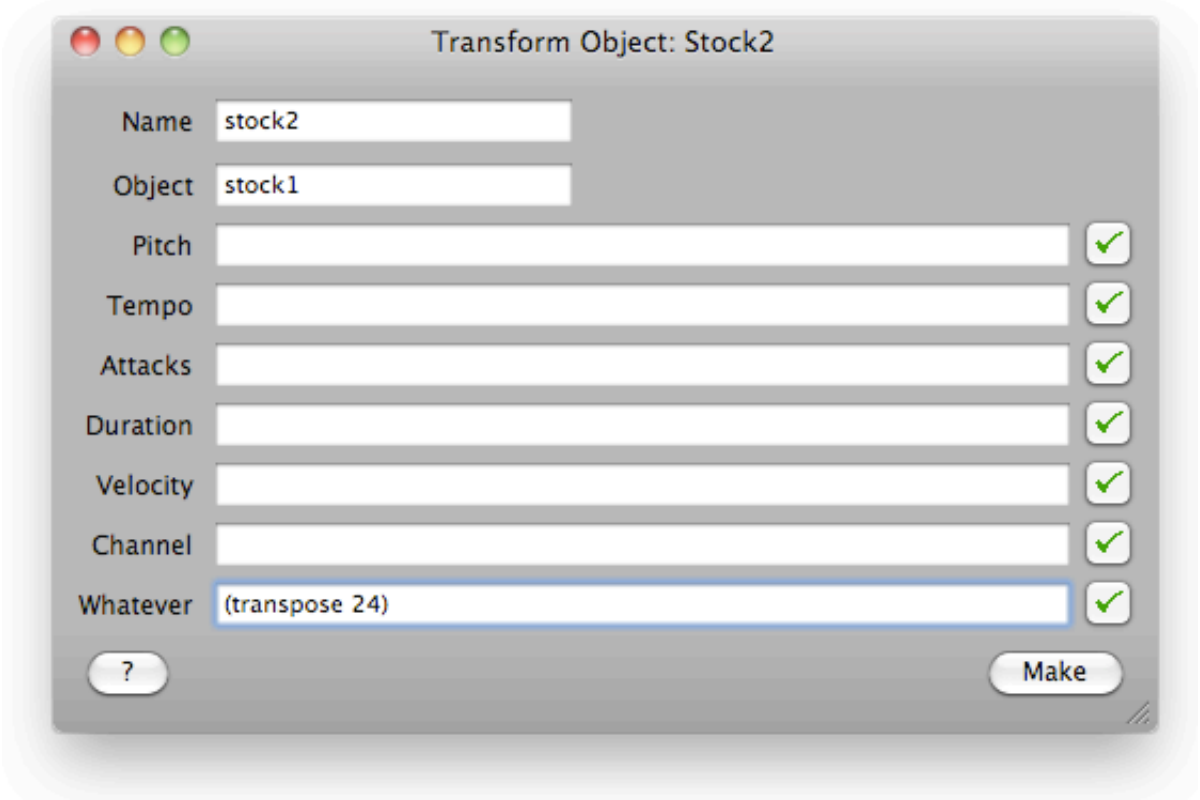
Additional information about transformers can be found in Tutorial 25. That tutorial includes examples using conditional transformers such as *transform-if, transform-and,* and *transform-or.* Simple transformers such as *translate*, *limit-range*, *quantize, insert-rest*, and *tran* are discussed. Tools and generators that change the order of values within an object are presented.

Transforming other objects
Objects other than sections can be transformed by specifying a transformer for the *whatever* parameter. A shape can be mirrored for example. Looking at the display of *sine-shape*, you can see that it is in the range of 0 - 100. This shape could be mirrored (top to bottom) entering *(mirror 50)* for the *whatever* parameter.

• Make *shape1* to make a mirror of the standard sine-shape.



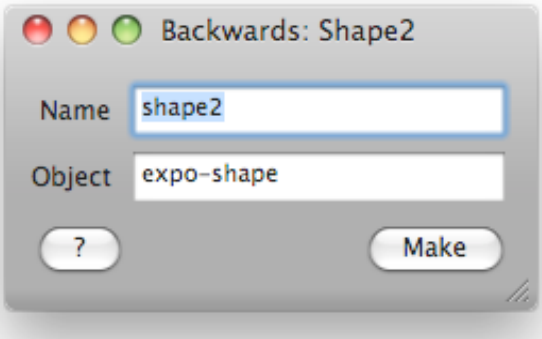
Stockpile *stock1* contains values from 36 to 48. *Stock2* is a transposed version of stock1.

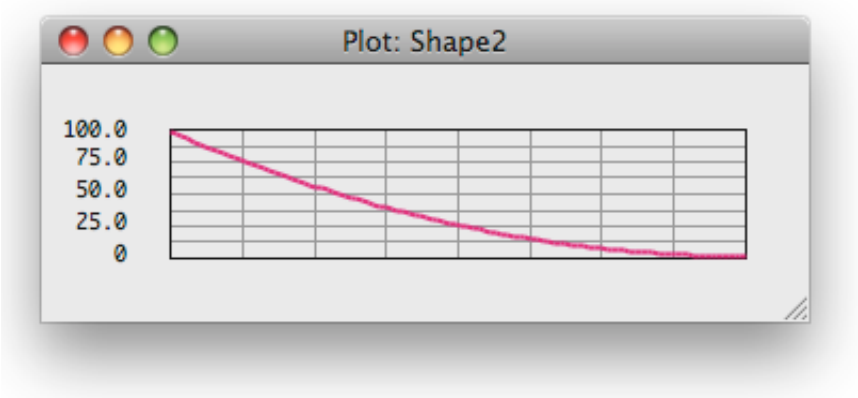


Transforming Midi objects requires an additional step. See Tutorial 11 for details.

Backwards

Backwards returns a copy of an object, but with the elements in reverse order. Backwards is a method that works on AC Toolbox objects such as shapes, masks, sections, stockpiles, and note structures.





Backwards is also available as a tool that can be used in an expression in a dialog box, e.g. when a section is being defined.

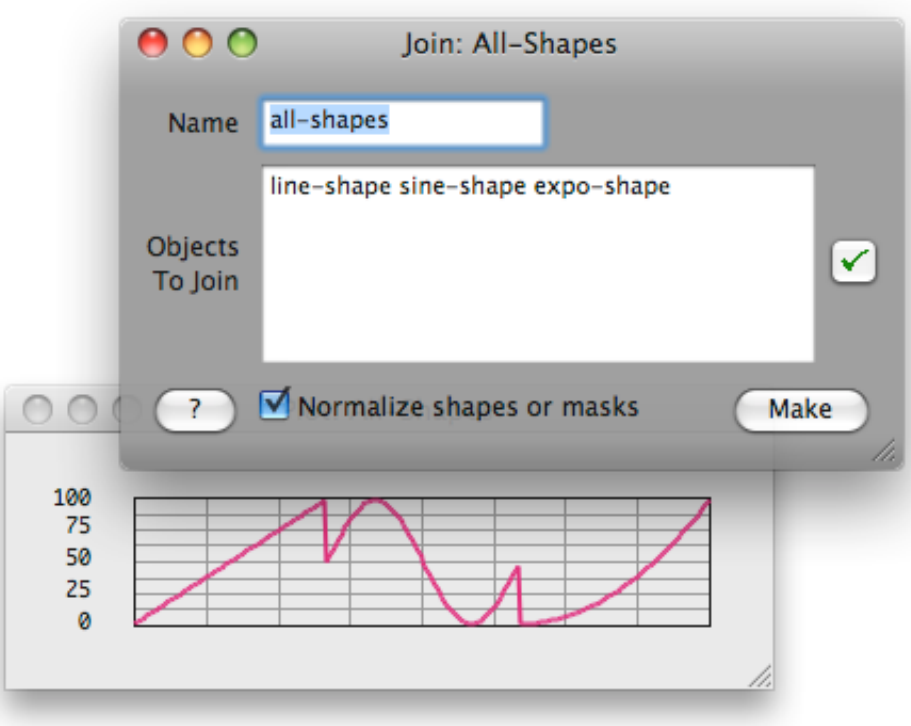
```
(convert (backwards expo-shape) 100 40 80)
```

When backwards is applied to a section, the notes will be in reverse order.

Join

Join is a method that joins (in sequence) two or more objects into a new object. For shapes and masks there is an option governing normalization. The default behavior is that shapes or masks are joined without any adjustment of their lengths. If the check box **Normalize shapes or masks** is checked, shapes and masks are first normalized to some standard length and then joined. Each shape and mask being joined will have the same length in the resulting new object.

Only objects of the same class can be joined.



Join is also available as a tool that can be used in an expression in a dialog box, e.g. to join two or more stockpiles together as input for a *–choice* generator.

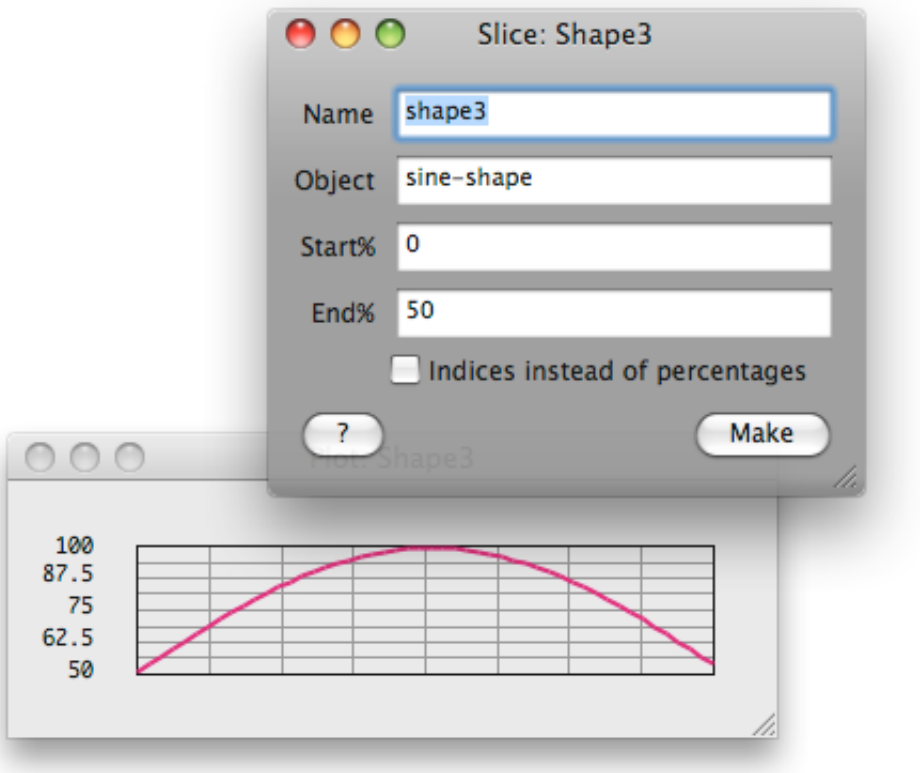
```
(random-choice (join stock1 stock2))
```

Join can also be used to concatenate transformers. The output of one transformer is the input to the next one. This works in the same way as the tool *compose*. The rightmost transformer is applied first.

```
(join (transpose -12) (compress 50 c3))
```

Slice

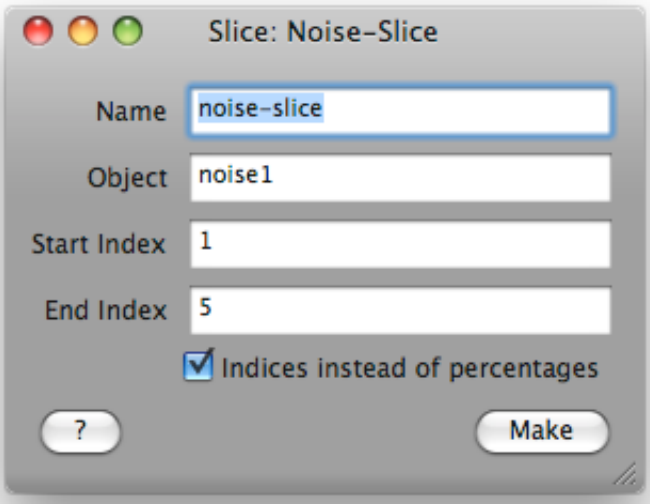
Slice copies part of an object and uses it as a new object. The part of the object to copy is indicated either in percentages or indices. The default behavior is to use percentages. From to 0 to 50 percent would give the first half of an object.



For most objects, the percentages refer to the number of values that are used to represent the object. For a section however, it refers to the duration of the section. The first half of a section is not necessarily half of the notes but half of the length of the section.

When the check box **Indices Instead of Percentages** is checked, the start and end values refer to index values. The first note or other value of an object has the index 1, the second the index 2, etc.

Object *noise-slice* is an example of slicing with indices. The first 5 notes of object *noise1* are copied to *noise-slice*.



Slice is also available as a tool that can be used in an expression in a dialog box, e.g. to use part of an object with a *–choice* generator.

To use the first 50% of a stockpile:

```
(random-choice (slice stock1 0 50))
```

To use the values in positions 8-13 from stock1:

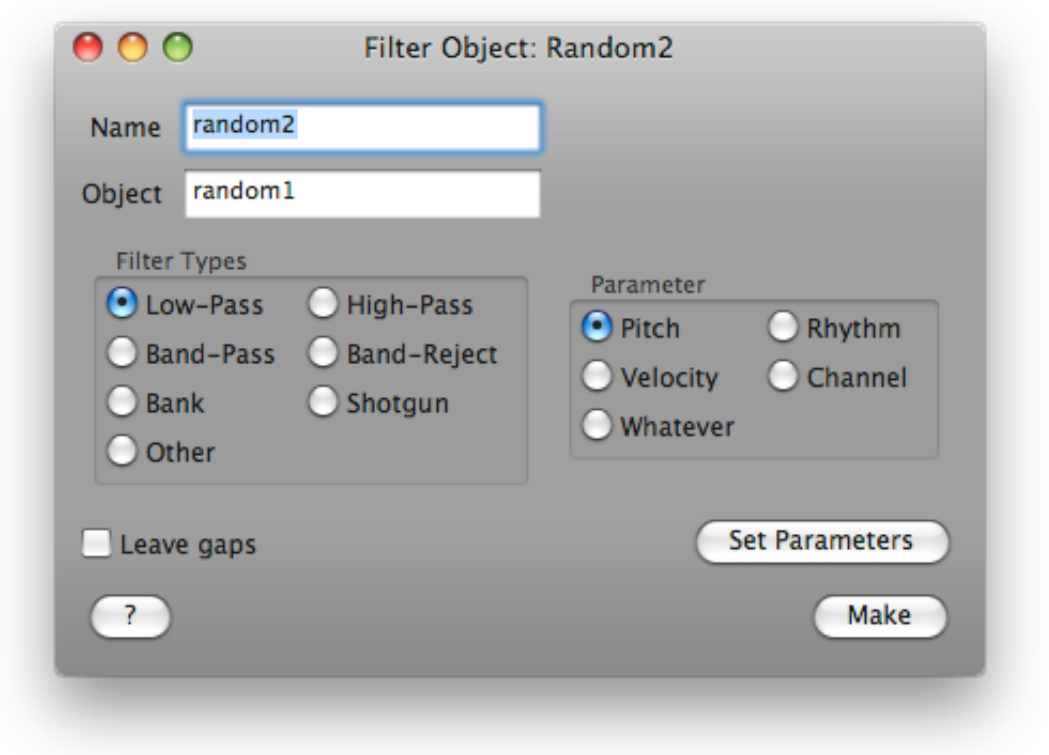
```
(random-choice (slice stock1 8 13 :by-number t))
```

Filter

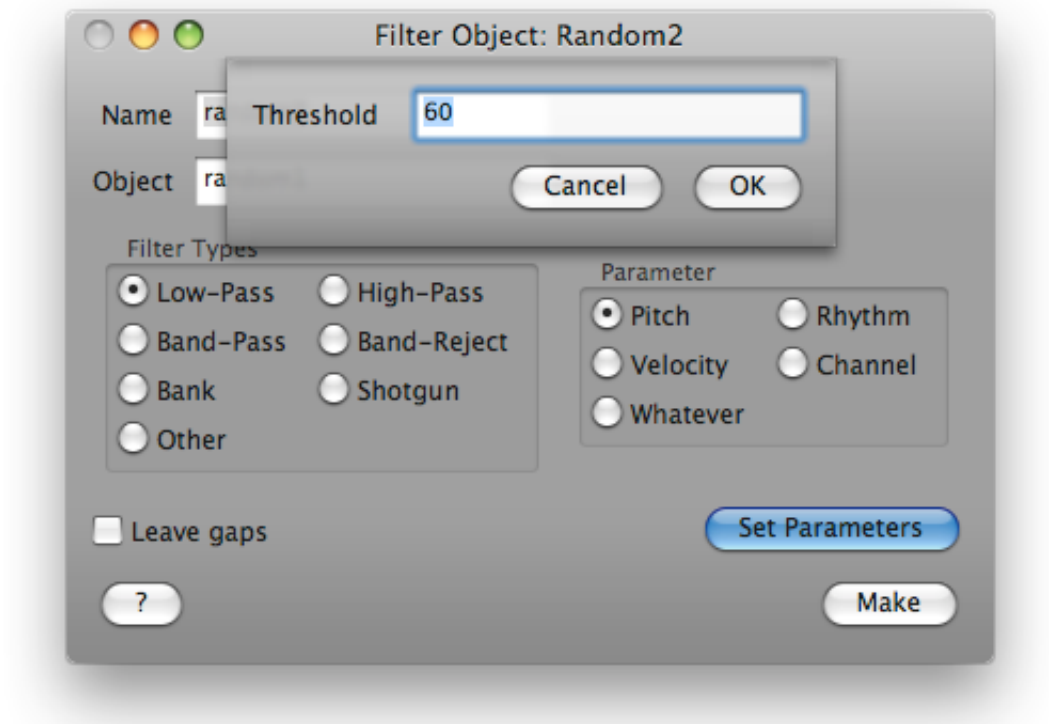
• Make section *random1*. It will be used as the basis for explaining the filter examples.

```
Name            random1
Clock unit      100
Number          100
Rhythm          1
Pitch           (random-value  30 90)
Velocity        mf
Channel         1
```

Various types of filters are available in the Toolbox. A user-defined filter can be entered as *Other*. As is the case with other methods, the filter method is non-destructive.



A *low-pass* filter passes all values less than or equal to some threshold. Choose *Set Parameters* to specify what that threshold should be.



Whatever is the parameter to use when the object being filtered is not a section or community. When an object is filtered, some values may be removed. In the case of a section, the starting times of the notes will be adjusted to make up for the hole that arises. If the hole is desired, check *Leave Gaps*. After all the data has been specified, *Make* must be clicked to filter the object.

The tutorial object *Random2* is an example of a low-pass filter with no gaps being left.

Random3 is an example of a high-pass filter with gaps being left. A high-pass filter passes values higher or equal to some threshold.

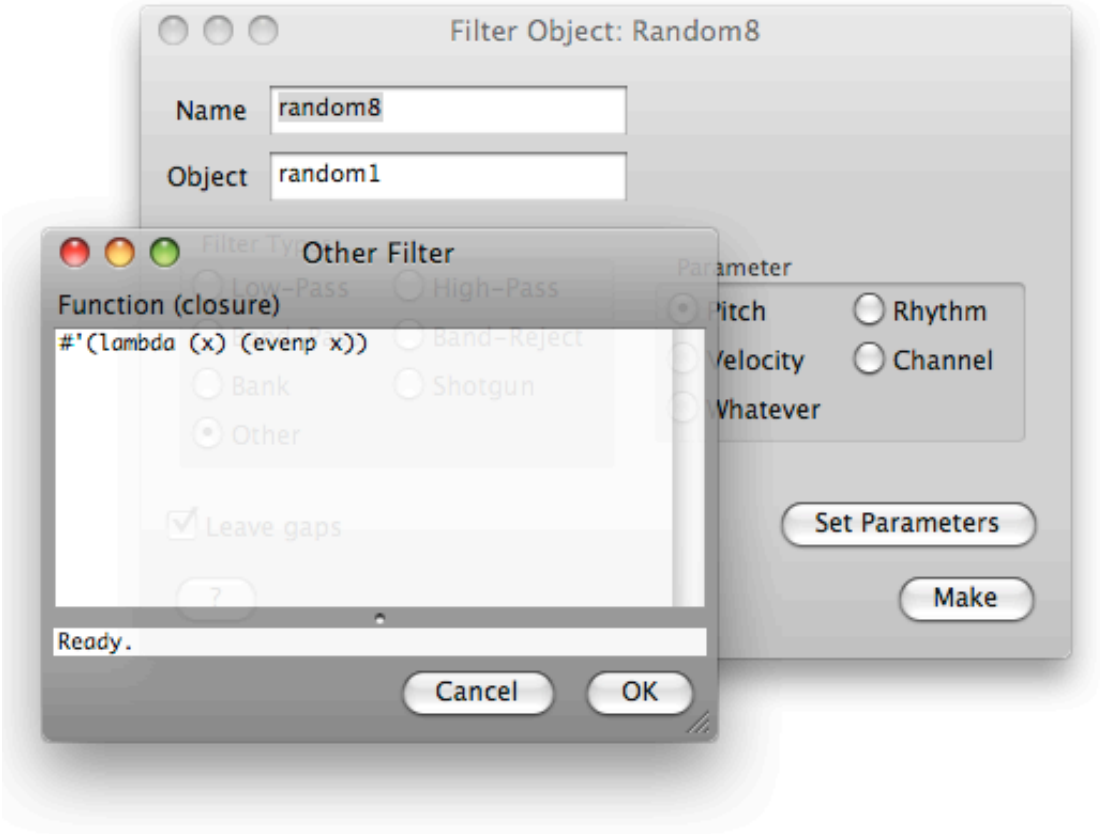
Random4 is an example of a band-pass filter. Values within a certain range (between two limits) are passed.

Random5 uses a band-reject filter. This filter passes all values except those within a certain range.

Random6 uses a filter bank. A bank is a group of filters. Several different ranges can be specified. Only values within one of those ranges will be passed.

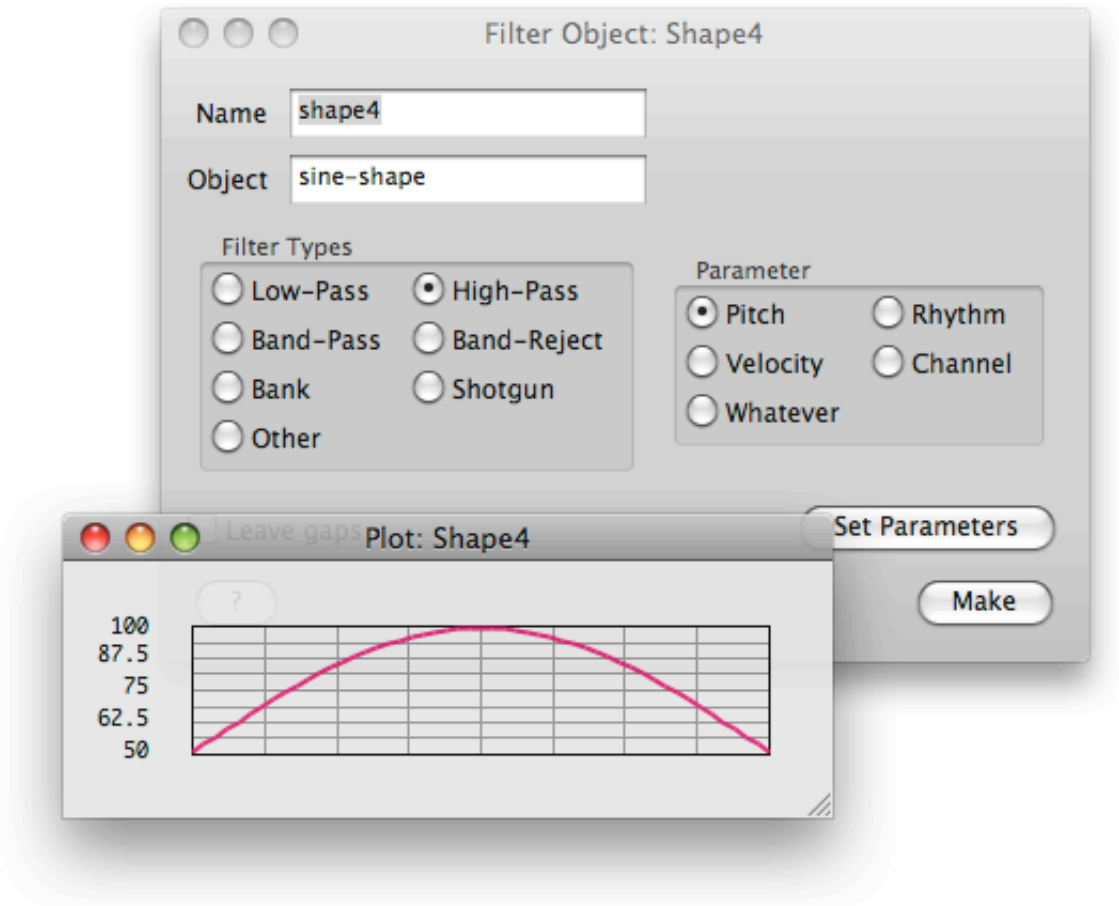
Random7 uses a *shotgun* filter. This is a nonstandard filter that filters out values at random. The percentage of values to be shot (filtered) must be specified.

Random8 is an example of an user-defined filter. The button for *Other* should be chosen and then a dialog box appears where a *lambda expression* or a function that produces one should be entered. If you do not understand what that is, do not worry. Just do not use this type of filter. The filter entered here passes even values.



Another example of an expression that could be entered for an *other* filter is *(note-pass-filter)*. This filters out anything that is not a note, e.g. Midi controller expressions or program changes get filtered. This is useful if Midi files are being imported into the Toolbox to be processed. Often only the notes should be processed and all other information should be filtered.

Filters can be used with other objects. Except for sections and communities, the *whatever* parameter should be chosen. *Shape4* is a filtered version of a sine shape. The threshold was set to 50.



Threshold parameters for filters may vary over time, i.e. a list, stockpile, or generator could be used instead of a constant value.

For information about filtering Midi objects, see Tutorial 11 or the item in the *Index* dialog called *Midi Objects: Filtering*.

Additional examples of using the filter method can be found in Tutorial 23. That tutorial discusses conditional filtering using *filter-if, filter-and,* and *filter-or*. Filtering with probabilities, filtering the results of generators, and filtering pitch intervals are also mentioned.

Summary

Menu items
backwards, filter, join, splice, transform

Tools
backwards, compose, join, show-transformation, slice

Transformers
follow, if-channel, mirror, transpose, stretch, transform-some

Tutorial 8

Filling time: density sections

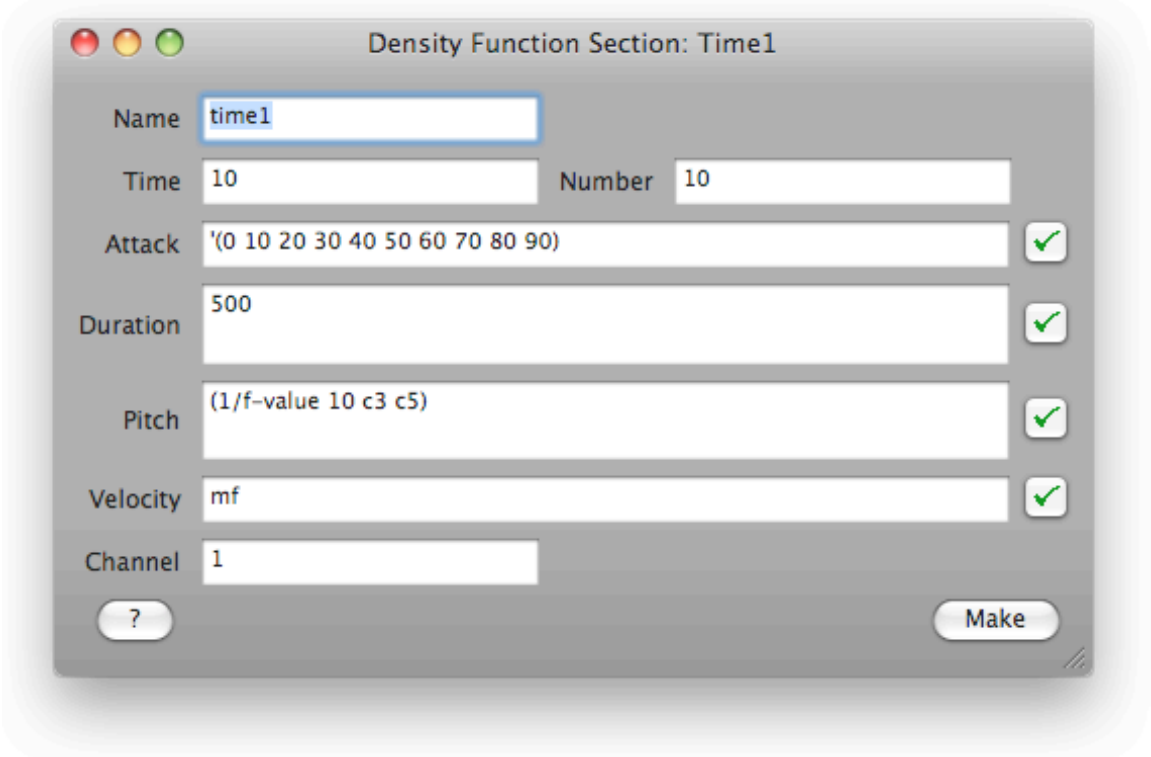
A *density section* in the AC Toolbox is a section where the time and density specifications are primary. The amount of time available, the number of events to be placed in that time, and a means of determining where those events will be placed are the essential ingredients of this specification. A density section can be specified with a function, a curve, or by mapping XY values.

Using a function
In a *density function section*, attack points are specified as a percentage of the total time available (0.0 - 100). Zero percent is the beginning of the available time, 100 percent the end. An attack point indicates where a note begins, not how long the duration is. Attack points should be a list or a generator that produces a value between 0 and 100. When using a generator for the attack points, attention should be paid to the difference between using 0 and 100 as data and using 0.0 and 100. In the former case, only integer values will be produced, limiting the attacks to 101 possible time values. In the latter case, real numbers will be produced, giving an infinity of possible attack points.

Duration is specified separately. It is a value in milliseconds (since there is no clock unit, duration is not a multiple of that). Duration can be specified in the customary Toolbox manner: constant, list, stockpile, generator, etc. It should be greater than 0.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 8 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Time in a density section is expressed in seconds. The attacks for *time1* occur within 10 seconds.



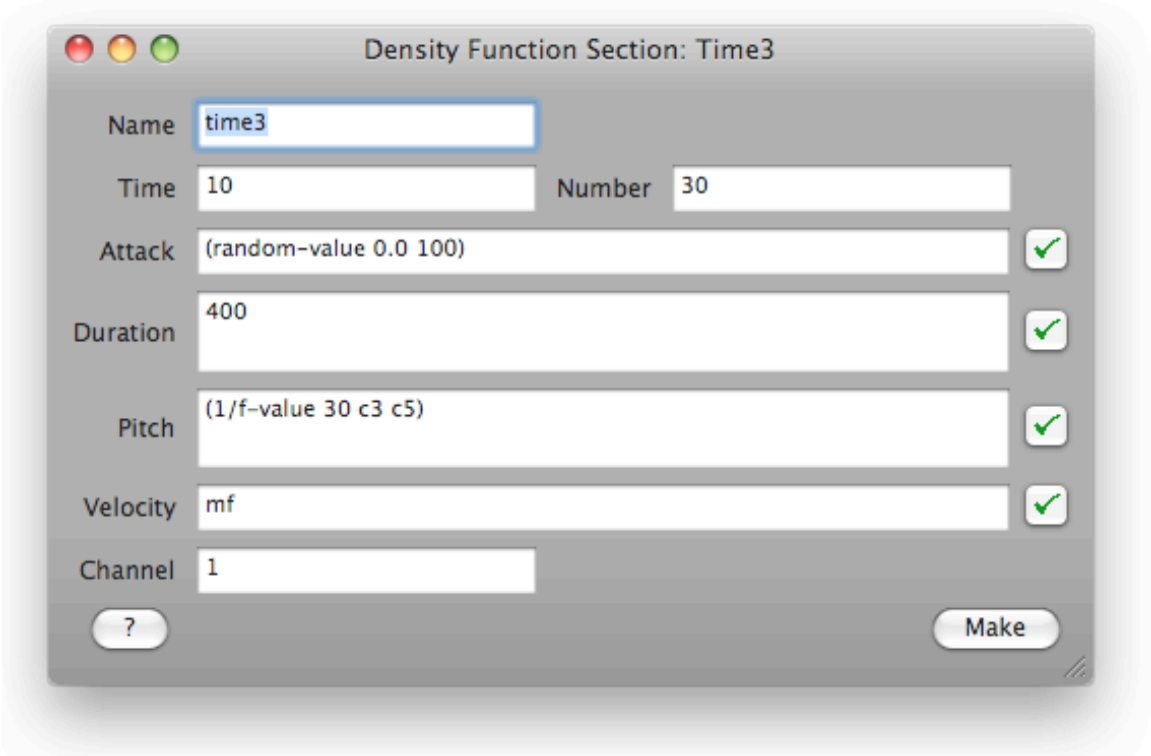
Note the list of values used to specify attacks. When a list of values is used, the number of values in the list determines the number in the section, regardless of the value entered for *Number* in the dialog box.

In the above section, the attack percentages *0 10 20 30 40 50 60 70 80 90* produced the following attack times (in milliseconds): *0 1000 2000 3000 4000 5000 6000 7000 8000 9000.*

• Make section *time2*. Note that the attacks points are more irregular:

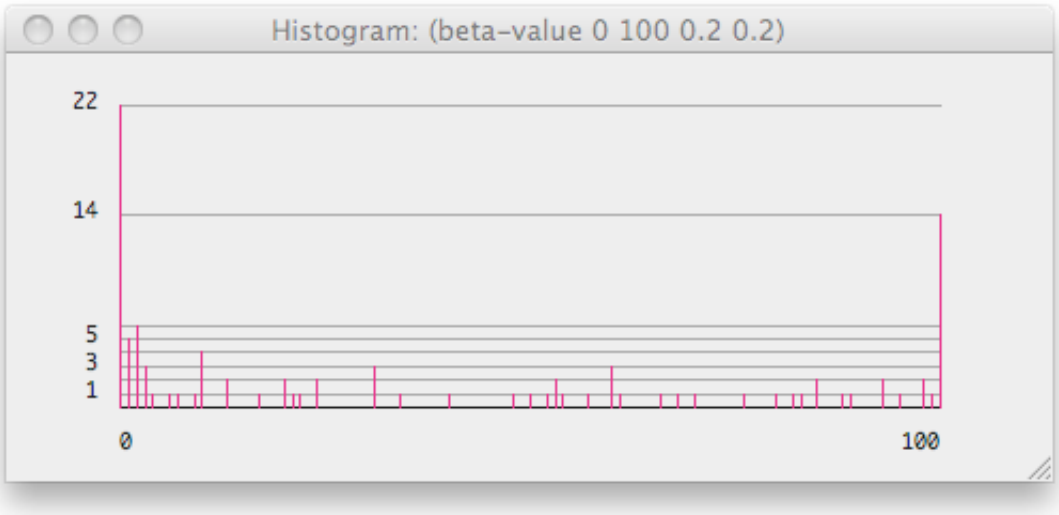
```
'(0 10 20 30 40 80 82 83 84 90)
```

Time3 uses a generator to produce the attack times. *Number* determines the number of attacks. The generator is applied that many times and the result is sorted numerically. The density curve of the section will be the same as the density curve of the generator.

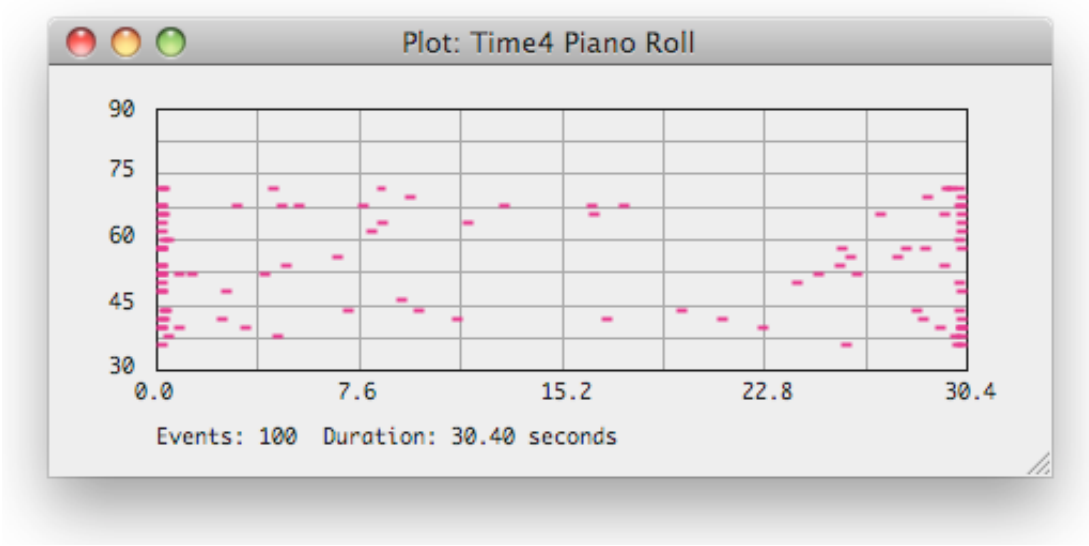


Time4 uses *beta-value* as the generator for attack points. Since this generator tends to produce values near its two extremes, attack points will be concentrated near the beginning and the end of the allotted amount of time.

A histogram for 100 values produced by *(beta-value 0 100 0.2 0.2)* shows this concentration near low and high values:

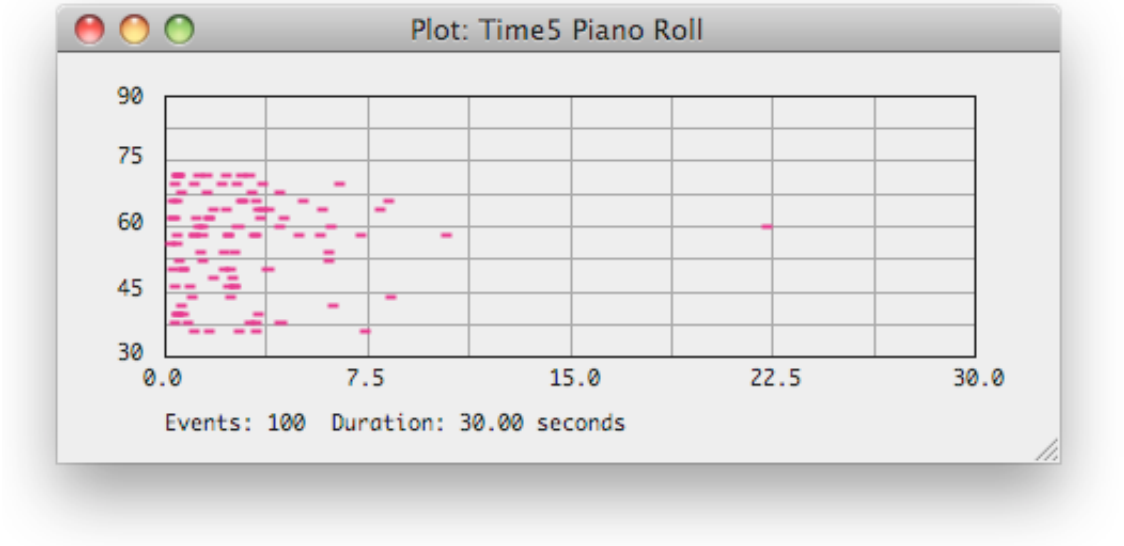


A similar distribution is found in the attack points for section *time4*:



Time5 uses *exponential-value* for the attack points. The density curve for this generator favors low values:

(exponential-value 0.0 0.1 100)



Time6 uses generators for both the time and the number of notes. This section is varied several times in *many-time6s*.

• Try several density function sections. Read the help window for *beta-value*. Vary the values for *a* and *b* in beta-value and notice the difference in the section.
• Read the help window for *exponential-value*. Use this generator for attacks in a density function section.

In the above examples, values were calculated with a function and then sorted into ascending order. The absolute starting position was important, not the distance between values.

Producing intervals
Another approach is to use a function to produce the intervals between the attack points. This is similar to what Xenakis did in his Stochastic Music Program where a probability function was used to decide the interval until the next attack.

Attacks is a tool that returns a list of attack points made by successively adding a new interval value to the previous result until some number of values have been made. This list of values is then mapped between *low* and *high*.

The arguments for *attacks* are *interval &key low high number*.

Interval can be a generator, stockpile, etc. The default value for *low* is 0.0. *High* will default to 100. If *number* is not specified, the number in the dialog specification *(from-number)* will be used. These default values make this tool convenient to specify a density section.

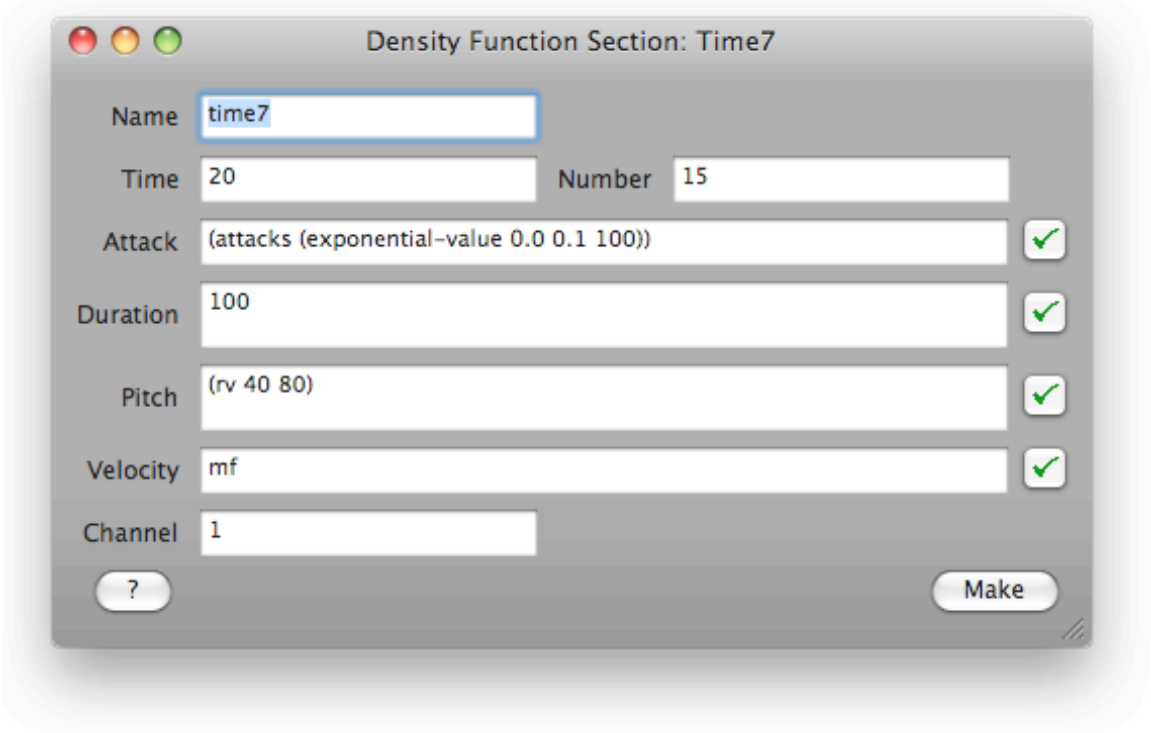
(attacks 1 :number 10) produces the following evenly spaced values between 0 and 100:

```
  0.000    11.111    22.222    33.333    44.444
 55.556    66.667    77.778    88.889   100.000
```

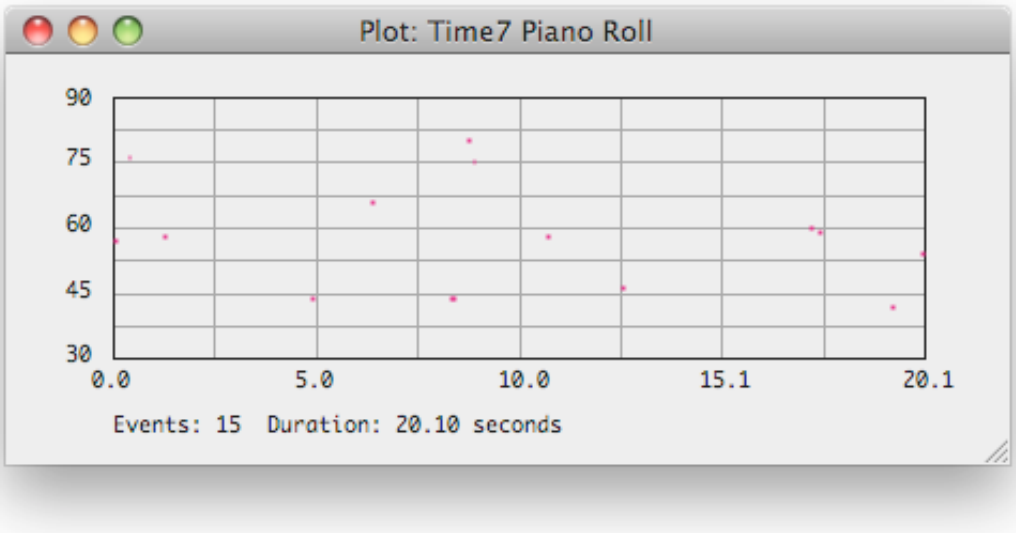
(attacks (exponential-value 0.0 .1) :number 10) could produce:

```
  0.000     9.560    20.759    24.784    62.276
 67.107    84.635    92.458    96.261   100.000
```

Time7 uses *attacks* to produce the attack times for the events in a density function section.

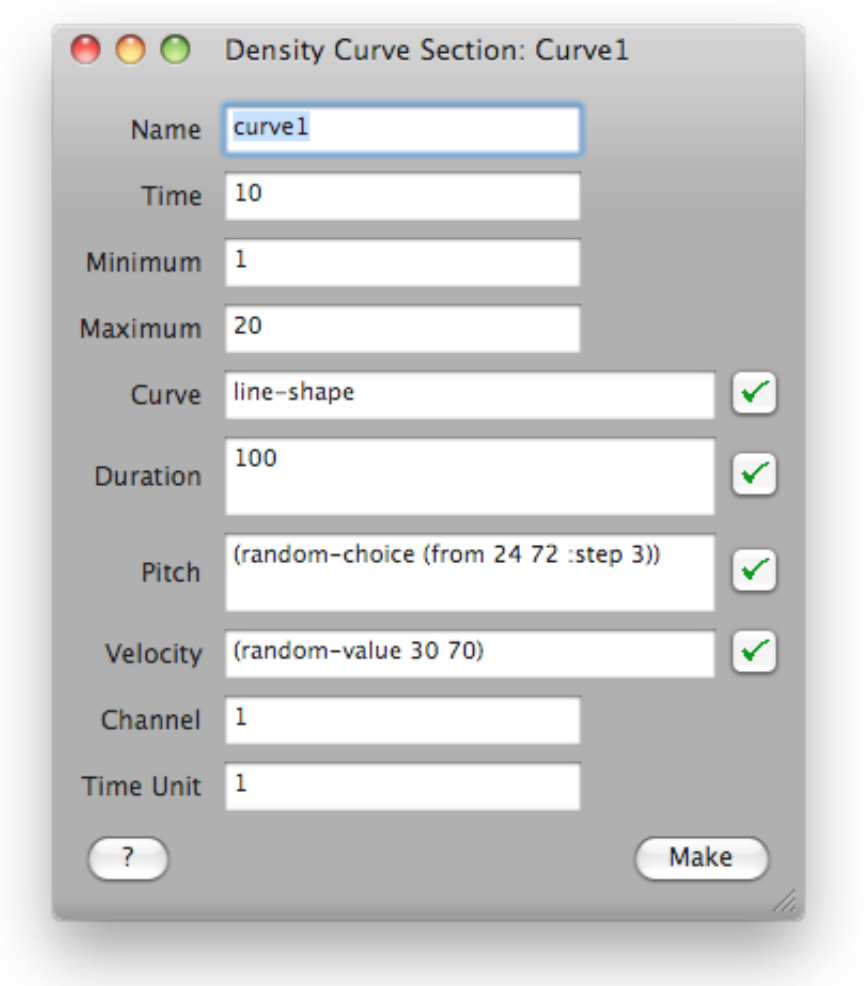


The plot of the note values shows the way that the attacks times are spread throughout the section. The interval generator controls the relative distance between the attack times that are always in ascending order.

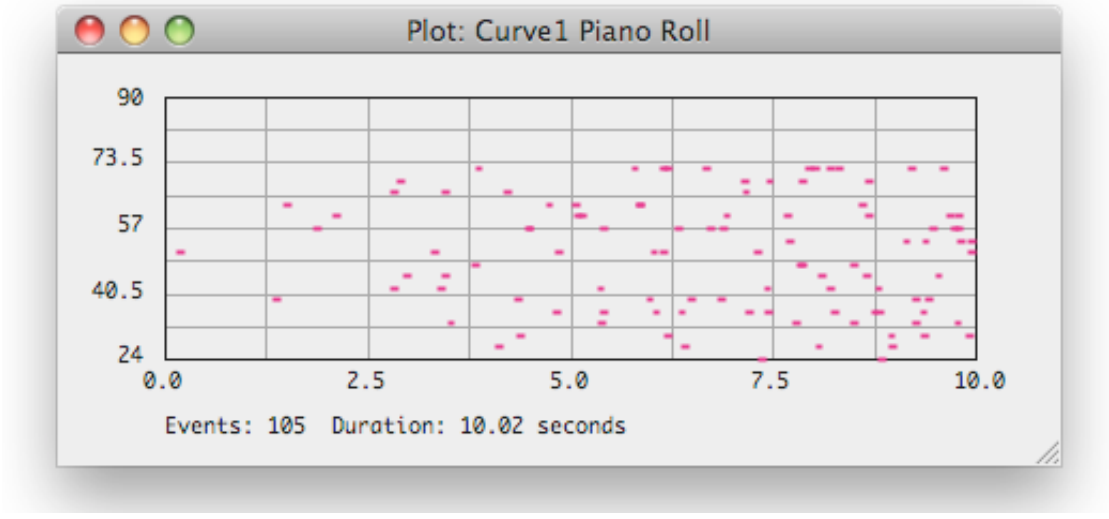


Using a curve
In a *density curve section*, the density of notes within an allotted amount of time is determined by mapping a curve (shape or mask) between a specified minimum and maximum number of notes per time unit. By default, the time unit is 1 second.

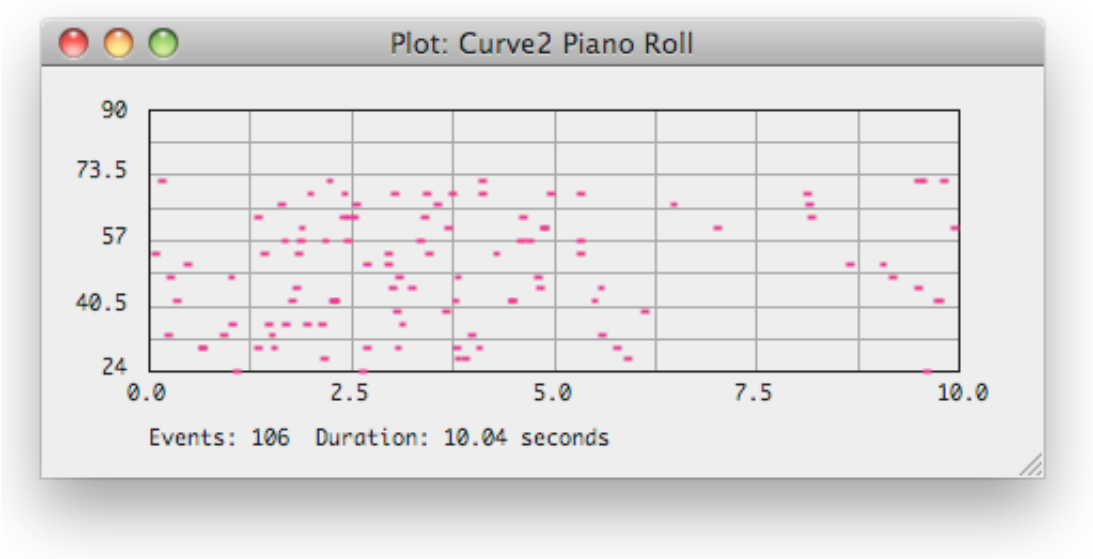
Time is expressed in seconds. Durations are expressed in milliseconds.



The density of events in section *curve1* increases linearly from 1 per second to 20 per second during the time of the section.



Using a different shape obviously produces a different pattern of density changes. *Curve2* uses *sine-shape*.

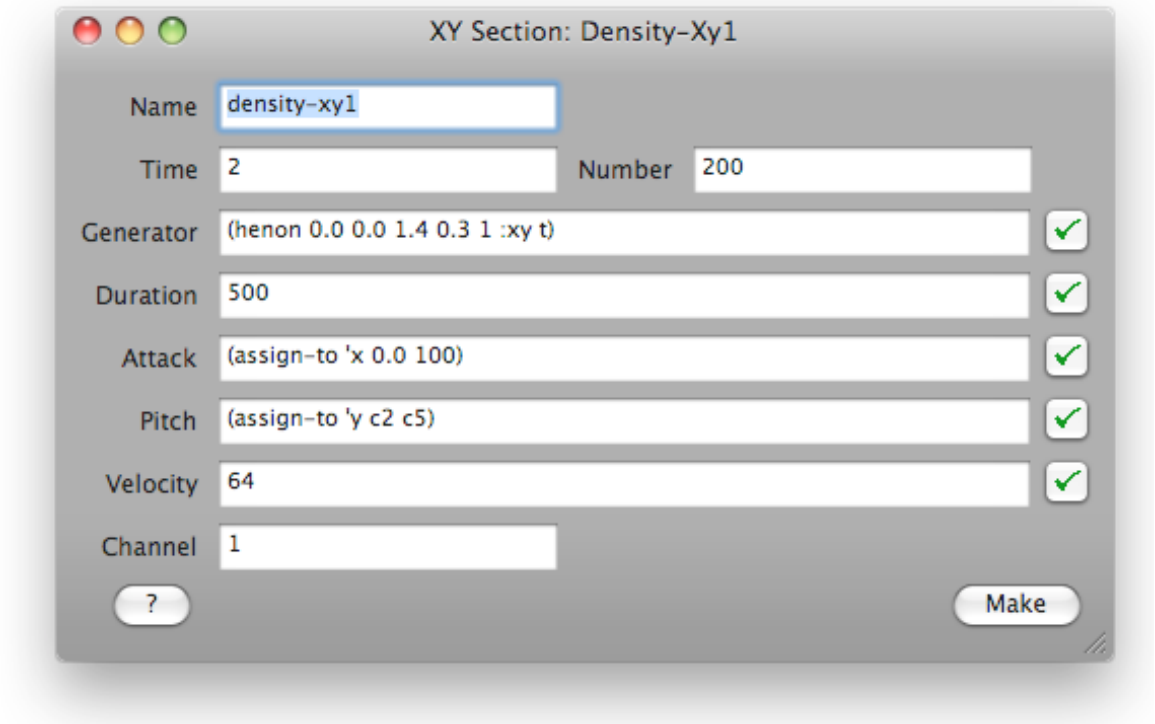


Curve3 uses generators for *time*, *min*, and *max.* These generators are applied once when the section is made. *Many-curve3s* produces a section with several variants of this short section.

If a mask is specified for the curve, it is converted to produce random values between its lower and upper limit for the number of notes per time unit.

• Make several density curve sections. Draw some shapes to control the density.

Mapping XY values
In a *XY section,* a generator that produces xy values or xyz values, such as some of the generators found in the chaos category, are used to determine the attack times and the pitches of a section. It is important that the generator produce xy or xyz values. These can be done by binding the xy or xyz keyword to *t*, *1*, or any value other than *nil*.

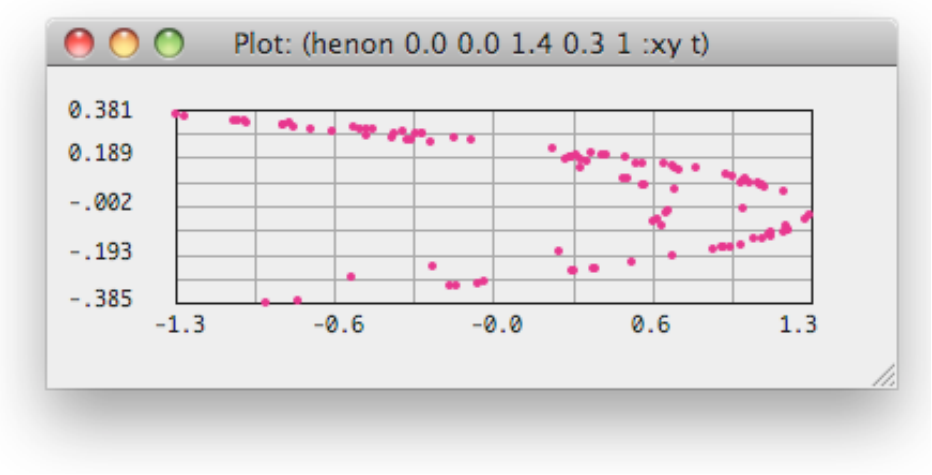


In object *density-xy1,* a henon generator is used. Note that the keyword xy is bound to *t*. The *x* value produced by the generator is used to calculate the attack times of notes. This

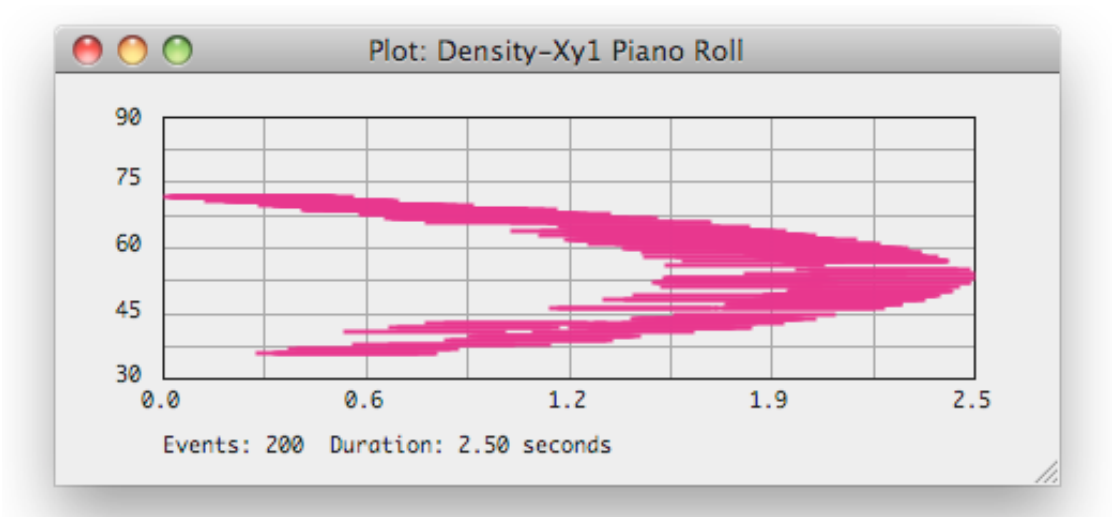
is done by entering the form *(assign-to 'x 0.0 100)* in the attack box. The values 0.0 and 100 represent percentage values of the total time, which in this case is two seconds. The *y* parameter is assigned to pitch with the form *(assign-to 'y c2 c5).* The y value produced by the generator is mapped to be between c2 and c5.

Duration is expressed in milliseconds and has nothing to do with the generator. Duration and velocity in this example are constants but they could also be a generator, list, stockpile, etc. Time is in seconds.

An xy-section essentially maps the phase-space of the generator into the pitch-time space of a section. Note the similarities between the plot of the henon phase-space below with the piano roll representation of density-xy1.



Henon phase space



If a generator produces xyz values, the z parameter can also be mapped using the *assign-to* syntax, e.g.: *(assign-to 'z 30 100)*. Section *density-xyz1* is an example of using x, y, and z values in an xy section.

Summary

Menu items
using a function , using a curve, mapping xy values (density section)

Tools
attacks

Tutorial 9

Representing 'real' music: note structures

In addition to importing and exporting notes via Midi files, notes can be entered into the Toolbox environment by using *note structures*.

A note structure is a note, a rest, a delay, or some combination of these structures. Note structures can be combined in sequence or in parallel. A note structure could also contain a section. This representation is similar to schemes suggested by Peter Desain and Roger Dannenberg.

A-note
A *note* is specified with *(a-note rhythm pitch velocity channel)*.
Rhythm is a numerical value (integer or real) specifying a relative length for a note.
Pitch is a Midi note number (0-127), a floating point number in the range 0-127, or a Toolbox pitch symbol such as *c3*. Pitch can also have a list of note numbers to occur at the same time (a chord).
Velocity is a Midi value (1-127) or a Toolbox velocity symbol such as *mf*.
Channel is a Midi channel number (1-16).

An example of a note is *(a-note 2 c4 mf 1)*. A chord example is *(a-note 1 '(60 64 67) mf 1)*.

A-rest
A *rest* is specified with *(a-rest rhythm)*.
Rhythm is a numerical value specifying a relative length for a note, for example, *(a-rest 1)*.

The actual length of a note will depend on the clock unit used when a section is made with a note structure.

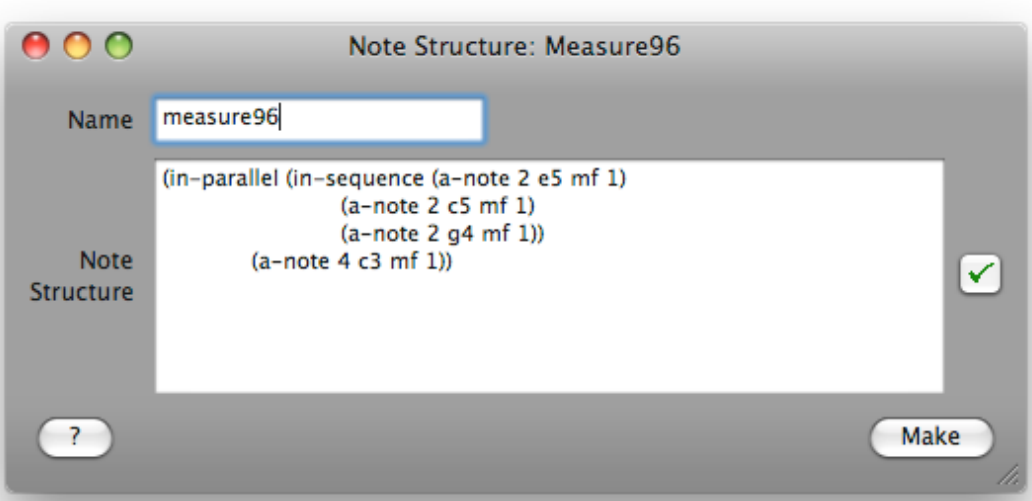
A-delay
A *delay* is an actual amount of time for a rest. It is not relative to any other value. It is expressed in milliseconds, for example, *(a-delay 100)*.

In-sequence
Note structures can be joined in sequence. They will be interpreted to occur one after another (in sequence). An indefinite number of structures or their combinations can be included in sequence.
For example,

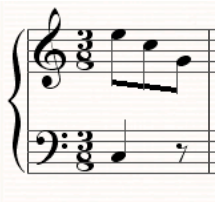
```
(in-sequence
  (a-note  2 e5 mf 1)
  (a-note  2 c5 mf 1)
  (a-note  2 g4 mf 1))
```

In-parallel
Note structures can be joined in parallel. They will be interpreted as occurring at the same time. Any number of structures or their combinations can be included in parallel.

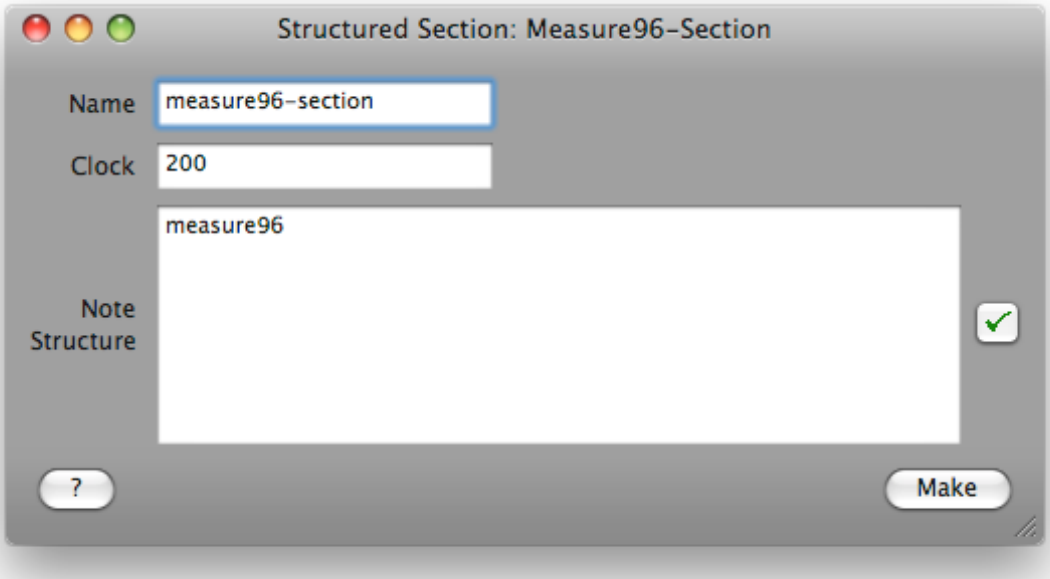
The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 9 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.



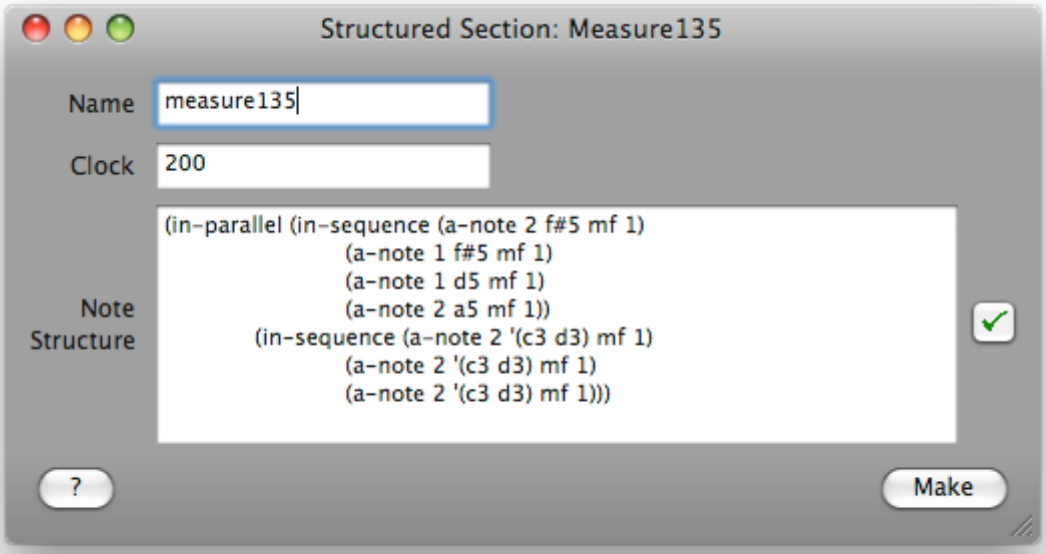
The note structure defined above is one measure from Mozart's *Musical Dice Game*. A common music notation representation of this note structure is:



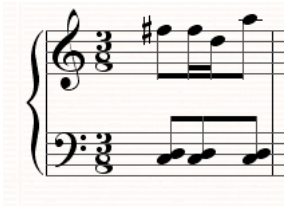
Structured section
A section can be defined using note structures. A clock unit is given. All rhythmic values are considered to be multiples of this clock unit. The name of a note structure



or a specification for a note structure is entered.

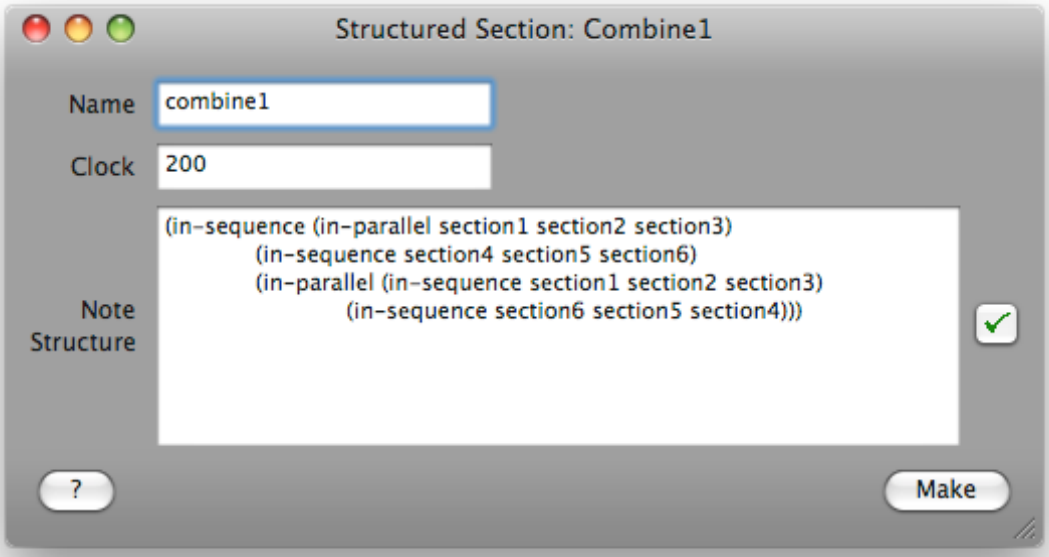


Section *measure135* represents measure 135 from the *Musical Dice Game*. A common music notation representation of this section is:



• Enter some note structures based on a score and use them in a structured section.

Using sections in note structures
In-sequence and *In-parallel* can be used with sections. In addition to joining notes and rests, sections and combinations of sections can be joined in this way. If six sections with the names *section1*, *section2*, ... *section6* existed, they could be joined as in the following note structure:



Remember that the sections would first have to be made. Section *combine1* would combine sections 1 - 3 in parallel, then sections 4 - 6 would occur in sequence, and the sequences sections 1-3 and sections 6-4 would occur in parallel.

When a section is present in the specification of a structured section, the clock unit of the structured section has no effect on the section being used.

Summary

Menu items
note structure (define), structured section

Tools
a-note, a-rest, a-delay, in-sequence, in-parallel

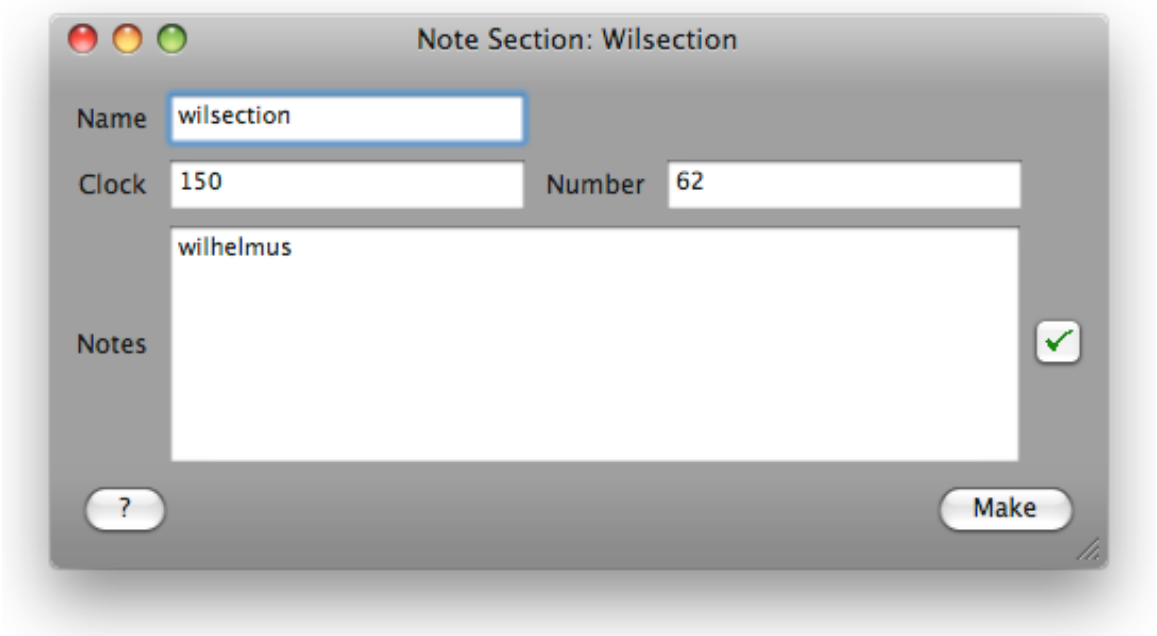
Tutorial 10

Making note sections

With note structures
In a *note section*, the parameters rhythm, pitch, velocity, and channel are dealt with together by using note structures (see Tutorial 9). This contrasts with data sections, where these parameters are generated independently of each other.

A note section should have a note structure (a-note, a-rest, in-sequence, etc.) as input or a generator that produces a note structure or a stockpile that contains one. *Wilhelmus* is the name of a note structure that is included in the AC Toolbox environment.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 10 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.



Clock unit is expressed in milliseconds. It can also be calculated with the tool *bpm*. *(bpm 160)* translates 160 beats per minute to the corresponding value in milliseconds. *Mm* will convert a metronome indication to a value in milliseconds, e.g. *(mm 120)*. The rhythmic values for each note are multiplied by this clock unit to determine the duration of the note. *Number* is the number of notes included in the section. To find out how many notes are included in a Toolbox object such as a note structure, apply *get-length* to the object. This can be done in the Listener (**Other>Listener**).

```
CL-USER 1 > (get-length wilhelmus)
62
```

The expression *(get-length wilhelmus)* can also be entered in the note section dialog as the value for *Number*.

With generators
Note structures can be used as stockpiles for the *choice* generators, such as *series-choice*, *random-choice*, etc. Notes will be chosen from the stockpile according to the algorithm of the generator. Note section *wilsection2* uses *series-choice* to select notes.

```
Name            wilsection2
Clock unit      150
Number          62
Notes           (series-choice wilhelmus)
```

Series-choice picks values at random but does not repeat a value until all have been used once. Since the note structure *wilhelmus* contains 62 notes, each note will only occur once in section *wilsection2.*

When a note structure is used as a stockpile for a generator, the note structure is flattened. No notes are in parallel, all are in sequence.

Wilsection3 uses a different generator for producing the notes: *beta-choice.* This generator will tend to pick the notes at the beginning and end of the stockpile.

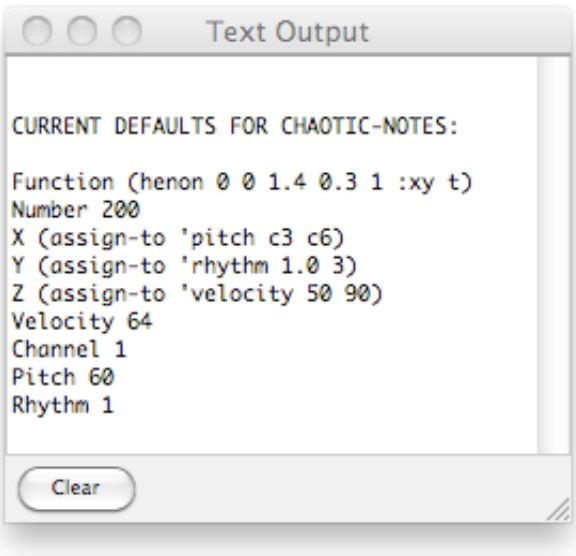
```
Name            wilsection3
Clock unit      150
Number          100
Notes           (beta-choice wilhelmus 0.3 0.3)
```

Instead of using a note structure as a stockpile for the *choice* generators, the name of a previously defined section may also be used. In this case, the notes will be extracted from that section, flattened, and used as a stockpile. For example, *(beta-choice wilsection 0.3 0.3)*. This is particularly useful if the section has been read in from a Midi file and the intention is to manipulate the notes that are contained in that file.

With tools

Chaotic-notes is a tool that produces a list of note objects by mapping the output of a chaotic function. This list is suitable as input for a note section. By default, the x value is mapped to pitch and the y value to rhythm. Keywords can be used to modify the various default assignments. The current default arguments will be printed in a Text Output window if the keyword *:defaults* is bound to t:

(chaotic-notes :defaults t).



The form *(chaotic-notes)* produces a list of 200 note objects using default parameters, including a default henon generator.

```
(chaotic-notes :function (lorenz 0.1 0.1 0.1 10 28 3/8 1 :xyz t))
```

The above expression uses the keyword *:function* to specify a lorenz generator for producing the note objects.

Chaotic-section maps the above lorenz function. The default assignments of x to pitch, y to rhythm, and z to velocity are used.

```
Name            chaotic-section
Clock unit      100
Number          200
Notes           (chaotic-notes
                    :function (lorenz 0.1 0.1 0.1 10 28     3/8 1
                   :xyz t)
                   :plot t)
```

With shapes
A stockpile of notes, either from a note structure or from a previously defined section, can be read using *read-from.* The order of the notes chosen can follow a shape. A line-shape would simply go from the beginning to the end of the stockpile. Another shape would read in a different order. The use of shapes to read from stockpiles is discussed in Tutorial 3. New in this tutorial is that *read-from* is reading note structures and is being used to produce note sections.

When a shape is used to read-from a note structure, the lowest value in the shape refers to the first note in the (flattened) note structure and the highest value refers to the last note in the note structure.

Wilsection4 uses a sine-shape to read-from the note structure.

```
Name            wilsection4
Clock unit      150
Number          100
Notes           (read-from wilhelmus sine-shape)
```

Note that *read-from* uses a default length 100 when using a shape to read from another object. In *wilsection4*, that value is the same as the number of notes in the section. If another length for the shape is desired, the number should be added as the last argument to read-from, e.g.

```
(read-from wilhelmus sine-shape 200).
```

Interpolating between two note structures
Interpolate is a generator suggested by Rainer Boesch. In its most simple form, it performs a linear interpolation between two lists or objects, gradually changing from the first object to the second one. The interpolation takes place in the course of a number of elements. During each step of the interpolation, either an element from *object1* or an element from *object2* is returned. In the default case, the probability that an element from object2 will be returned increases linearly during the interpolation. Object1 and object2 can be lists, stockpiles, note structures, etc.

For example, *(interpolate '(a b c d) '(1 2 3 4) 40)*

```
a       b       c       d
1       b       c       d
a       b       c       d
1       2       3       d
a       2       c       d
a       b       c       4
1       b       3       d
a       b       3       4
a       2       3       4
1       2       3       4
```

Object1 and object2 do not have to have the same length. Each one cycles through its elements. If object1 has 3 elements, and object2 has 4, then either a or 1 is returned, b or 2, c or 3, a or 4, etc.

For example, *(interpolate '(a b c) '(1 2 3 4) 40)*

```
a       b       c
a       b       c
3       b       c
a       b       4
a       b       c
a       b       c
a       b       c
```

```
2       3       4
1       b       3
4       1       2
3       4       1
a       3       4
1       2       3
4
```

Instead of a linearly increasing probability that an element from object2 will be chosen, any shape can be specified with the shape keyword.

For example, *(interpolate '(a b c d e f g) '(1 2 3 4 5 6) 40 :shape sine-shape)*

```
1       b       c       4       5       6       1
2       3       4       5       6       1       2
3       4       5       6       e       2       3
a       b       c       d       e       f       g
a       b       c       d       e       4       g
a       1       c       3       e
```

To use *interpolate* with note structures, first *Make* note structure *battle* (in the file *Tutorial 10 Examples*). Make *battlesection* and play it if you do not recognize the melody from looking at the note structure.

Note section *interpolate1* is a linear interpolation between note structure *wilhelmus* and note structure *battle*.

```
Name            interpolate1
Clock unit      150
Number          400
Notes           (interpolate wilhelmus battle 400)
```

Interpolate2 uses *sine-shape* for the interpolation.

```
Name            interpolate2
Clock unit      150
Number          400
Notes           (interpolate wilhelmus battle 400 :shape sine-shape)
```

Summary

Menu items
note section

Generators
interpolate, read-from

Tools
chaotic-notes

Tutorial 11

Specifying Midi controllers and program changes

Midi objects

Midi hardware and software may allow possibilities for controlling timbre though Midi controller commands and/or program changes. To allow experimentation with the algorithmic control of these parameters, a number of Midi objects have been defined. These objects can be used independently or mixed with other Midi objects or sections, e.g. as a parallel section.

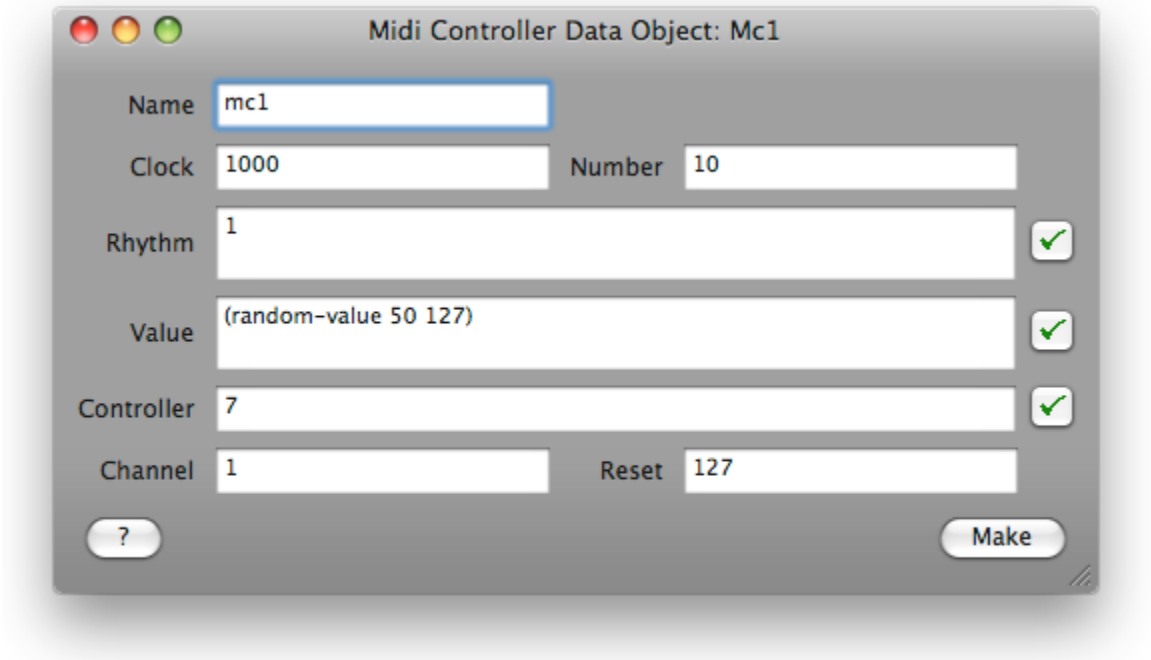
The use of external Midi controllers to control the real-time state of input parameters within the Toolbox is discussed in Tutorial 12.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 11 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box

Midi controllers

Only 7-bit Midi controllers are supported. The class is called *midi-controller-object*.

Controller data object
The menu item **Define>Midi Object>Controller Data** creates a dialog box for a *Midi Controller Data Object*. Information is specified in a way similar to the specification of a data section. Rhythm is a multiple of a clock unit. Controller values and controller numbers are specified in the customary Toolbox manner: as a constant, a list, a stockpile, a generator, etc.



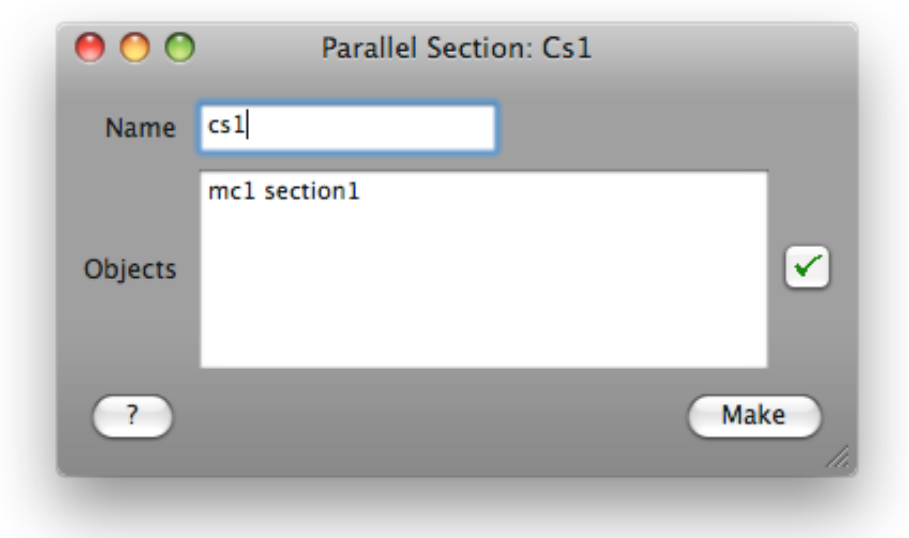
The object will contain 10 controller values. Every 1000 milliseconds (rhythm 1 * clock unit of 1000), a new value generated by *random-value* will be output when the object is played. The values are assigned to controller number 7 and Midi channel 1. Controller 7 is predefined as volume.

The dialog box allows the Midi controller value of the Midi destination to be reset after the object has been played. In the above object definition, the reset value 127 has been entered. After the last value for this object has been sent, the reset value will be sent to the designated controller. If no value is entered for rest, the controller is not reset.

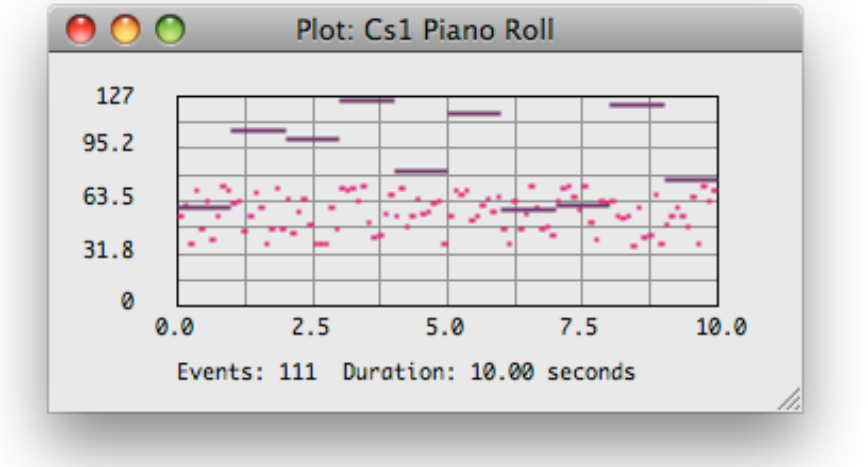
A Midi object such as *mc1* can be played in the same way that a section is played.

Midi objects can be combined with each other or with sections via the menu items for **Sequential Section**, **Parallel Section**, or **Timed Section**.

• Make *section1* from the tutorial file. It is a data section with 100 random pitches. Combine it with *mc1* to produce a parallel section.



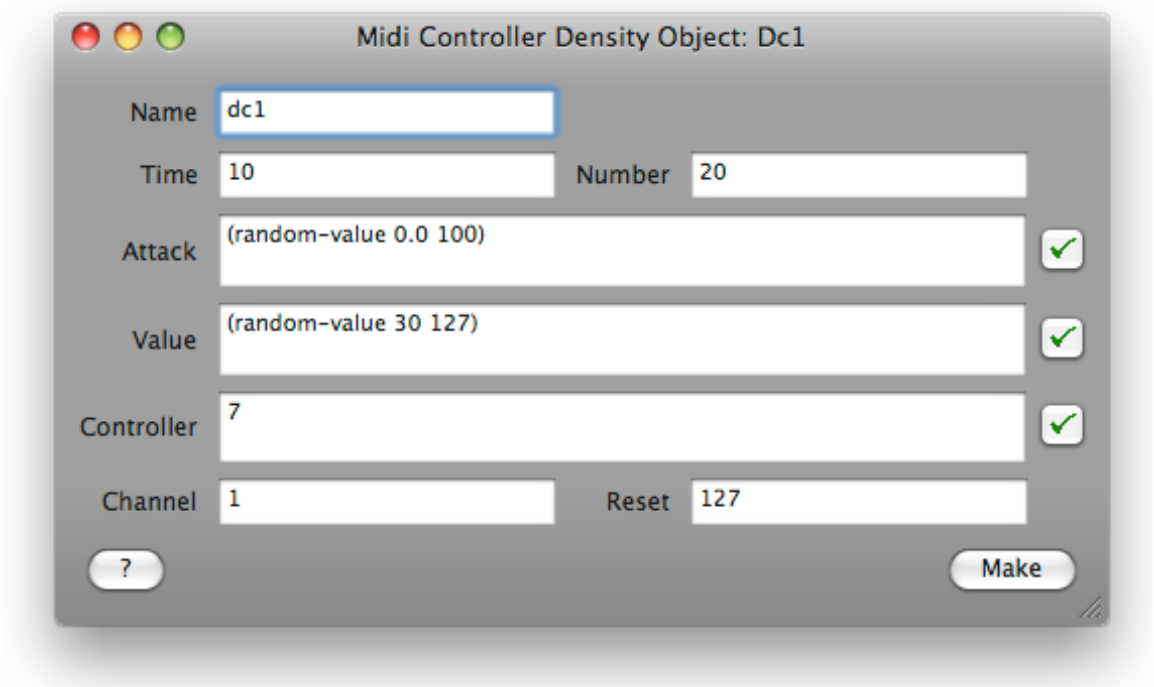
CS1 is a combination of a Midi object and a section. This combination is called a *midi-section*.



When a combination of a section with controller information is plotted, the section is plotted in the default color. Each new controller is plotted in another color. In *cs1*, there is only one controller so there is only one other color. Controller values do not have a separate duration. They last until it is time for the next one.

Controller density object
The menu item **Define>Midi Object>Controller Density** creates a dialog box for a *Midi Controller Density Object*. Information is specified in a way similar to a density section (see Tutorial 8). A certain amount of time (expressed in seconds) is filled with attack points. The attack points are specified using a function that produces values between 0.0

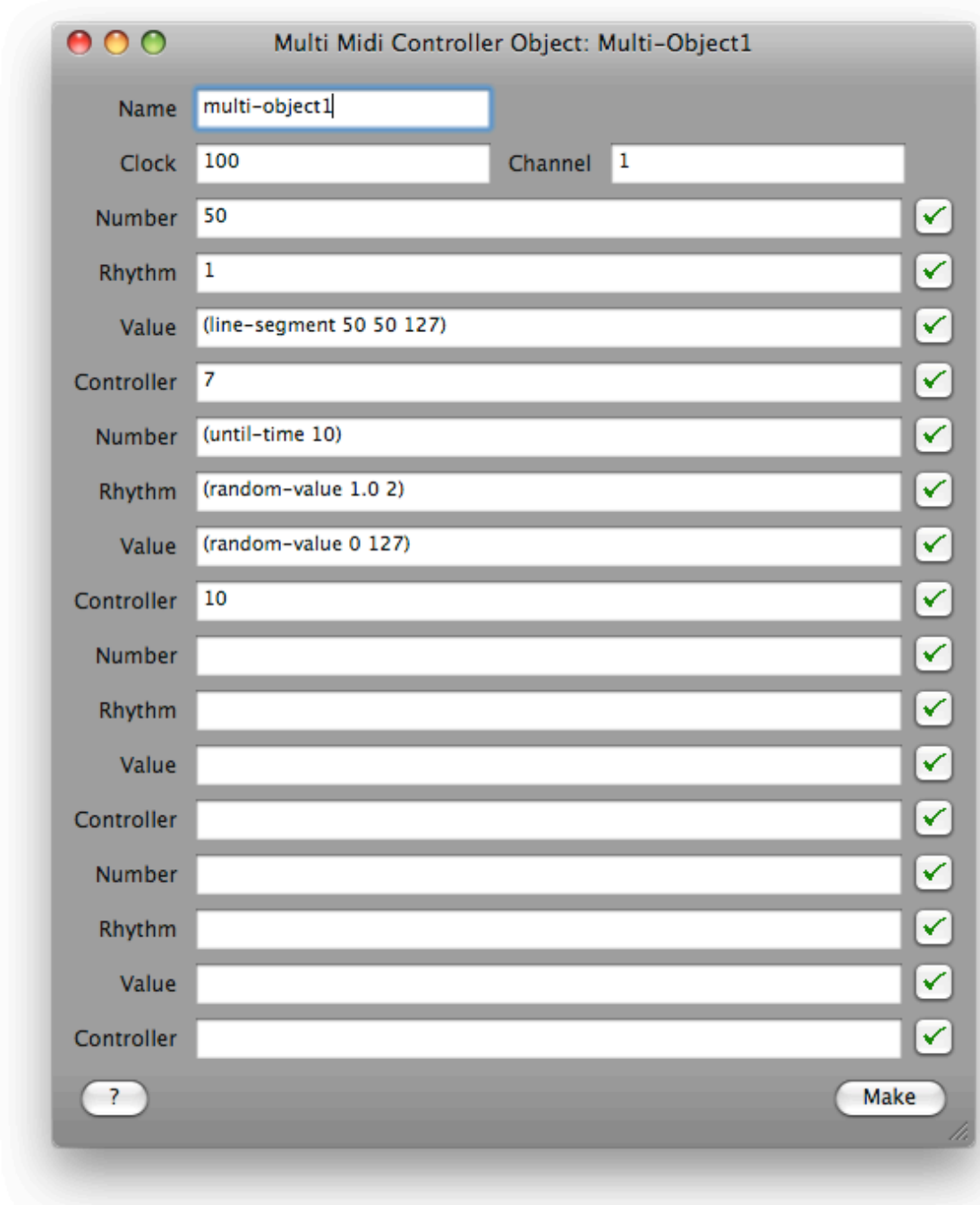
and 100 percent of the available time. Each attack point will be assigned a controller value, a controller number, and a Midi channel all of which can be expressed in the customary manner (constant, list, stockpile, generator, etc.) A controller maintains its value until it receives a new one therefore there is no separate parameter for duration.



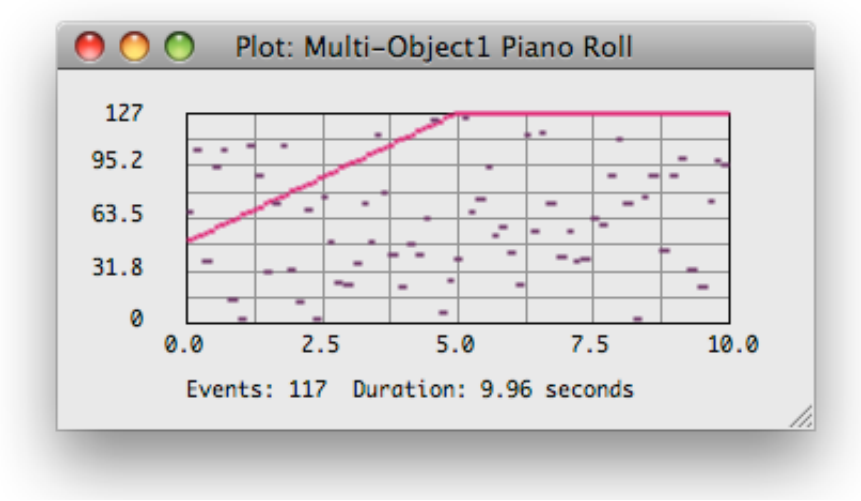
Ten seconds will be filled with 20 Midi controller messages. The attack points are chosen at random between 0 and 100 per cent of the available time (10 seconds). At each attack point, a controller value between 30 and 127 will be sent to controller 7 on Midi channel 1. After the 10 seconds have passed, the controller will be reset to 127 because the *Reset* box has been filled.

Dc1 can be played independently or combined with a section. Object *cs2* from the tutorial file combines *dc1* with *section1* in parallel as midi-section *cs2.*

Multi controller
Several Midi controllers can be specified using a dialog created with the *Multi Controller* menu item. The idea is similar to a controller data object except several controllers can be specified. Each controller has a value for the number of values to be generated, rhythm (as a multiple of the clock unit), the controller values, and the controller number. All controllers share the same Midi channel and clock unit.



With *multi-object1*, two controllers are specified. The first is controller 7 (volume). Fifty values are calculated. These values will increase linearly. Given the rhythm value of 1 and the clock unit of 50 ms, the peak of the line will be reached in 4.9 seconds. The second controller is 10 (panning). Random values for this controller will be calculated until ten seconds have been filled.



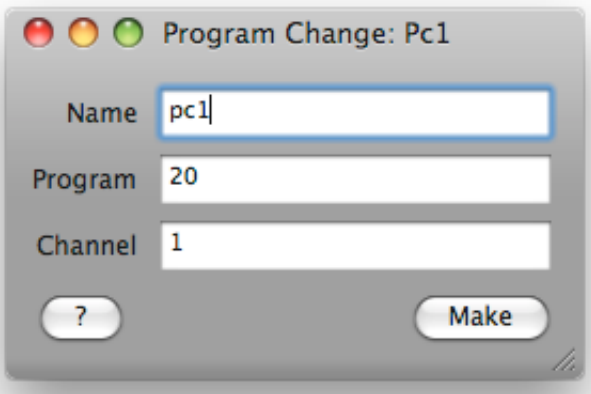
The display shows the linearly increasing volume values and the random panning values in different colors. The AC Toolbox will automatically assign different colors to different controllers when displaying a multi controller object.

This object is convenient if several (up to four) Midi controllers should be specified. The same result could be achieved, with a bit more work, by defining four controller data objects and combining them in a parallel section.

Midi program changes

Midi program changes can be realized through a *midi-program-object*. Each object has a name, one or more program change messages (program and channel) and relative times at which they should occur.

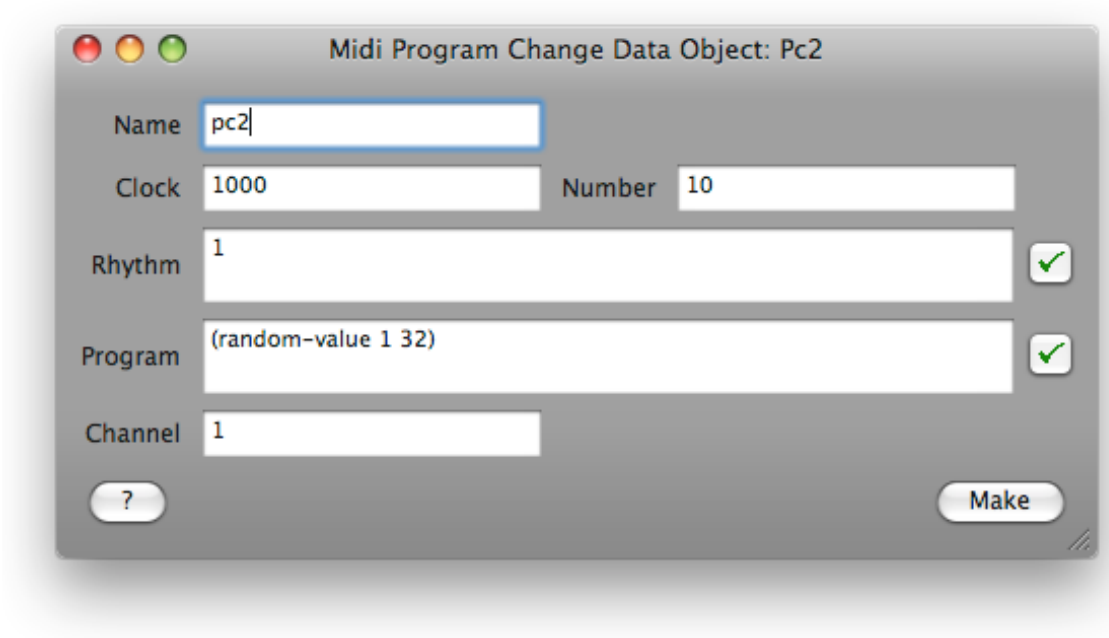
The most simple of the objects can be created by selecting **Define>Midi Object>Program Change**. One program change and one channel number may be entered.



The resulting object can be used in a sequential section to make a program change before or between sections. The object *ps1* from the tutorial file is such a sequential combination. Note that a short rest (1 ms) was inserted in the sequential section specification between the program change and the section.

A midi-program-object can be played in the same was as a section.

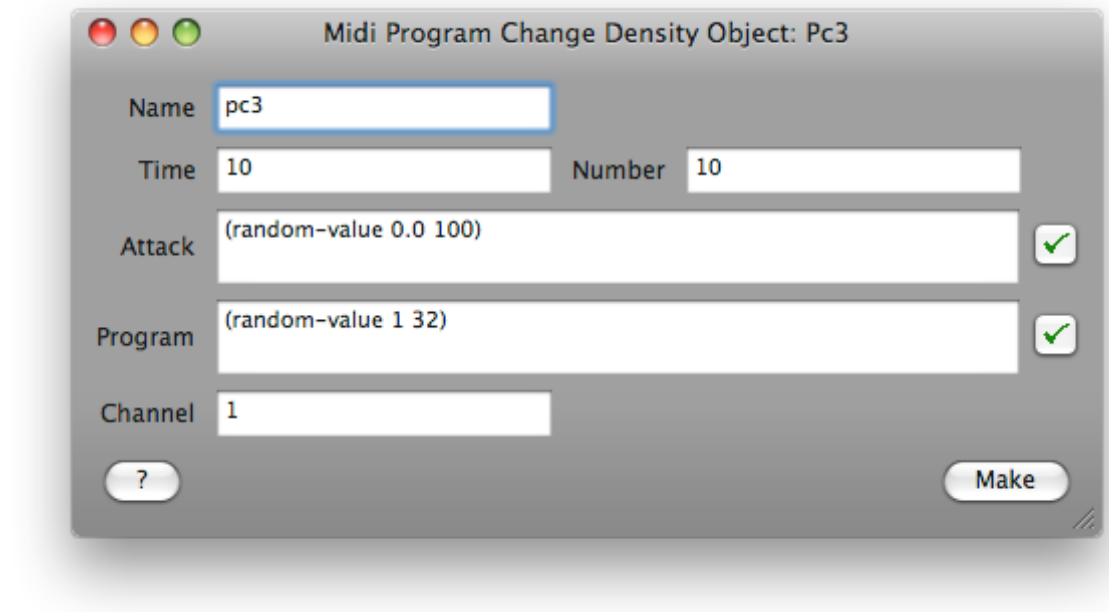
Program data object
Information for a *Midi Program Change Data Object* is specified similarly to a data section. Rhythm is a multiple of a clock unit in milliseconds. A program change and channel are selected at points corresponding to each rhythmic value.



An object with 10 events is specified. Each event contains a program change command. An event occurs every second (rhythm 1 * clock unit 1000 milliseconds).

Object *ps2* in the tutorial file combines *pc2* with *section1* in parallel. A millisecond delay was specified between the start of *pc2* and the start of *section1*.

Program density object
Information for a *Midi Program Change Density Object* is specified similarly to a density section. A length of time is specified in seconds. A number of attack points are calculated within that time. For each point, a program change and channel are specified.



Ten seconds are filled with 10 events. Each event contains a random program change between 1 and 32 for Midi channel 1.

Object *ps3* in the tutorial file combines *pc3* with *section1* in parallel.

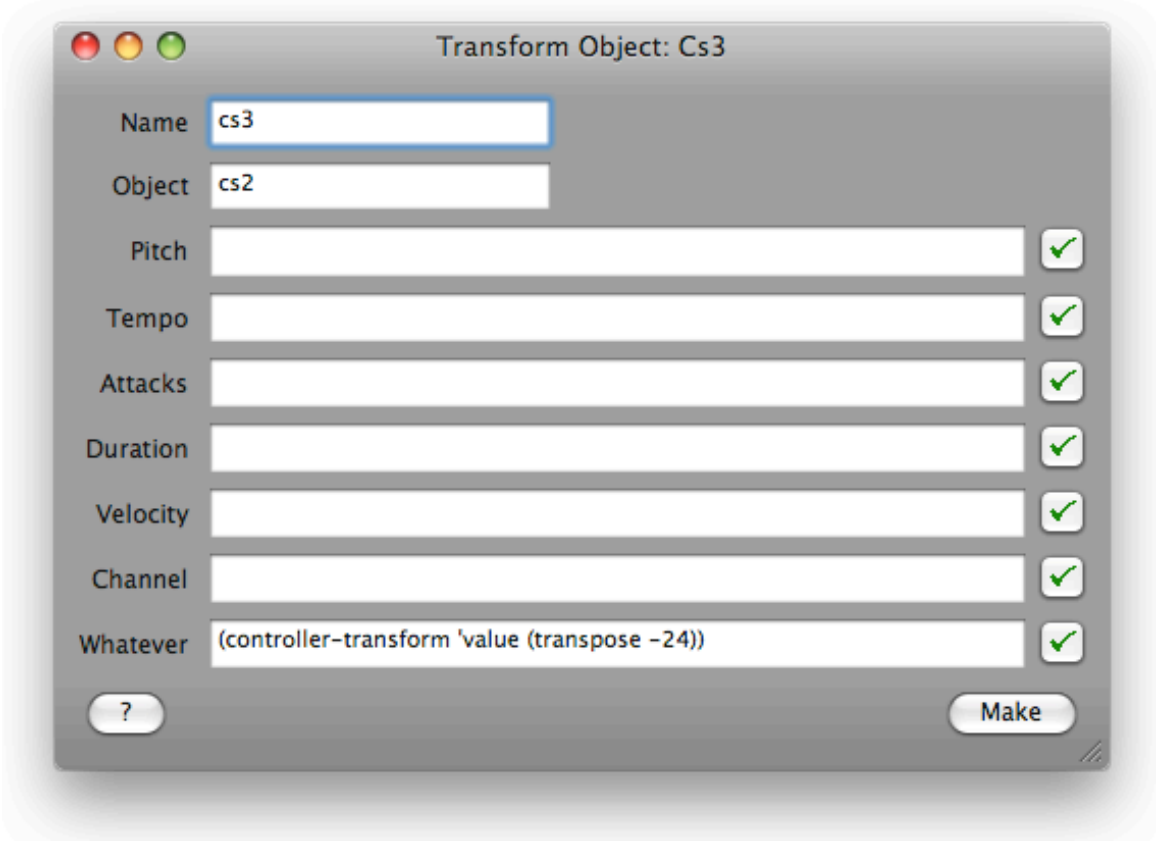
Manipulating Midi objects

Individual parameters of Midi objects cannot currently be plotted or have histograms made of them. They can be plotted using the *Plot* button in the *Objects* dialog. A plot of a midi-controller-object will be the controller values over time. A plot of a midi-program-object will be the program numbers over time.

A list of all Midi objects that have been made, can be found using the *Midi Objects* popup menu item in the *Objects* dialog. This list does not include midi-sections because they are reported with the other sections.

Midi objects can be written to a Midi file. They can be imported from a Midi file. The methods such as *backwards*, *join*, etc. can be applied to them. The methods *transform* and *filter* require special attention.

Transforming Midi objects
The *Transform* dialog box has no box for choosing a transform for a controller value. To be able to transform a controller value in either a midi-section or a midi-controller-object, the *whatever* box should be used together with a function that specifies that the transform should be applied to the controller value. This function should 'encapsulate' the transform function.



Controller-transform encapsulates the transformer and indicates that it should only be applied to controller data. Object *cs2* contains both note and controller commands. The transformation in *whatever* will only be applied to the controller values.

Controller parameters can be transformed by specifying *value*, *number*, or *channel* in the *controller-transform* expression:

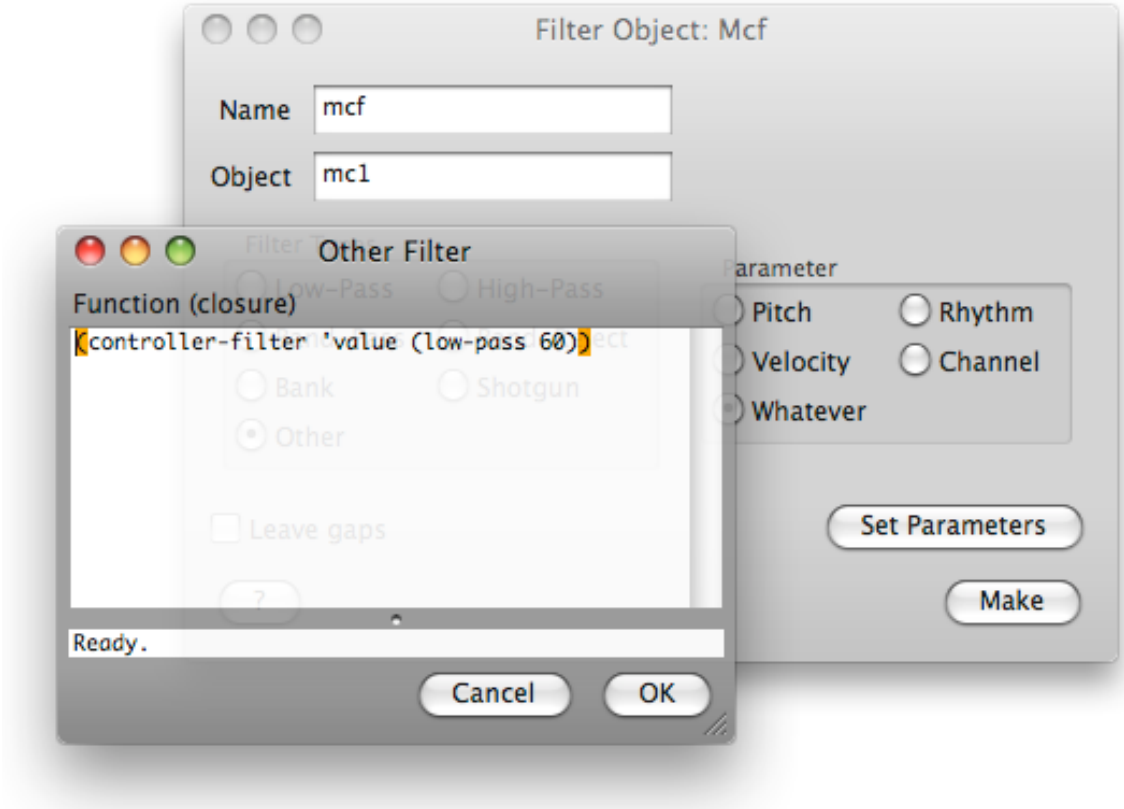
(controller-transform 'value transformer)
(controller-transform 'number transformer)
(controller-transform 'channel transformer)

Program changes may be transformed in a similar way by using *program-transform*:

(program-transform 'number transformer)
(program-transform 'channel transformer)

Filtering Midi objects

The *filter* dialog box has no button for filtering Midi objects. Instead the option for *Other filter* should be chosen. An expression that will produce an appropriate filter form should be entered. The idea is similar to what is used with transform: a filter expression is encapsulated in a form that indicates which parameter is to be filtered.



(controller-filter 'value (low-pass 60)) creates a low-pass filter for controller values.
The various filters should be entered as a Lisp expression:

(low-pass threshold)
(high-pass threshold)
(band-pass low high)
(band-reject low high)
(bank low1 high1 low2 high2 ...)
(shot-gun percentage)

The available specifiers are:

(controller-filter 'value filter)
(controller-filter 'number filter)
(controller-filter 'channel filter)
(program-filter 'number filter)
(program-filter 'channel filter)

Summary

Menu items
controller data, controller density, multi controller, program change, program data, program density (midi objects)

Transformers
controller-transform, program-transform

Tools
controller-filter, program-filter

Tutorial 12

Streams

A stream is like water coming from a faucet. When the faucet is turned on, water flows until it is turned off. In the AC Toolbox, several streams can flow at the same time.

A stream, when turned on, will produce Midi commands for notes, controllers or program changes.

The structural description of a stream is similar to the descriptions for sections or some Midi objects. There is an important difference: there is no predetermined limit on the number of values. A section contains some finite number of predetermined notes. A stream can continue producing notes as long as it is running. It does not remember the notes it has made.

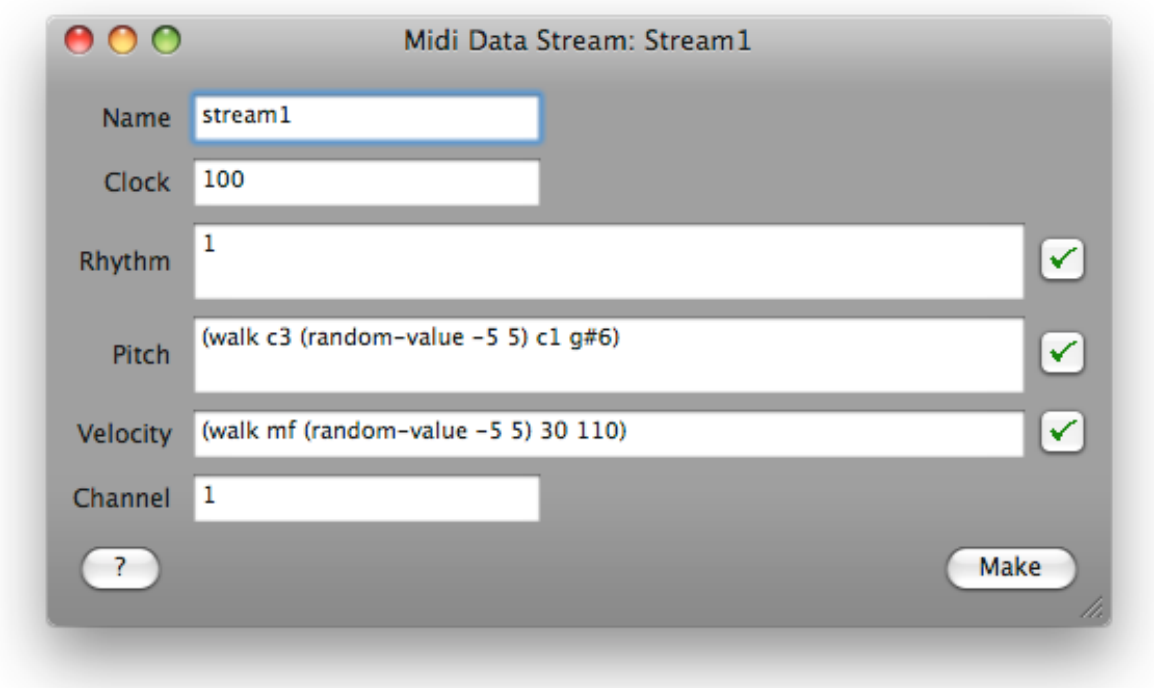
Since streams produce values in real-time, external parametric control becomes possible. This control could come from Midi controllers, IAC input, number boxes associated with *test-value,* or a set of on-screen sliders.

The price you pay for this interactivity is the possibility of scheduling mistakes from overloading the program. The AC Toolbox is not intended as a program for real-time performance. The stream facility is a convenient way to quickly test behavior and parameter values. How reliable a stream is depends on the speed of the machine, the available memory, and what the stream is doing.

Currently, a *data stream* (similar to a data section), *note stream* (note section), *controller stream* (Midi controller data object), multi-controller stream (a multi Midi controller object), *program stream* (Midi program change data object), and *parallel stream* (parallel section) are available.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 12 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Data stream
The specification of a data stream is similar to the specification of a data section. All parameters are treated independently of each other. The big difference is that the number of notes is not fixed.



A stream starts producing values when it is turned on by the **Play** button. It stops producing values when the **Stop** button is clicked. When a stream is turned on again, it continues from where it was. A section, if it is played again, starts at the beginning and is always the same.

If *Make* is clicked with the right mouse button (or with CTRL-Click), a *Play* popup menu item appears. This item can also be used to play or stop the stream.

External controllers
Several generators in the Toolbox allow their input parameters to vary over time. When these generators are used in a stream, their parameters can be controlled by (external) Midi controllers. The generator *external-value* reads a seven-bit Midi controller and maps its value to be between two specified limits. The format is:

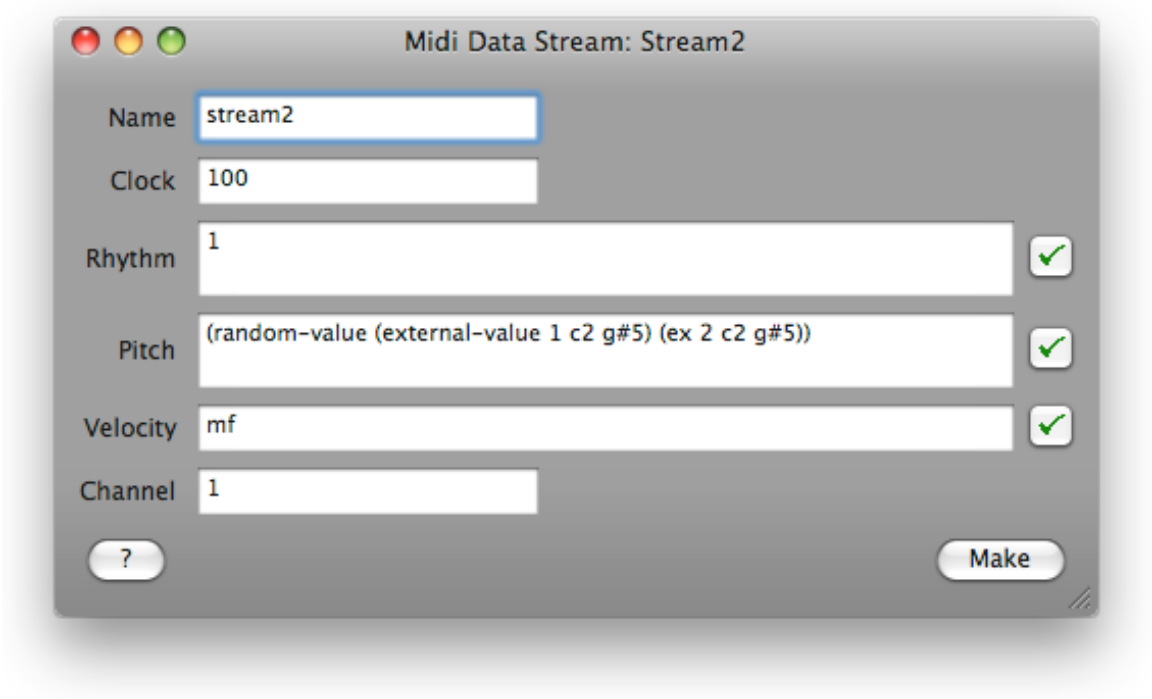
```
(external-value controller-number low-value high-value &key round)
```

External-value can be abbreviated to *ex*, e.g.:

```
(ex 1 c2 c5)
```

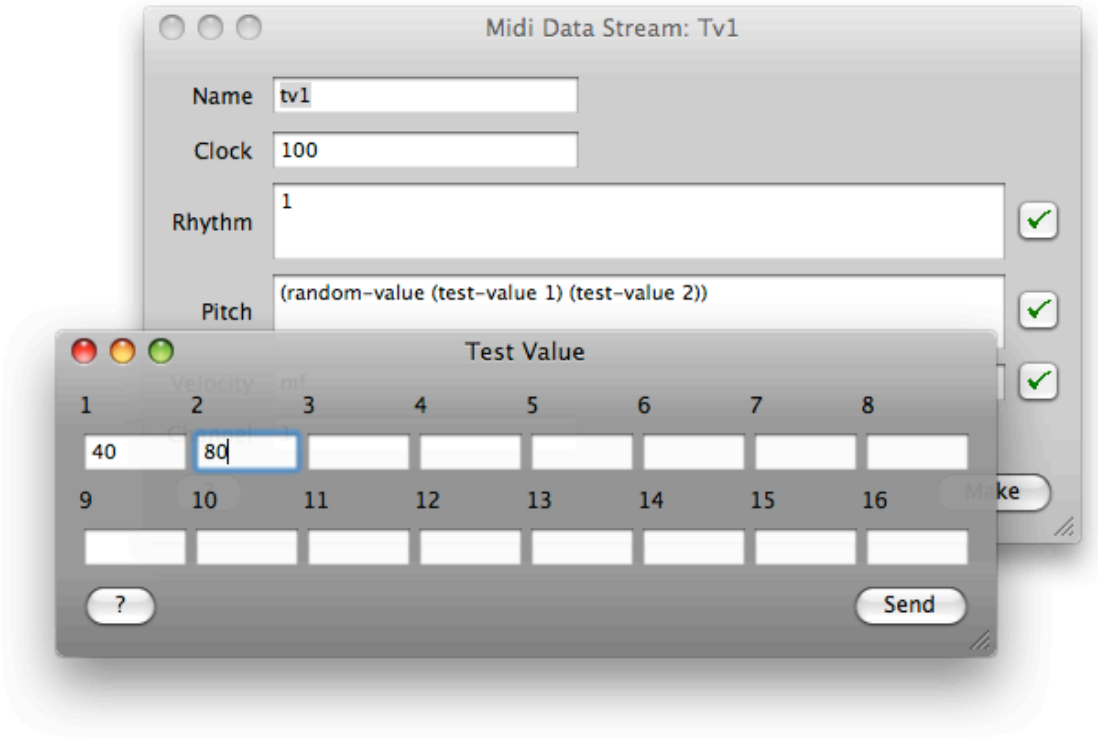
The way in which the external controller data is mapped is controlled by the *External Value* option in the *Preferences*. The generator external-value maps either the values of a particular Midi controller (e.g. 1-127) regardless of the channel (radio button *Controllers*) or the values of a Midi controller on a particular channel (e.g. 1-16) regardless of the controller number (radio button *Channels*). *Controllers* is the default mapping.

Object *stream2* uses an external Midi controller such as a fader box to control the lower and upper limits of a *random-value* generator. The input device or source should be selected in **Midi Setup**. Controller 1 is for the lower limit that can be between *c2* and *g#5.* Controller 2 is for the upper limit and can also be between *c2* and *g#5*.



Note the use of both external-value and its abbreviation *ex*. *External-value* can be used anywhere where a generator is allowed, including for the *Clock Unit*.

Test-value
Streams can also be controlled in real-time using the *Test Value* dialog available via the **Tools** menu. The dialog contains boxes numbered 1 - 16. The contents of these boxes can be read in a stream using the *test-value* generator. The input for this generator should be the number of the dialog box that will provide the values.

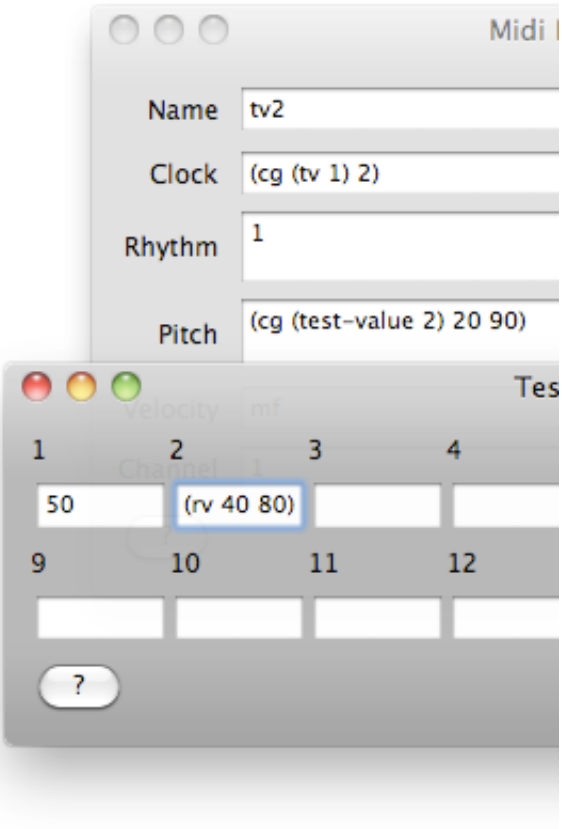


In object *tv1*, pitch will be randomly chosen between the limits specified in test-value boxes 1 and 2. When a new value is entered, *Send* should be selected to send the value to the generator. Keyboard shortcut Cmd-B can be used instead of clicking on *Send*.

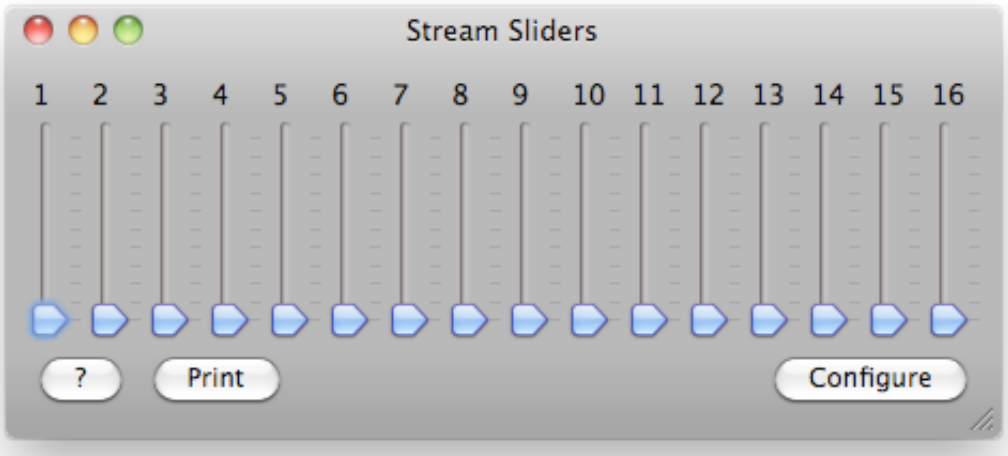
Test-value can be abbreviated *tv*, e.g. (*random-value (tv 1) (tv 2)*).

Generators can be entered into the edit boxes of *Test Value*. In object *tv2*, the pitch specification is *(cg (test-value 2) 20 90)*. If a generator is entered into the second edit box for *Test Value*, it will be used to generate pitch values.

Cg is the clipping generator. In the above example, it limits pitch values to the range 20-90. The clock unit is read with *(cg (tv 1) 2)*. In this case, *cg* limits numbers read from *Test Value* to be >= 2. An upper limit was not supplied.



Slider-value
On-screen sliders are available to control streams. A set of sliders can be created via **Tools>Stream Sliders**. The configuration option allows from 8 to 64 sliders.



A slider position is mapped to a parameter value in the range *low* to *high* using the generator *slider-value*.

```
(slider-value slider-number low high &key round)
```

If either *low* or *high* is a real number, the result is a real number. Otherwise the result is an integer.

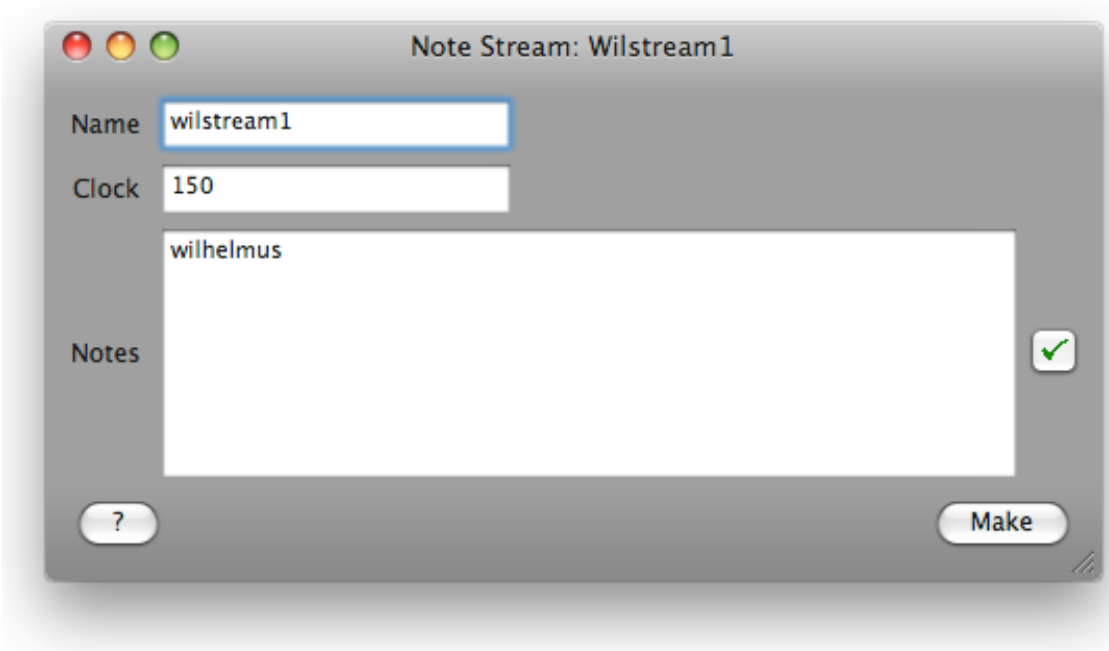
Slider-value can be abbreviated to *sl*.

Stream *slider1* uses the sliders 1 and 2 to determine the limits for randomly generated pitch values. Sliders 3 and 4 are limits for choosing rhythm values. Slider 5 controls velocity.

```
  Name            slider1
  Clock unit      50
  Rhythm          (random-value (slider-value 3 1.0 3) (slider-value 4
1.0 3))
  Pitch           (random-value (sl 1 c2 c6) (sl 2 c2 c6))
  Velocity        (sl 5 50 100)
  Channel         1
```

The Print button on the slider pane prints the values of the sliders after they have been mapped with slider-value. The values are printed in the Text Output window.

Note streams
A *note stream* is similar to a note section except it never ends. Notes are specified as an entity rather than as independent parameter values for rhythm, pitch, etc. Input for a note stream can be a note structure, a generator that produces notes, a tool that produces notes, a list of notes, etc. Note structures are described in Tutorial 9 and note sections in Tutorial 10.

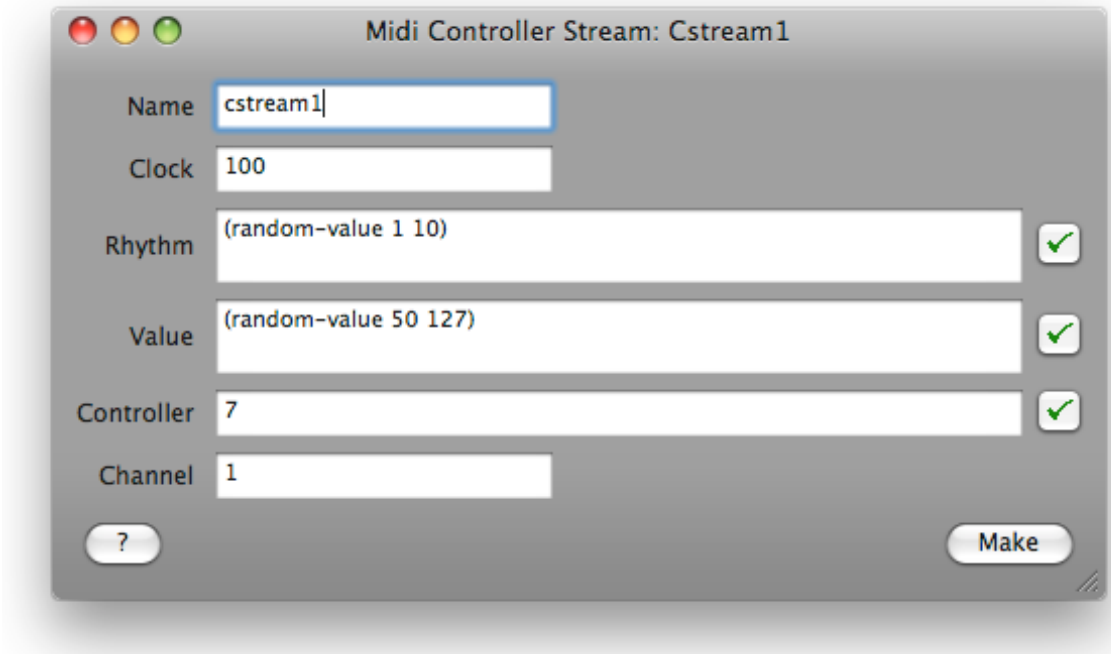


Wilhelmus is a predefined note structure containing notes for the melody of the *Wilhelmus*. *Wilstream1* will continuously play the *Wilhelmus* until it is stopped.

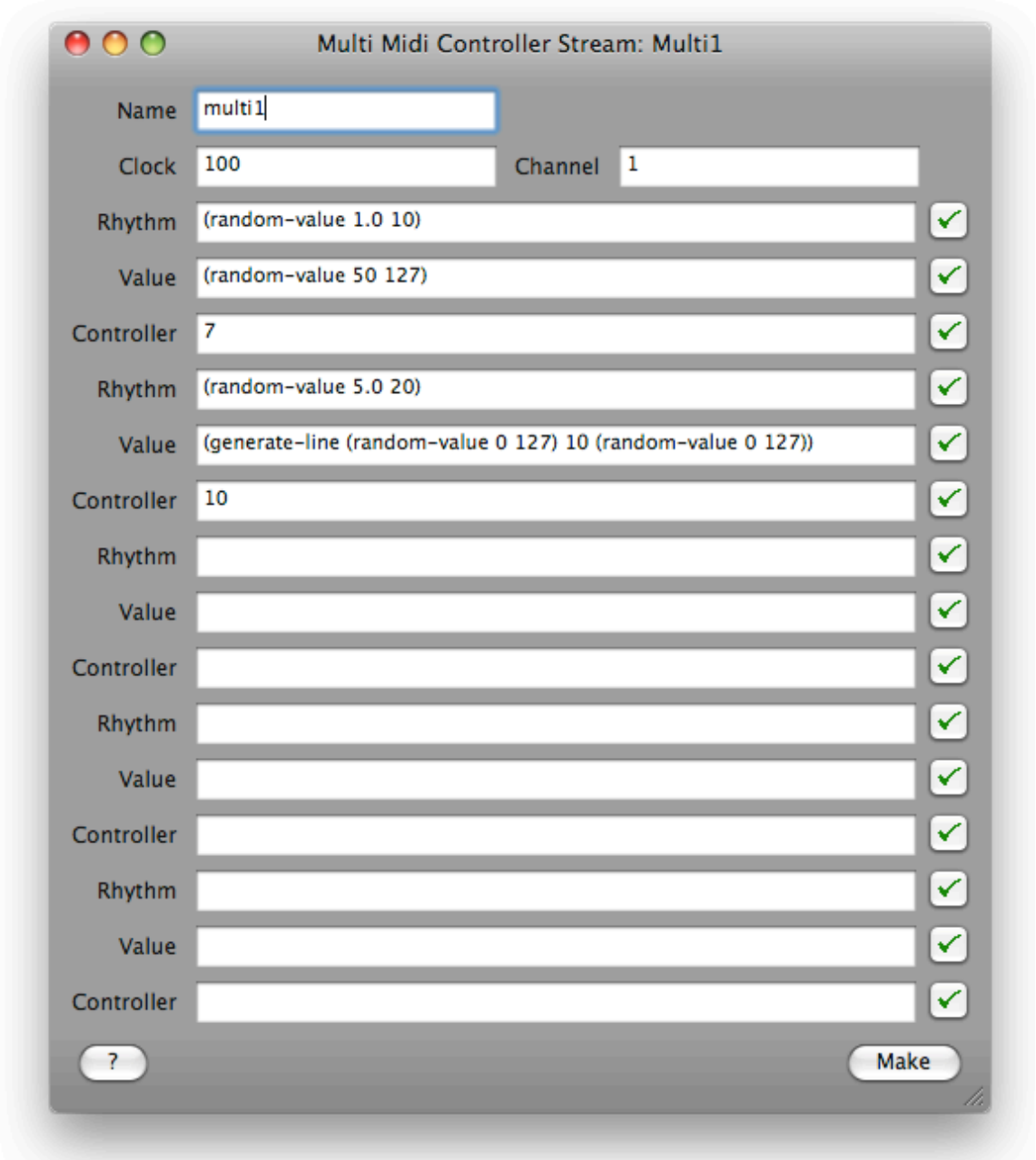
Other possible inputs for a note stream include *(random-choice wilhelmus)*, *(beta-choice wilhelmus .2 .3)*, etc. Choice generators choose from a stockpile, note structure, etc. See the help windows for more information.

Controller streams
A *controller stream* continually produces Midi controller messages. It is analogous to a Midi controller data object. A new value for the specified controller is produced each rhythm value * the clock unit.

Cstream1 produces random values between 50 and 127 for controller 7, which normally is volume.



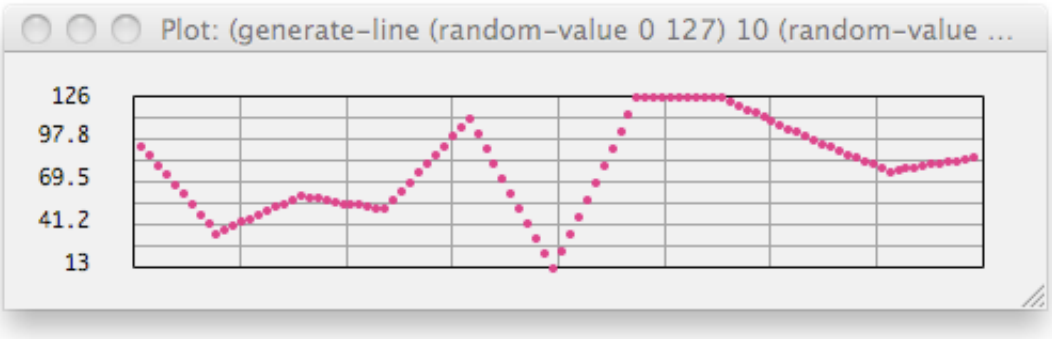
Multi-controller streams
A *multi-controller stream* is similar to a multi Midi controller object. Several controller streams can be specified as part of one object.



Multi1 produces random volume values (controller 7) and linearly changing panning values (controller 10). In this object, panning is controlled by the generator *generate-line*. The initial value is chosen with random-value, a line with ten steps is produced to the new value chosen by a second random-value generator, then there are ten steps to the next value chosen by the second generator, etc.

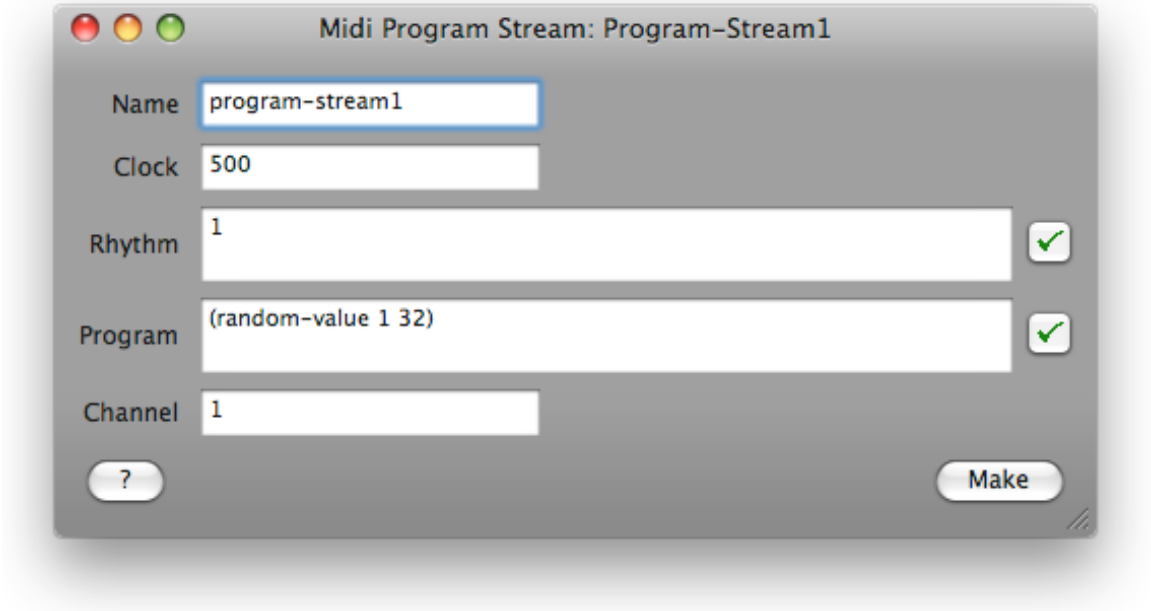
An example of plotting 100 values produced by

```
(generate-line (random-value 0 127) 10 (random-value 0 127))
```



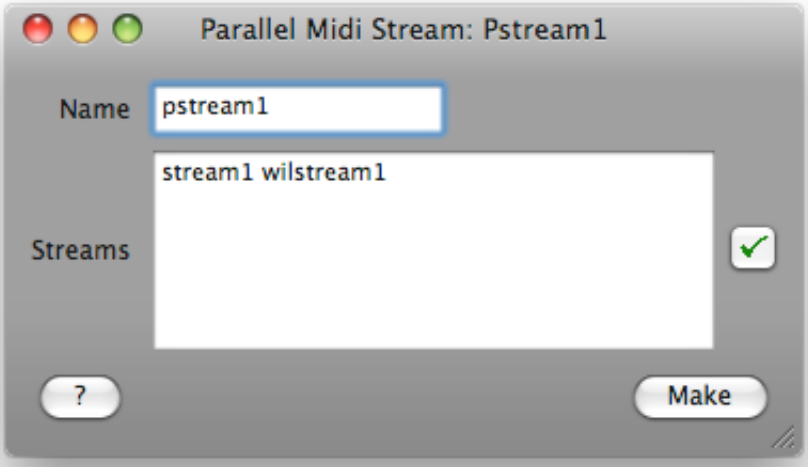
Program streams
A *program stream* continually produces Midi program change messages. It is similar to a Midi program change data object.

Program-stream1 produces random program change messages twice a second.



Parallel streams
A *parallel stream* is similar to a parallel section except its elements should be streams. A parallel stream will start and stop all of its streams at the same time when played. A parallel stream can combine any type of stream.

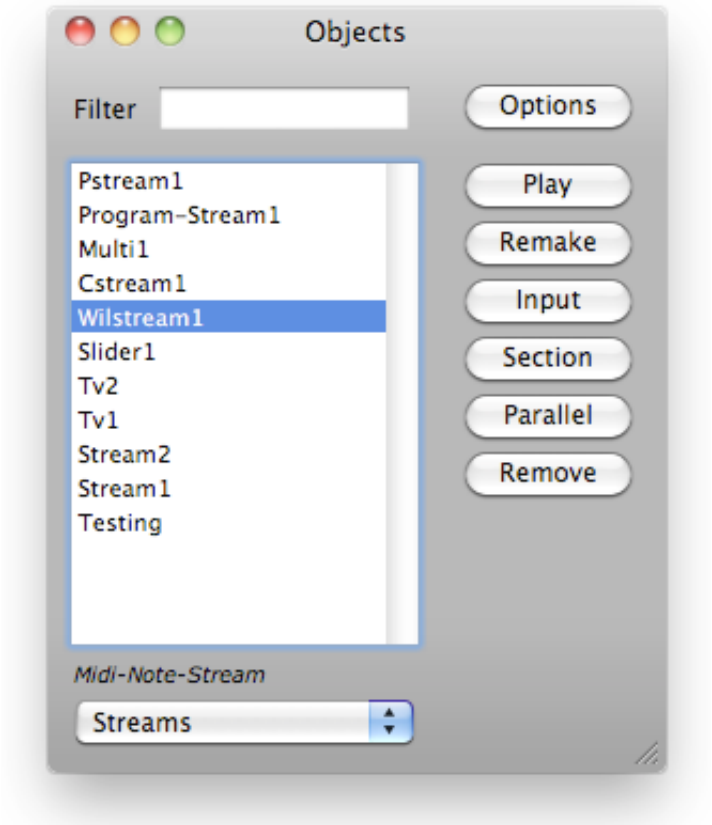
Object *pstream1* is an example of a parallel stream.



To return all the streams in a parallel stream to their initial state, select the **Remake** button in the *Objects* dialog.

Stream generators
The Annotated Index includes a separate category for **Stream**. While any generator can be used in a stream, the ones listed in this category allow for time-varying behavior that is particularly useful when defining streams. These generators include *map/time*, *transform/time*, and *masks&values*.

Streams to sections, sections to streams
A stream specification can be opened in a dialog for a section when the popup menu item for **Streams** has been selected in the **Objects** dialog. Choose a stream, then click on the **Section** button. This facility is available for data streams and note streams.



Similarly, a section specification can be opened in a dialog for a stream in the **Objects** dialog when the popup menu item for **Sections** has been selected. Choose a section, then click on the **Stream** button. This facility is available for data sections and note sections.

Summary

Menu items
data stream, note stream, controller stream, multi-controller stream, program stream, stream control, test value

Generators
external-value, ex, test-value, tv, slider-value, sl, generate-line, cg

Preferences
external value

Tutorial 13

A level higher: controllers and schemes

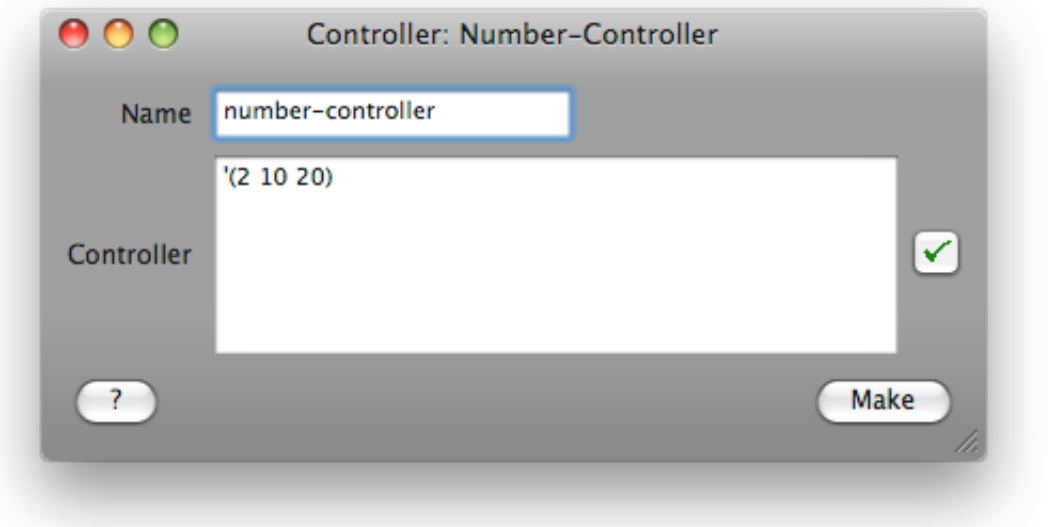
Controllers

Generators used in specifying a section have no state (history) outside of the section specification. Each time a section is specified, a new generator is made that knows nothing about what happened previously. No relationships among sections can be specified by using generators in the specification of sections.

A *controller* is a Toolbox object that retains its state outside of a section specification. It remembers what it has done and knows what it should do. It is useful for specifying relationships at a higher level than those within one parameter of a section. A controller should *not* be confused with a Midi controller.

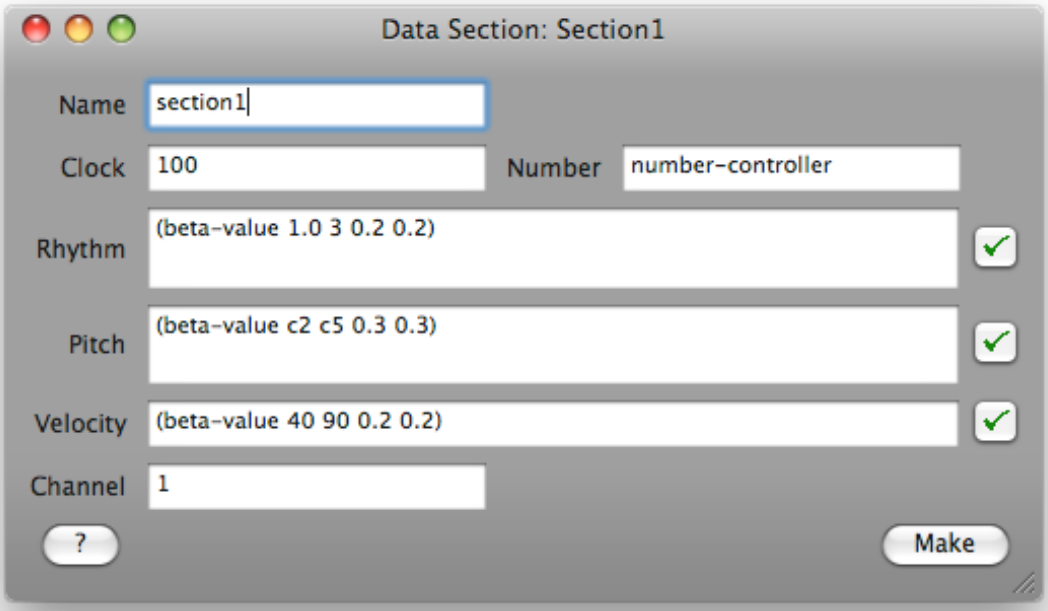
The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 13 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

A controller can have a list, stockpile, or generator as input. Each time the controller is used, the next value from the controller will be returned. If *number-controller* is used to control the number of notes in a section, the first time it is used there will be 2 notes, the second time it is used 10, and the third time 20 notes.



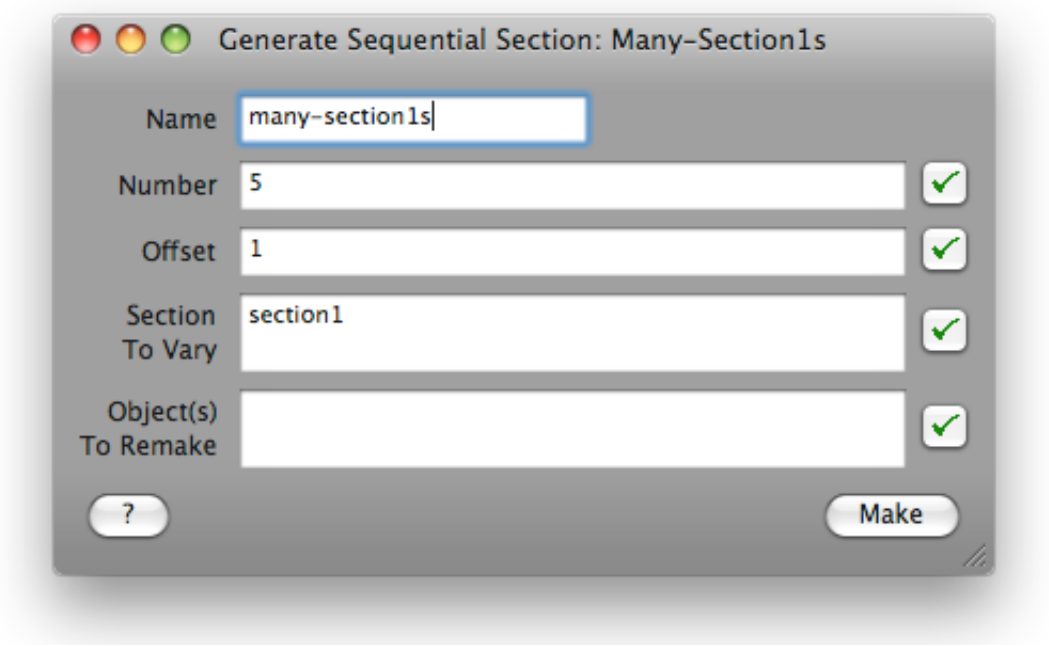
• Make controller *number-controller*.

Using controllers
A controller can be used for any parameter that accepts various types of input. If an input parameter accepts lists, stockpiles, generators, etc., it will also accept a controller. In *section1*, a controller is used to specify the number of notes in a section.



• *Make section1, section2,* and *section3. Section1* has 2 notes, *section2* has 10 notes and *section3* has 20. To confirm this, check the info for each of these sections. (In the *Objects* dialog, choose the *Section* popup menu item, select the section, and then select *Info*.)

• *Make many-section1s*. It generates a sequential section by varying *section1*. Between each variant is a delay of 1 second. This is an example of change on a higher level being controlled with controllers.

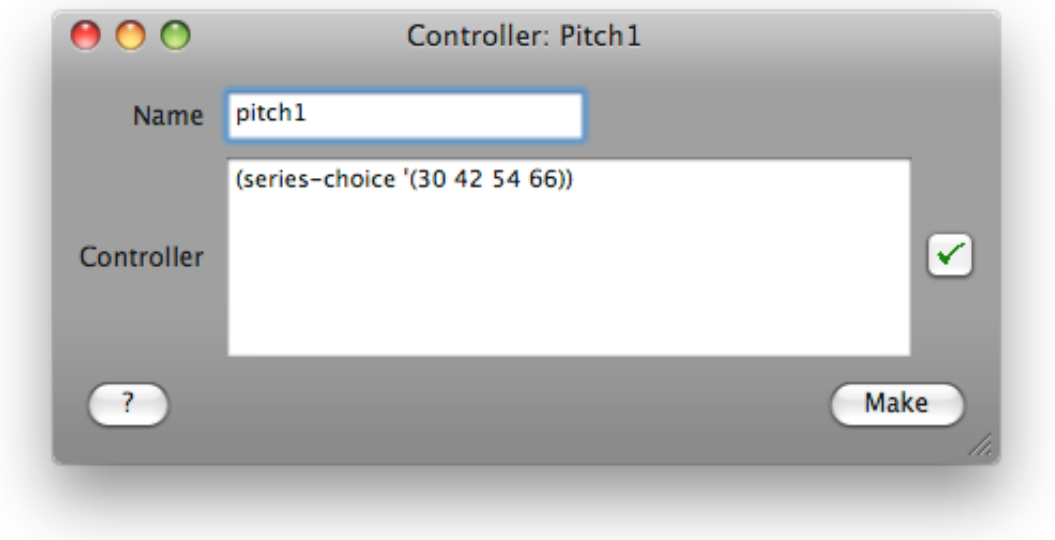


The values produced by a controller can be requested in the *Objects* dialog: choose the popup menu item for *Controllers* and then click on the button labeled *History*. A controller can be *Reset* from this table (that is, the history can be wiped out and the controller will start again from the beginning of its series). This can also be done with the popup menu made by right-clicking (CTRL-Click) the *Make* button of the controller dialog.

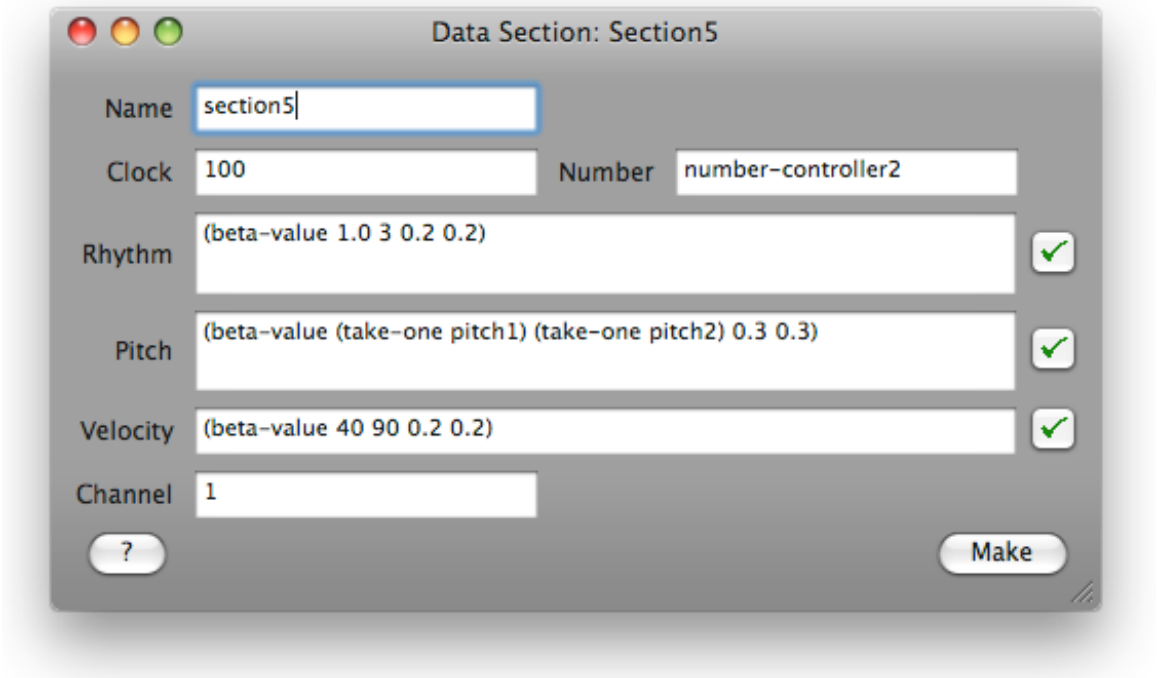
A controller can also be specified with a generator. If the generator *(series-choice '(5 10 20))* is used for a controller, the three values will occur in a random order, but none will be repeated until they all have been used once. If this is applied to the number of notes in sections, each group of three sections will have one with five, one with ten, and one with twenty notes.

• Make controller *number-controller2*.
• *Specify* (not *make*) *section4*. When an object is specified, its definition is stored but the object is not made. In the case of section4, when it is made, the controller is applied. When it is specified, the controller is not yet applied. (To specify an object, hold down *Cmd* when clicking *Make*.)
• *Make many-section4s*. Play it and identify the variants and be aware of their various lengths.

Take-one
In *section4*, the lower and upper boundaries for determining pitch are the same for each variant in the section *many-section4s*. These values could be varied with a controller, picking one value for each section. *Pitch1* will control the lower boundary and *pitch2* will determine the upper boundary.



• *Make pitch1* and *pitch2*.



Take-one returns the next value from a controller. When it is used to specify input to a generator as in *section5,* only one value will be used for the entire section. If the name of the controller had been used instead of the form *(take-one pitch1)*, beta-value would have accessed a value from the controller every time it calculated a value. *Take-one* accesses one value and passes that one value to beta-value for all of the calculations.

• *Specify* section5 (Cmd-click on *Make*).
• *Make many-section5s*. You should be able to hear (or at least see) the different combinations of boundaries for the beta-value generator.

• *Make rhythm-high* and *velocity-low*. They are controllers that will be used for *section6. Rhythm-high* has the following input specification: *(line-segment 5 3.0 1.5)*. *Line-segment* is a generator that produces *n* values between a *first* and a *final* value. In this case, 5 values between 3.0 and 1.5 are produced: 3.000, 2.625, 2.250, 1.875, 1.500. *Velocity-*

low has the following input specification: *(line-segment 5 40 70).* Five values between 40 and 70 will be produced.

• *Specify* (Cmd-click Make) data section *section6*.

```
Name            section6
Clock unit      100
Number          number-controller2
Rhythm          (beta-value 1.0 (take-one rhythm-high) 0.2 0.2)
Pitch           (beta-value (take-one pitch1) (take-one pitch2)
                                0.3 0.3)
Velocity        (beta-value (take-one velocity-low) 90 0.2 0.2)
Channel         1
```

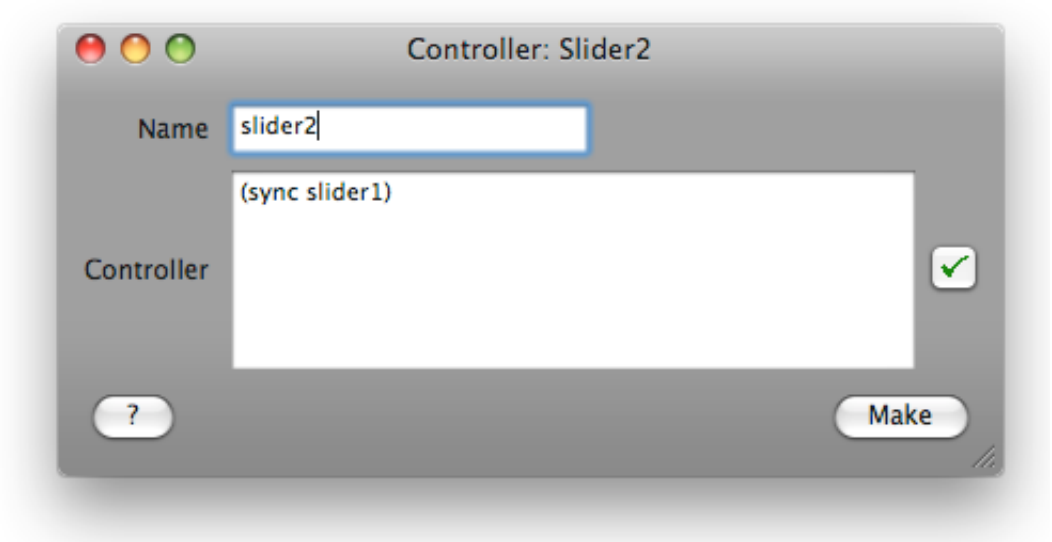
• *Make many-section6s*. This varies *section6* 5 times. The pitch boundaries will change for each generated section. The upper boundary for rhythm will decrease for each section, causing each section to tend to be faster than the previous one. The lower boundary for velocity will increase for each section, causing each section to tend to be louder than the previous one. If this is not the case, reset the controllers used in *section6* and make *many-section6s* again.
• Specify a section using controllers to determine the clock unit and the number of notes in a section. Generate a sequential section of variants of this section.
• Specify a section using controllers to control the upper and lower boundaries for the pitch specification. Generate a sequential section of variants of this section.

Using the same value
Some times the same value needs to be used or referred to in different parameters of an object. An example would be if one wanted short rhythmic values to have a low velocity value and long values to have a high velocity value. One solution would be to calculate a stockpile of values first and then use the same stockpile in the specification of both parameters. Another solution is to use the same controller for both parameters. This has the advantage that no stockpile has to be calculated first. The disadvantage is that some additional work is required to make sure that the controller does not produce different values in the different parameters since it would be applied twice.

The solution to the problem is to synchronize two controllers to a third controller that produces the values. Each of these synchronized controllers will return the next value from the third controller. The synchronized controllers force the third controller to produce a value when it is needed.

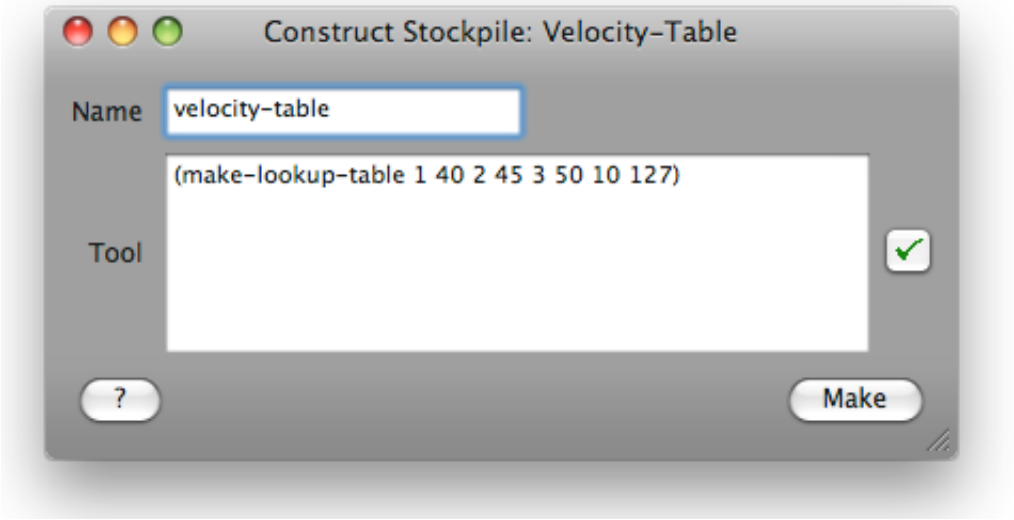
A simple example involves relating velocity values to generated values for rhythm by using a lookup table.

Slider1 is the controller that will generate rhythmic values. It uses the form:
`(random-choice '(1 2 3 10)).`
Slider2 and *slider3* are synchronized to it.

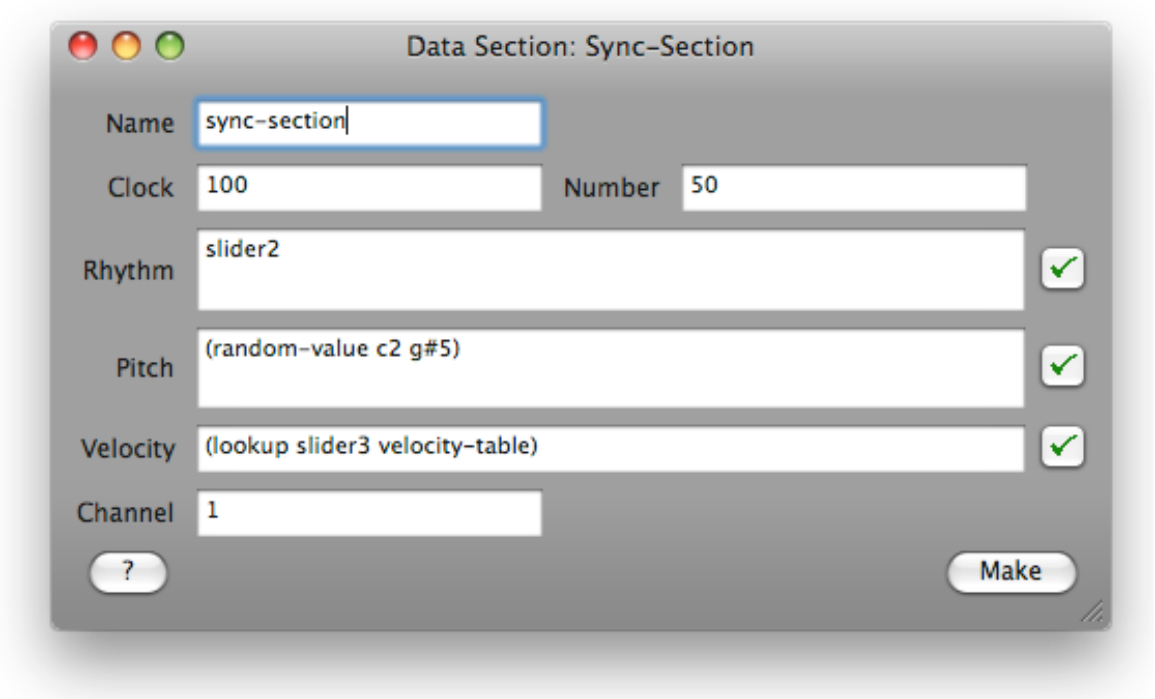


• Make controllers *slider1, slider2,* and *slider3*.

A lookup table will be used to relate the generated rhythmic values with velocity values. Rhythm value 1 will be mapped to velocity value 40, rhythm value 2 will be velocity 45, etc. The lookup table is constructed as a stockpile.



• Make stockpile *velocity-table*. This will be used as the lookup table.
• Make section *sync-section*. This section uses controller slider2 for rhythm, and slider3 as input to the lookup generator that maps the rhythm value to a velocity value using the previously defined lookup table.



Schemes

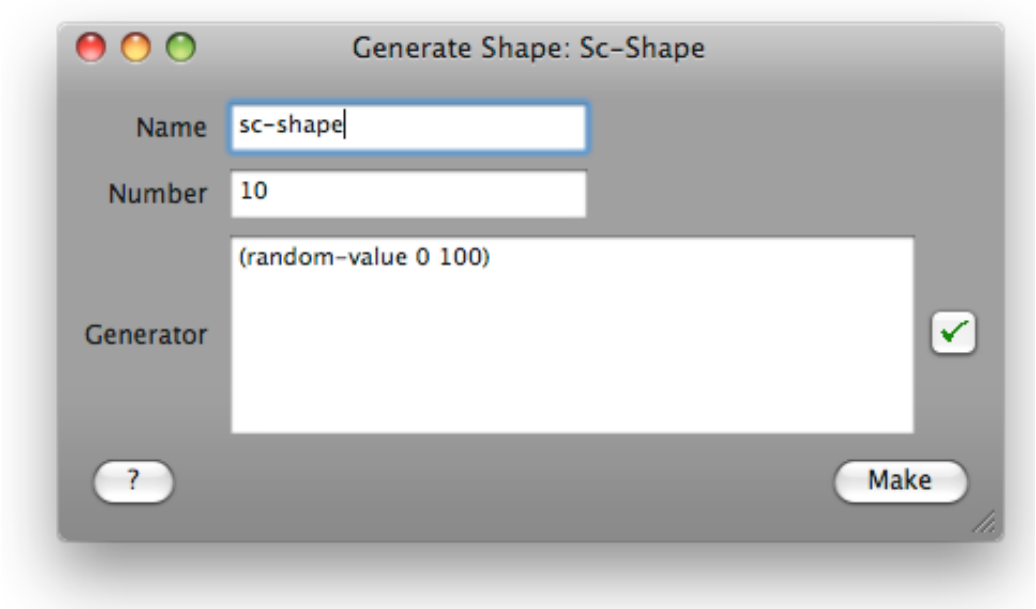
A *scheme of variation*s contains the names of objects that are to be remade in a certain order. When the scheme is applied, those objects are made in that order, from left to right, top to bottom. If an object is made as part of a scheme, the previous output of that object is lost.

The two basic reasons to use a scheme of variations are:

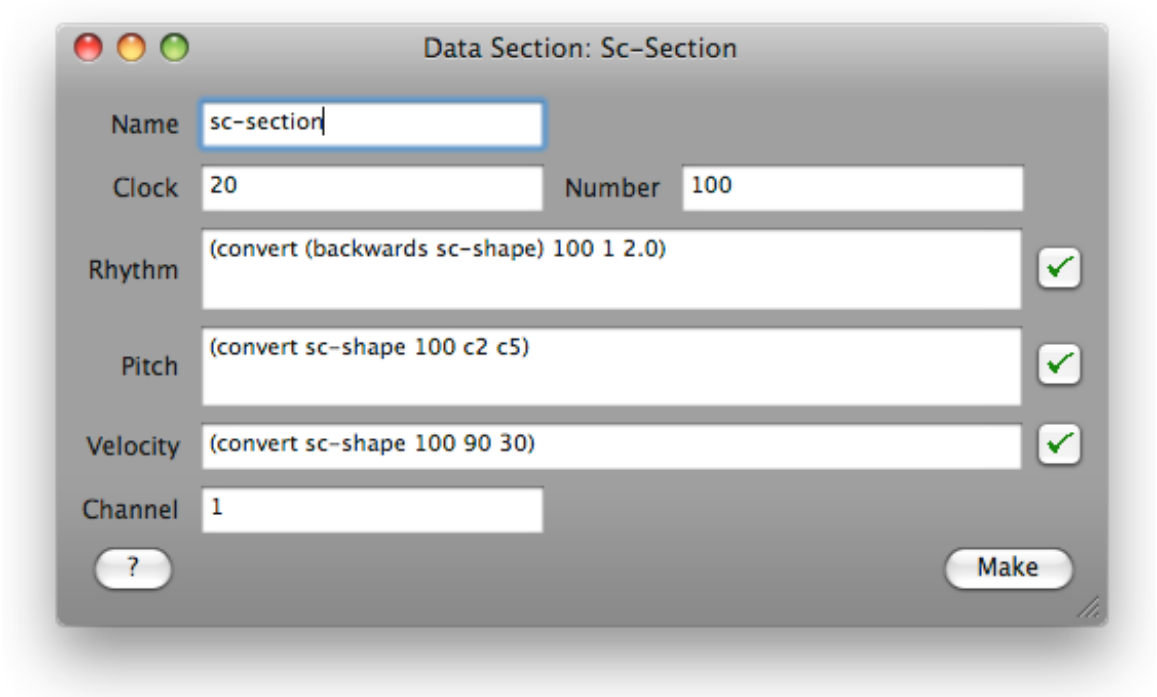
1. convenience
2. design.

The *convenience* of a scheme comes from the fact that it is an easy way to redefine several objects. If you for example generate a shape to use in a section, instead of remaking this object 'by hand' each time you want a new section, you could include it in a scheme and remake both the shape and the section by just applying the scheme. The more objects that must be redefined, the more convenient a scheme becomes.

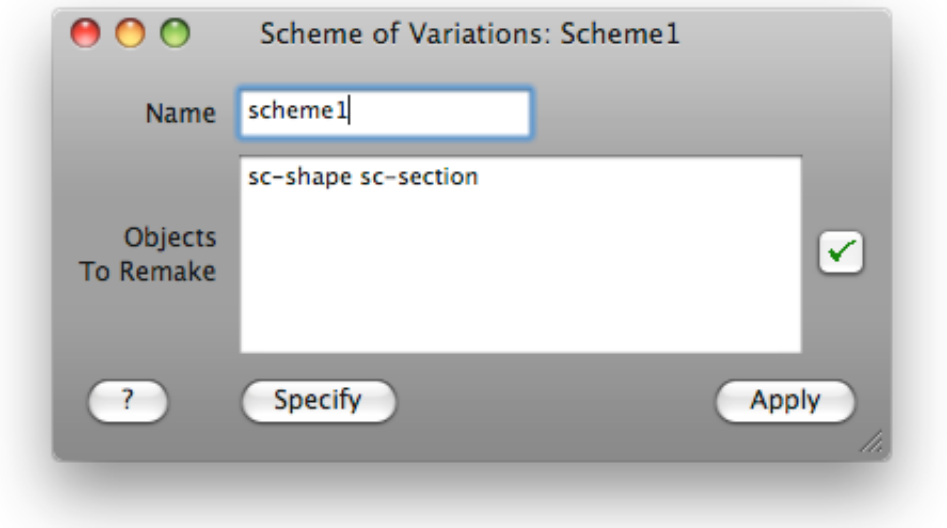
Sc-shape is a generated shape:



Sc-section uses this generated shape in various parameters:



Scheme1, when applied, will remake both *sc-shape* and *sc-section* with just one click.



A scheme also lets you use a top-down *design* strategy. When a object such as a section is *made* in the Toolbox, all other objects used in the specification must have previously been made. To make a section, all stockpiles or shapes used in the specification must have first been made. To make a sequential section, the sections must have been made, etc. This encourages an bottom-up implementation of objects.

When an object is *specified* in the Toolbox (instead of *made*), it is not necessary for the other objects used in the specification to already exist. A top-down strategy allows objects to be specified in any appropriate order and then subsequently made. The objects are specified by filling in a dialog box and clicking *Make* while holding down the Cmd key. The objects are made by including them in a *scheme of variations* and then applying that scheme. The objects should be included in the scheme in the order in which they must be made.

Available objects can be added to a scheme dialog in the *Objects* dialog. A button labeled *Scheme* will cause the object currently chosen in the table to be entered into a scheme dialog. If no scheme dialog is open, a new one is made.

Objects can also be dragged from the *Objects* dialog to the scheme dialog.

Summary

Menu items
controller, scheme

Generators
line-segment, sync

Tools
take-one

Miscellaneous
specify

Tutorial 14

Midi files

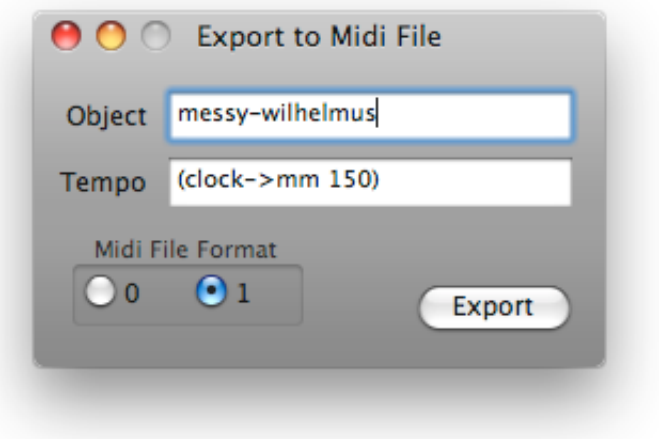
Exporting

Sections and Midi objects can be exported to a standard Midi file. This allows data prepared in the AC Toolbox to be used by other programs such as sequencers or notation programs. Either Midi file format 1 (multi-track) or Midi file format 0 (single track) can be used. If file format 1 is used, each Midi channel is written to a separate track.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 14 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

To demonstrate the exporting of a section,
• Make tutorial objects *wilhelmus-section*, *wilhelmus-sustain*, and *messy-wilhelmus*. This last section that contains both notes and Midi controller commands will be exported.

• Choose **Export Midi** in the **File** menu.



• Enter the name *messy-wilhelmus* in the dialog. Use the radio buttons to choose Midi file format 1 or format 0 for the resulting Midi file. Choose format 1. The tempo of 120 mm is the default value. Other tempo values could be entered.

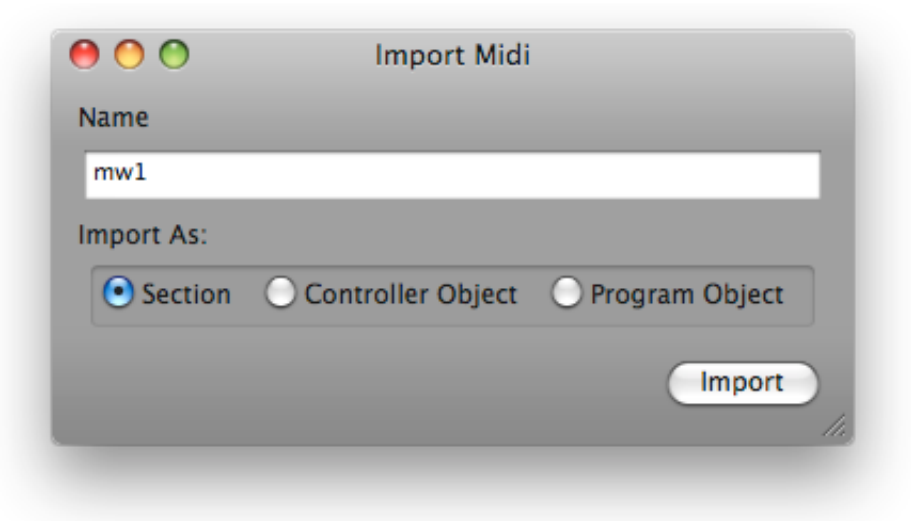
When exporting files for use in a notation program, it is often important to pay attention to the tempo. Sometimes it is useful to put in a very high tempo so that the notation program does not have to quantize too much data. To calculate a tempo based on the clock unit used in a section, use the form: *(clock->mm clock-unit)*. For the section *messy-wilhelmus* a clock-unit of 150 could be used: enter *(clock->mm 150)* in the box for tempo.

• Export section *messy-wilhelmus*.

Importing

Midi files can be imported and used as sections or Midi objects. Midi files containing notes should be imported as a section. A Midi file containing only Midi controller data could be imported as a controller object. A Midi file containing only program change information could be imported as a program object. Midi file formats 0 and 1 can be imported into the AC Toolbox.

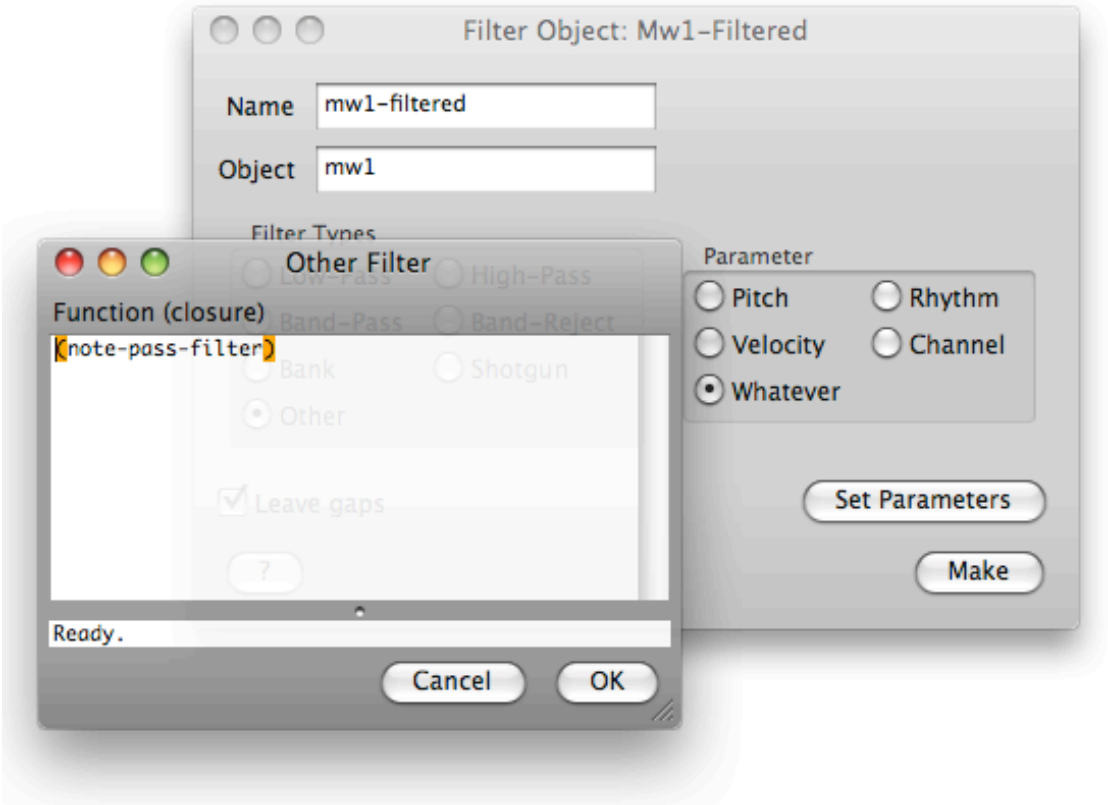
To import the Midi file made in the previous section:
• Choose **Import Midi** from the **File** menu. Enter the new name for the section being imported. Enter the name *mw1* for the object.



A section named *mw1* will be created using the notes and controller information from the previously stored Midi file. This section can be played, plotted, etc. It can be used in any way that a section defined in the Toolbox can be used.

Using imported sections
A few examples of using a section that was imported from a Midi file will be given. Section *mw1* produces both notes and Midi controller values. To only use the note values, the section should be filtered to only keep the note values.

• Examine and make object *mw1-filtered*. A filter of type *other* has been specified. The parameter is *whatever*. The form *(note-pass-filter)* produces a filter expression that will only allow notes to pass and will filter out any controller values, program changes, etc. A general discussion of the use of filters can be found in Tutorial 7.



Section *mw1-filtered* can be used as a stockpile of values from which choices can be made in a note section.

• Make section *mw1-notes*. A random choice is made among the notes found in section *mw1-filtered*.
The form is: `(random-choice mw1-filtered).`

A transition table can be derived from section *mw1-filtered*. A transition table indicates the probability of occurrence of a value. A zero-order table looks at zero previous values. Higher order tables look at one, two, etc. previous values.

• Make object *table0*. This constructs a zero-order table using the form:
`(derive-transition-table mw1-filtered 0).`
• Make object *transition1*. This is a section where notes are chosen according to the transitions specified in table0. The generator *transition* is used for this. The form is:
`(transition table0).`
• Make object *table1.* This constructs a first-order table using the form:
`(derive-transition-table mw1-filtered 1).`
• Make object *transition2*. This is a section where notes are chosen according to the transitions specified in table1.

Summary

Menu items
export Midi, import Midi

Generators
transition

Tools
derive-transition-table, note-pass-filter

Tutorial 15

Generating Csound score files

Text files, suitable for use as Csound score files, can be generated in the AC Toolbox. If you do not understand or like Csound, skip this tutorial.

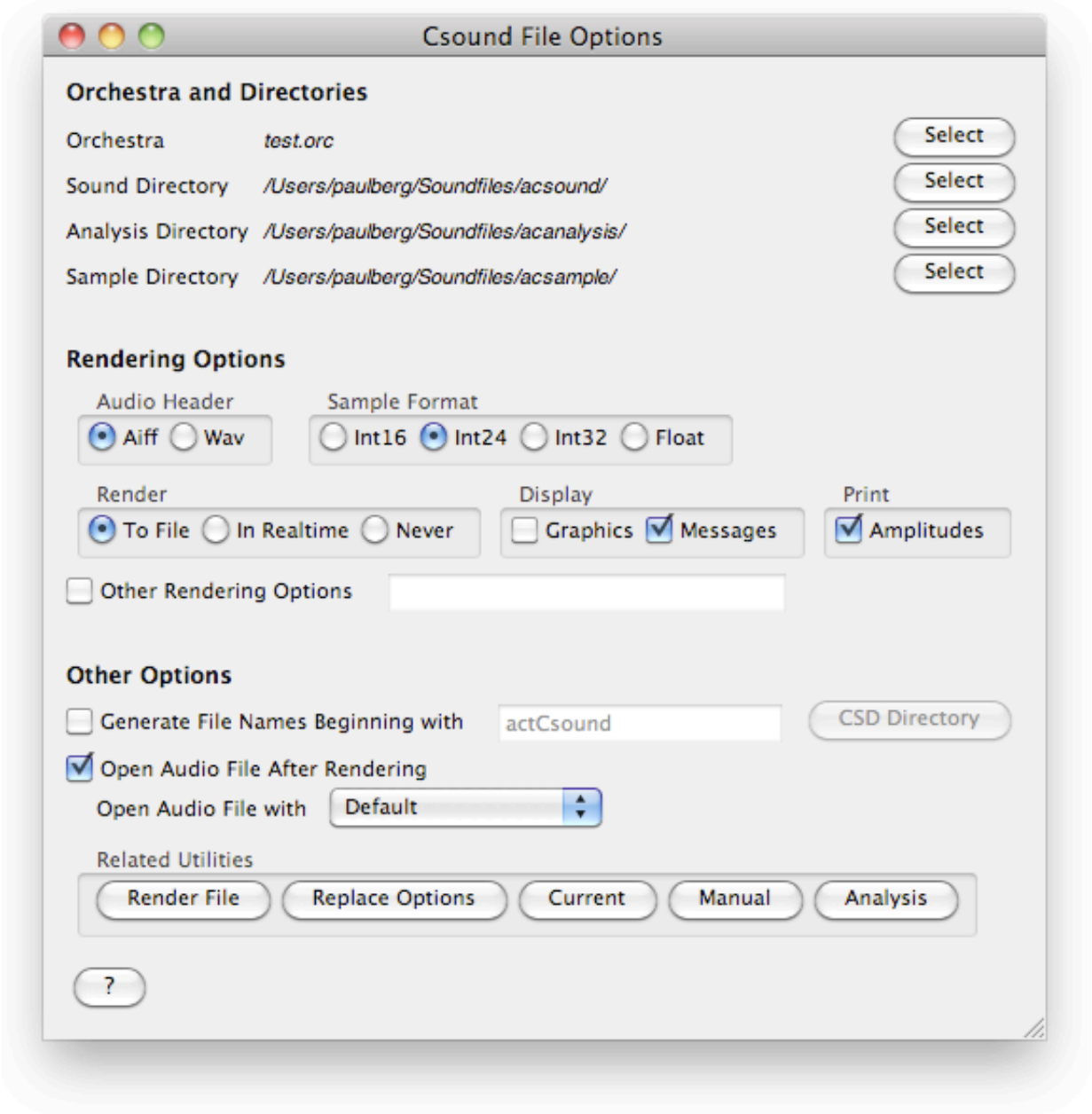
The AC Toolbox can work with *Csound objects* that contain a description of how to generate a Csound file. Csound objects are different from most other Toolbox objects because the output made by the object is not stored within the AC Toolbox environment but in a separate file. The input for a Csound object is however stored as part of an AC Toolbox environment.

This tutorial describes how to make and use Csound objects. Csound itself is not explained. It can be downloaded from http://csound.sourceforge.net/. Other information about Csound can be found at csounds.com.

The Csound score files produced by the AC Toolbox use the unified file format for Csound that includes orchestra, score, and option data in one text file with the extension .csd. These files will also be referred to as csd files.

Options
Csd files can be rendered to audio in the AC Toolbox according to options that are specified via **Other>Csound File Options**. In this dialog, matters such as sample format, audio header, and whether the rendering should happen in realtime or to an audio file can be specified. Related utilities can also be invoked via this dialog to open the Csound manual, make phase vocoder or hetrodyne filter analyses, etc. The *Current* button can be used to open the most recent csd file produced in the current session.

It is possible for file names to be automatically generated, starting with a prefix supplied in this dialog and followed by consecutive numbers, e.g. *actCsound1.csd*, *actCsound2.csd*, etc. The rendered audio file will named *actCsound1.aif* etc. if *Aiff* is chosen as the audio header. The user will be asked for a folder for writing the files. The choice of folder can be changed via *CSD Directory*. The numbering of the files begins at 1 for each session.



When an audio file is rendered, it can be opened in another application for playback. The default situation is to use the program that the system uses to open a file of that file type. For AIFF and WAV files, this is probably iTunes. It is also possible to specify another application to open the file by selecting *Choose Application* in the popup menu for *Open Audio File with* in the Csound File Options.

Render File will render (again) an existing csd file. This is particularly useful if rendering happens in realtime. *Replace Options* will render an existing csd file after it has replaced the rendering options (such as sample format) that are stored in the file with the current options in the Csound File Options. For example, this can be used to change rendering from realtime to an audio file.

Messages allows all of the Csound messages to be printed to the window Text Output. If there are error messages in the orchestra definition, they will only be reported if this option is checked. *Amplitudes* allows the user to see how far along Csound is in the rendering processes since a message is printed when something happens with an event. These messages should be printed at least until it is clear that there are no syntax errors. Csound is rather verbose which is the reason for the option to turn off all of the chatter.

The menu item **Other>Render CSD** will render the most recent csd file from the current session. It uses the rendering options in that file. If no file is available, it will open a dialog to select a file. This menu item is useful when csd files are rendered in realtime. With this item, they can be rendered again (and again). The keyboard shortcut is CMD-shift-R.

The AC Toolbox assumes that the user has installed Csound5. Select the *Index* item *Installing Csound* in the application for further information about that process.

Score object dialog
In a *Csound Score Object* dialog, functions statements should be entered in the box marked *Header*. Any other statements that are not instrument statements should also be entered in *Header*.

Instrument statements are generated by entering constants, lists, stockpiles, generators, etc. for creating the parameters in the boxes marked *Instrument*, *Start, Duration, P4*, *P5*, and so on. If more than 10 *P* fields are needed, expressions to produce the remaining P fields (such as P11, P12,...) can be included in the box *Extra*. The statements in *Extra* should be separated by a space or a newline.

In the score file, one or more layers can be produced. Layers are parallel combinations of outputs.

The number of instrument statements to be produced in a layer should be entered in *Number*.

Instrument should be expressed as an integer or some expression (generator, etc.) that will produce an integer. The instrument number should correspond to some instrument number in the orchestra file you want to use.

The start time (P2) for each line in the score file is cumulative, starting at the time indicated in *Start*, when *Start* is a constant. The value for each subsequent P2 is the previous value plus the duration of the previous event. Later in this tutorial another way of dealing with start times will be discussed.

Duration (P3) is expressed in beats. By default, a beat is one second. Negative values cause a rest equal to the absolute value of the duration. No score line is written for a rest and the values for other parameters are discarded.

The meaning of parameters *P4* through *P10* and any *Extra* parameters depends on the instrument definition. These boxes can be left empty if the parameters are not needed.

Per Layer could contain the name of objects that should be remade each time a layer of the score is generated. This is similar to the *Object(s) To Remake* box when using *Generate Parallel Section*.

Per Score could contain the names of objects that should be remade each time the score object is applied to generate a score file. *Per Score* is remade once before the score is applied. *Per Layer* is remade before each layer is made.

The boxes for *Number*, *Start, Instrument*, and *Duration* must be filled. The use of any other boxes is optional.

When a Csound object is specified (by selecting the *Specify* button), the definition of the object is stored but the score file is not made. When a Csound object is applied (by selecting the *Apply* button), the definition of the object is stored and this definition is used to create a Csound file. Depending on the choices made in **Csound File Options**, the file may be rendered to audio after *Apply* has been selected.

Examples

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 15 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make a score file from the object, click on *Apply* in the dialog box.

All of the examples can be rendered with the orchestra file *test csound.orc* that is found in the folder *Support/FileExamples.* Instrument 1 in that orchestra file is a simple oscillator with an envelope. In all of the examples, a sine function is assigned to the oscillator. Besides *Instrument*, *Start*, and *Duration*, the parameter values in the score for this instrument are:

P4 amplitude (between 0 - 1)
P5 frequency (in Hz)

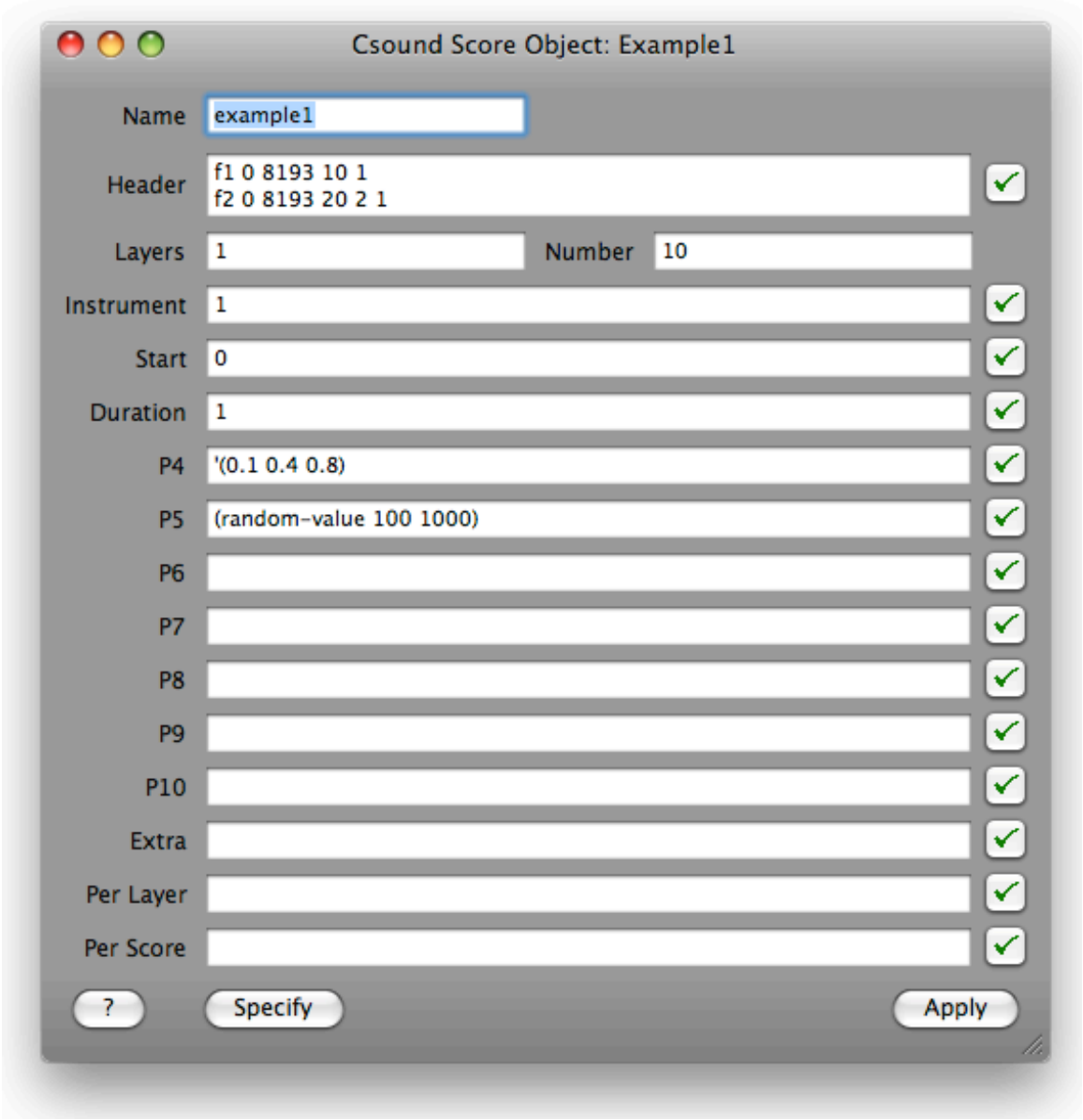
Note that in other instrument definitions, *P4* and *P5* could be used for parameters other than amplitude and frequency. In the examples in this tutorial, *P4* and *P5* are always amplitude and frequency.

The definition for instrument 1 is:

```
        instr 1
kctrl   oscil1i 0,p4,p3,2
asig    oscili kctrl,p5,1
        out asig
         endin
```

• Select *test csound.orc* as the orchestra in the *Csound File Options* dialog.
• Select a folder for the sound directory. Because of a feature of Csound, the path to the sound file should not include any spaces.

Example 1



Header contains the function statements. F1 is the waveform table for the oscillator. In this case it is a sine. F2 is a table to use as an envelope. Here it is a Hanning window. (If several objects are going to be made with the same header, it may be convenient to drag the header to the desktop and drag it back to the new header box when defining another Csound object.)

The first example produces a score with 10 events for instrument 1. The first event is at time 0. Each subsequent event starts when the previous one ends. Each event has a duration of 1 second. P4 is for amplitude. The first event has an amplitude value of 0.1, the second event 0.4, the third 0.8, the fourth 0.1, etc. The total amplitude at any given moment should probably be <= 1. P5 is for frequency. The generator will produce a random value between 100 and 1000 Hz. Since the instrument only uses 5 parameters, the remaining boxes in the dialog are left empty.

• *Apply* the object. The score produced can be viewed in the AC Toolbox (choose the *Current* button in the Csound File Options or select the file with **File>Text>Open**). A number of markup strings will be seen. Somewhere in that mess will be something similar to:

```
; This score was generated from an AC Toolbox Csound score object.

; Object:   Example1
; Type:     Csound-score-object


;;;;;;;;;;;

; Header:

f1 0 8193 10 1
f2 0 8193 20 2 1


;;;;;;;;;;;


i1 0 1 0.1000 107
i1 1 1 0.4000 118
i1 2 1 0.8000 113
i1 3 1 0.1000 628
i1 4 1 0.4000 544
i1 5 1 0.8000 779
i1 6 1 0.1000 710
i1 7 1 0.4000 203
i1 8 1 0.8000 808
i1 9 1 0.1000 881
```

Statements preceded by a semicolon are comments and ignored by Csound when a sound file is made. The comments are followed by the statements included in *Header*. These statements create the two function tables. The instrument statements follow. Each line contains five parameter values:
instrument, start time, duration, amplitude, and frequency.
Ten events were generated.

When this score file is rendered with *test csound.orc* in Csound, ten sine tones with the specified frequencies will be written to a sound file or played in realtime.

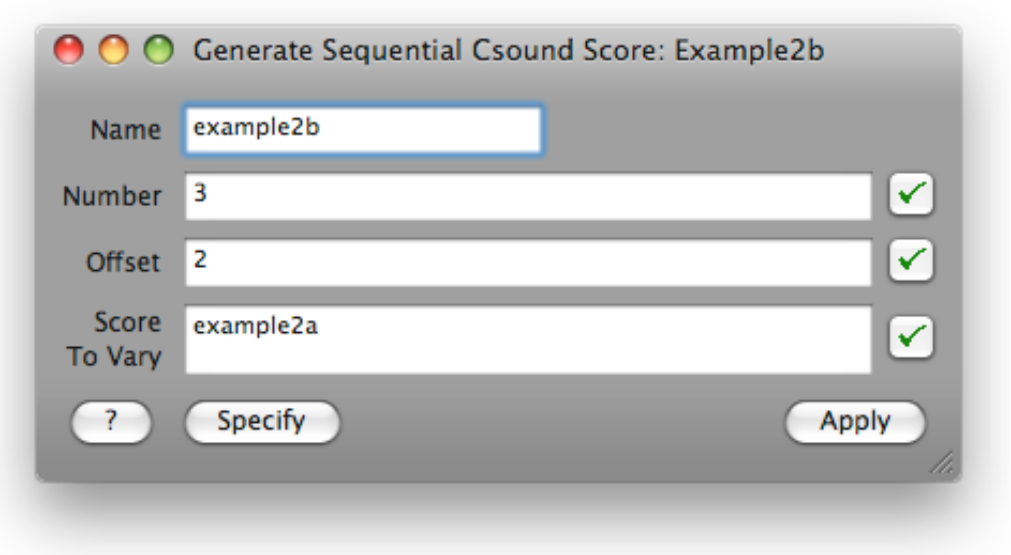
Generating variants of a score
Variants of a score can be generated and gathered in sequence with **Define>Csound Objects>Generate Sequential Scores**. The dialog is similar to the one for generating a sequential section. The name of a score can be entered (or dragged from the Objects dialog) and the results of applying this Csound object several times will be gathered into one file. *Offset* is an amount time in seconds that can be inserted between the variants. It can be a constant, generator, etc.

Example2a is a Csound Score Object with a random walk for the frequency parameter (P5).

The input for *example2a* is:

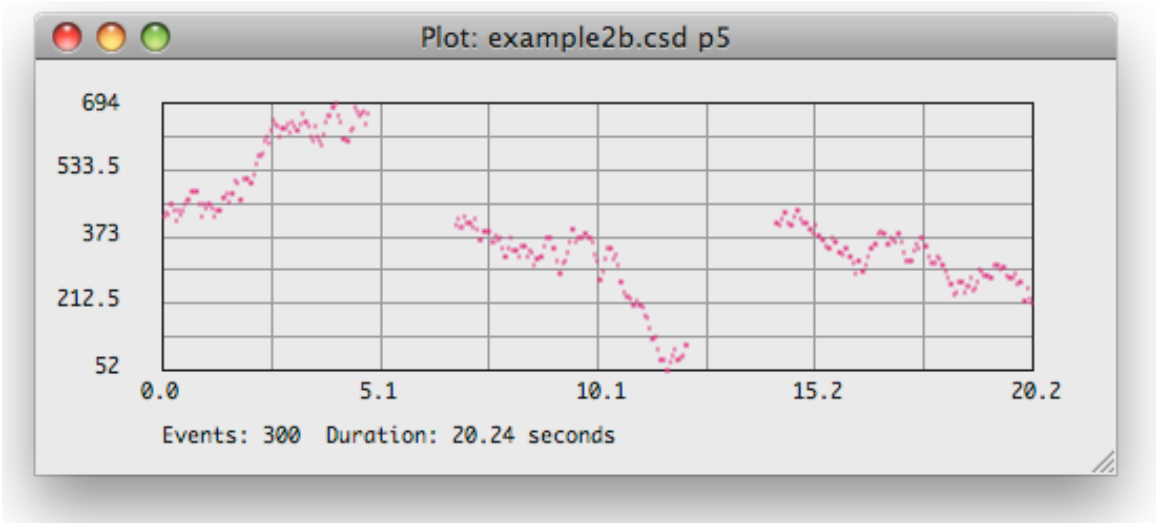
```
Layers:      1
Number:    100
Instrument: 1
Start: 0
Duration: (random-value 0.01 0.1)
P4:  (random-value 0.1 0.2)
P5:  (walk 400 (random-value -30 30) 50 1000)
```

Example2b will generate three variants of example2a with a delay of two seconds between each variant. Frequency is determined by a random walk between 50 and 1000 Hz. The walk begins with 400 Hz.



Each variant evaluates the input specification. If the input is a generator, a new closure will be made for each section. This means that each variant will begin with a frequency of 400 Hz and start walking from there.

A possible output produced by *example2b*, plotting parameter *P5* (frequency) over time:



To plot a parameter of a Csound score, use **Tools>Plot>Csound File.** Remember to plot the score file and not the audio file.

More examples of combining or varying Csound objects can be found later in this tutorial.

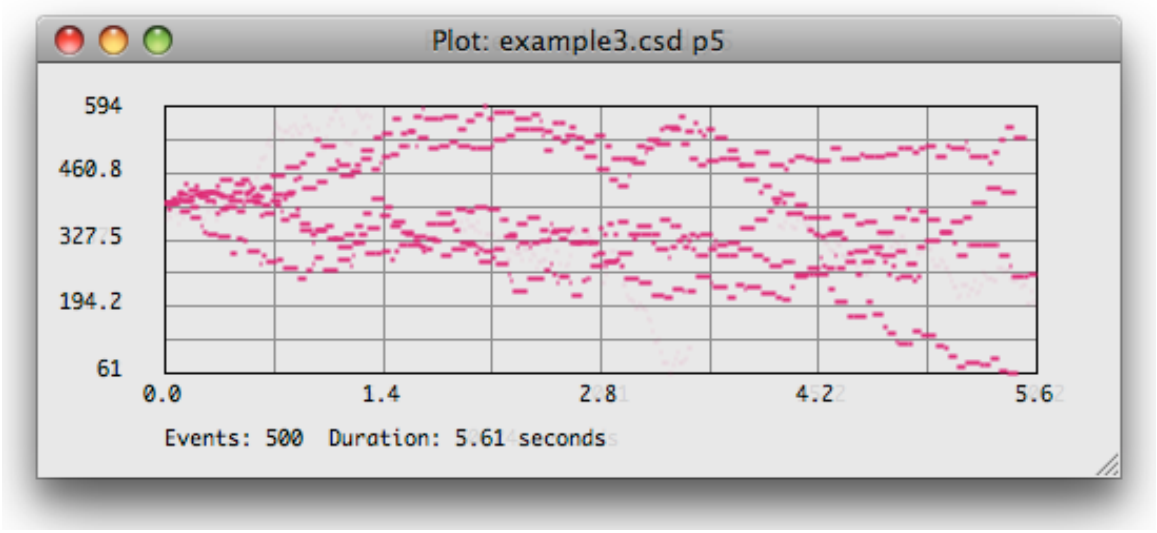
Layers
Layers are the parallel combination of generated scores. Attention should be paid to the sum of the amplitudes in all layers.

The input for *example3a* is:

```
Layers:      5
Number:    100
Instrument: 1
Start: 0
Duration: (random-value 0.01 0.1)
P4: (random-value 0.1 0.2)
P5:  (walk 400 (random-value -30 30) 50 1000)
```

Five layers of 100 events are generated. Each layer will begin at time 0. The frequency values in each layer will start with 400 Hz. Amplitude values (*P4*) for each layer do not exceed 0.2 therefore five layers will stay within the maximum of 1.

A plot of *P5* (frequency) over time:



Another way of dealing with the amplitude issue is to use the generator *scale-by-layers*. The input to this generator can be another generator, etc. The value is scaled by the number of layers. In *example3b*, *P4* uses scale-by-layers. The larger range for random-value is used since the results will be scaled anyway. *Scale-by*-layers has an optional *factor* parameter to scale by a fraction of the number of layers. This option is explained later in this tutorial. *Scale-by-layers* can be abbreviated as *sbl*.

```
Layers:      5
Number:    100
Instrument: 1
Start: 0
Duration: (random-value 0.01 0.1)
P4: (scale-by-layers (random-value 0.5 0.9))
P5:  (walk 400 (random-value -30 30) 50 1000)
```

With a mask
Masks and other objects can be used when making Csound objects. Example 4 uses the object *mask1* to determine the frequencies in the score.

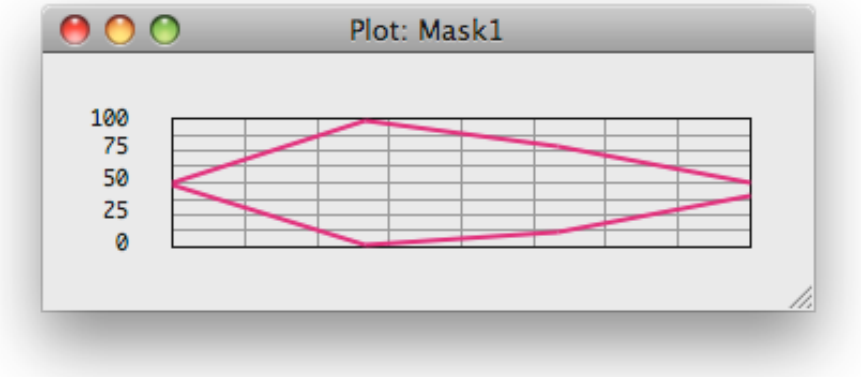
The input for *example4* is:

```
Layers:      1
Number:    100
Instrument: 1
Start: 0
Duration: (random-value 0.01 0.1)
P4:  (random-value 0.1 0.2)
P5:  (convert mask1 100 50 500)
```

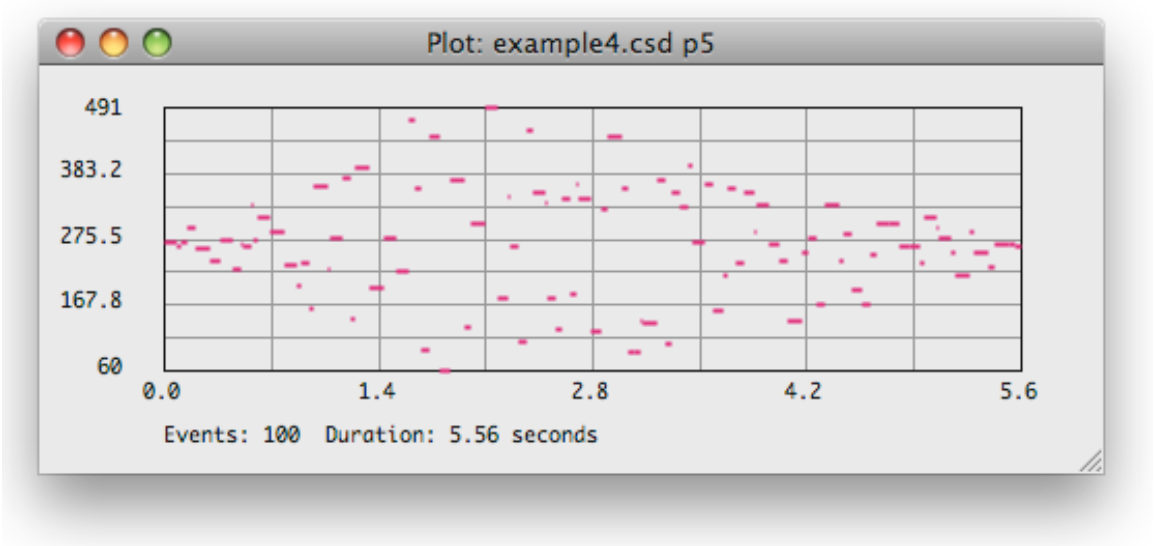
C*onvert* is used in *P5* to convert *mask1* to a list of values chosen between the mask boundaries scaled to 50 and 500.

• *Make mask1* before applying *example4.*

Mask1:



A plot of *P5* (frequency) over time:



If a generator were used for *Start*, the start times would have to be sorted so that the mask would be used in the correct order. See the section Sorting start times for a discussion.

Using a generator for start
When the value for *Start* is a constant, all start times in a layer are cumulative, i.e. one event starts when the previous one is finished. If *Start* is a list, generator, etc. all start times will be derived from the input. This means that if *Start* is produced by a generator, the generator will be used to generate all of the start times. If the generator is *(random-value 0.0 5)*, start-times will be produced between 0 and 5 beats. Note the difference between using integers and real numbers with random-value. If integers are used, all events will fall on integer values between 0 and 5. If one or more real numbers are used, as in this example, the values can occur anywhere between 0.0 and 5.

The input for *example5* is:

```
Layers:      1
Number:     10
Instrument: 1
Start: (random-value 0.0 5)
Duration: (random-value 3.0 5)
P4:  (random-value 0.1 0.2)
P5:  (random-value 100 500)
```

The instrument statements could be similar to these:

```
i1 4.3549 3.5466 0.1807 204
i1 3.9190 3.4125 0.1643 378
i1 1.2035 3.6343 0.1223 231
i1 1.2575 4.6906 0.1924 139
```

```
i1 1.6502 4.3505 0.1443 450
i1 2.8743 3.9574 0.1402 489
i1 0.9812 4.2516 0.1377 176
i1 1.6069 3.5147 0.1452 221
i1 2.4201 4.5417 0.1075 255
i1 3.0701 3.4949 0.1271 151
```

Sorting start times
In the previous example, the start times were not sorted. Random-value does not produce sorted output. Csound will accept statements in any order and sort them according to their start times. When start times are not sorted in the score, the order of the events will be different than the order in which they were calculated. If parameters are chosen to reflect some statistical distribution in a period of time, this is probably not a problem. If parameters are to reflect the movement of a shape or mask, this is a problem.

To generate a number of start times and then sort them in ascending order, use *make&sort*. The parameters are *N*, the number of values to be generated, and *generator*, the generator that should be used to generate the values.

E.g., *(make&sort 10 (random-value 0.0 5))* could return the following list:

```
(0.015377244009407692 1.4057785298782932 1.5438746224043418
1.8198886920137118 1.8398871213940662 1.904497690320942
  3.68248114116952 3.7313657882276283 3.9149525089467923
4.6053417299042305)
```

The input for *example6* is:

```
Layers:      1
Number:    100
Instrument: 1
Start: (make&sort 100 (random-value 0.0 20))
Duration: (random-value 3.0 5)
P4:  (random-value 0.05 0.1)
P5:  (convert mask1 100 100 500)
```

An example of some possible instruments statements (note that the start times in the second column are sorted in ascending order):

```
i1 0.0057 3.0628 0.0952 297
i1 0.2068 4.9022 0.0875 304
i1 0.3266 3.3038 0.0662 287
i1 0.9083 3.3018 0.0611 313
i1 1.3932 3.7385 0.0815 298
i1 1.6050 3.1588 0.0506 277
i1 1.7647 3.5036 0.0757 283
i1 1.8993 3.7184 0.0902 274
i1 1.9192 4.6229 0.0521 271
i1 1.9956 3.7857 0.0920 263
...
```

Make&sort can be abbreviated as *ms*.

Filling an amount of time
Until-time can be used to determine the number of events to be calculated in the score. *Until-time* indicates that events should be generated until a certain length of time in seconds has been reached. It can only be used if *Start* is a constant value.

For parameter *Number*, *(until-time 10)* could be entered. This requests events to be generated until ten seconds have been filled. The actual length of the score could be longer than ten seconds since the last event is allowed to finish instead of being cut off at ten seconds.

If another expression needs to know how many events were generated with *until-time*, *(from-number)* can be used to get the number.

The input for *example7* is:

```
Layers:      5
Number:(until-time 30)
Instrument: 1
Start: 0
Duration: (random-value  3.0 5)
P4:  (random-value  0.01 0.05)
P5:  (convert mask1 (from-number) 100  500)
```

Events will be generated until 30 seconds have been filled. Frequency (P5) is determined by converting a mask. The number of frequency values is specified by using *from-number*. Five layers have been requested. Each layer will last 30 seconds.

Possible instrument statements for one layer:

```
i1 0 3.7406 0.0166 299
i1 3.7406 4.4027 0.0237 278
i1 8.1434 3.0802 0.0317 424
i1 11.2235 3.7541 0.0298 187
i1 14.9777 3.4866 0.0306 221
i1 18.4643 4.4983 0.0337 400
i1 22.9626 4.2162 0.0461 280
i1 27.1788 3.5665 0.0213 295
```

Until-time can be abbreviated as *ut. From-number* can be abbreviated as *fn.*

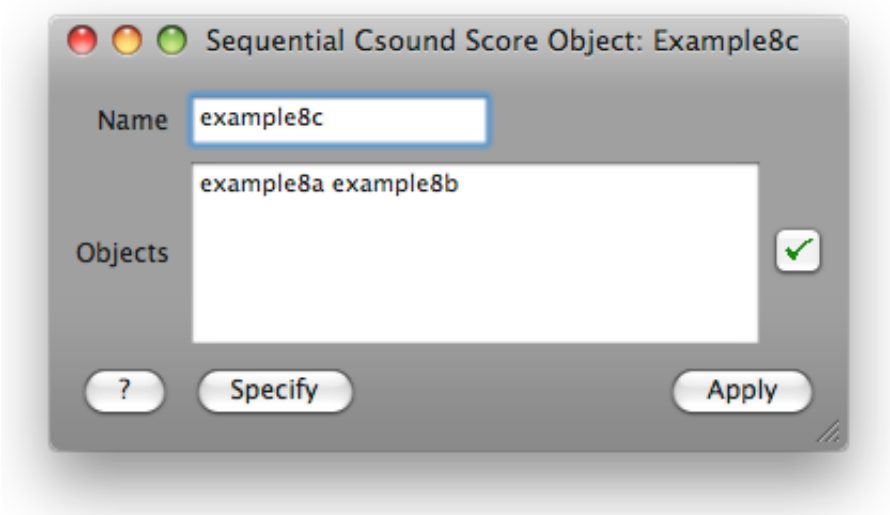
Combining score objects

Csound objects can be combined in sequence and in parallel just as is possible with sections. This can be done using a *Sequential Score* or a *Parallel Score*. The objects may use different instruments with different numbers of parameters. All instruments for one sequential or parallel section should be in the same orchestra file.

Object names can be entered by typing them or dragging from the Objects dialog.

Care should be taken when specifying function numbers. If each score being combined uses the same function number, the last definition is the one used.

Example8a and *example8b* use different frequency boundaries for *P5*. *Example8c* joins these two objects into one score. Note that it is not necessary to apply *example8a* or *example8b* before applying *example8c*. It is sufficient that the examples used in a sequential score have been *specified*.

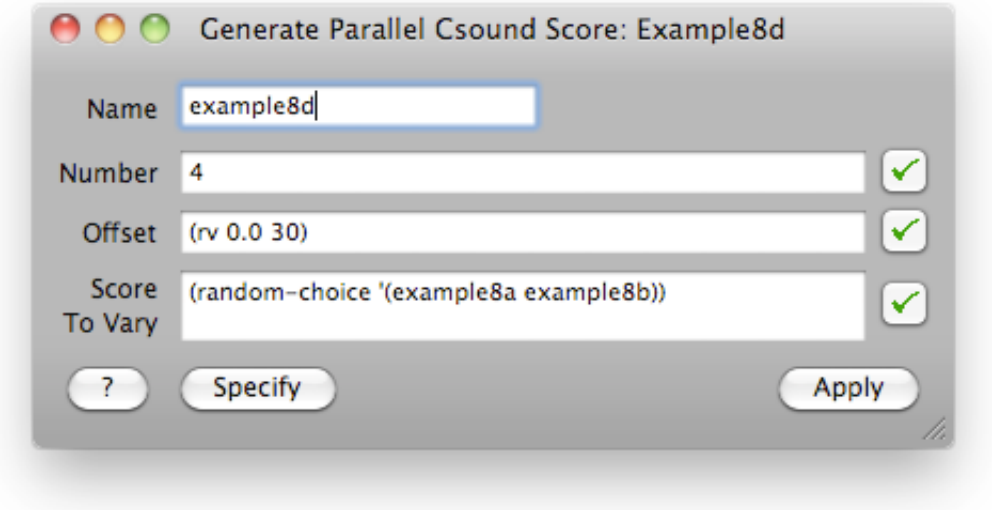


It is also possible to indicate a time delay in seconds between the scores. The input to *example8c* could specify a two second delay between the generated scores as follows:

```
example8a 2 example8b
```

When combining scores in parallel, attention should be paid to the amplitudes.

A sequential or parallel score can be generated, e.g. **Define>Csound Object>Generate Parallel Score**. One or more existing scores is varied and the result is written to one file. *Example8d* generates a score with four variants. For each variant, a choice is made between *example8a* and *example8b*. The start times for the variants are controlled by the offset. This is the time from the beginning. The first variant will start at time 0. Each succeeding variant will start somewhere within the first 30 seconds.



Converting Midi note numbers to frequency
Midi->hz is a generator that will convert a Midi note number to a frequency value in Hertz. The input can be a constant value, a list, a generator, etc.

The input for *example9* is:

```
Layers:      1
Number:    100
Instrument: 1
Start: 0
Duration: (random-value  0.01  0.1)
P4:  (random-value  0.1 0.2)
P5:  (midi->hz (walk 60 (random-value  -3 3) 30 90))
```

P5 is the frequency parameter. It uses *Midi->hz* to convert Midi note numbers in the range 30 to 90 to frequency values. This example is similar to one layer of example 3 except that the frequency values correspond to values from a chromatic scale. In example 3, the random walk uses arbitrary frequency intervals.

To convert the other way around, from Hz to Midi, use generator *hz->midi*. A table of Midi note numbers and the corresponding frequency values can be made using *show-midi->hz*. With this tool, a reference frequency for a4 (note 69) can be specified.

Controlling density with a shape
Density-of-start-times makes a list of start times by mapping a shape to represent varying densities between a minimum and maximum density. The total length of the score must be specified. The density is mapped according to the shape for some unit of density, such as per second. That default value for that unit is one second.

The input for *example10a* is:

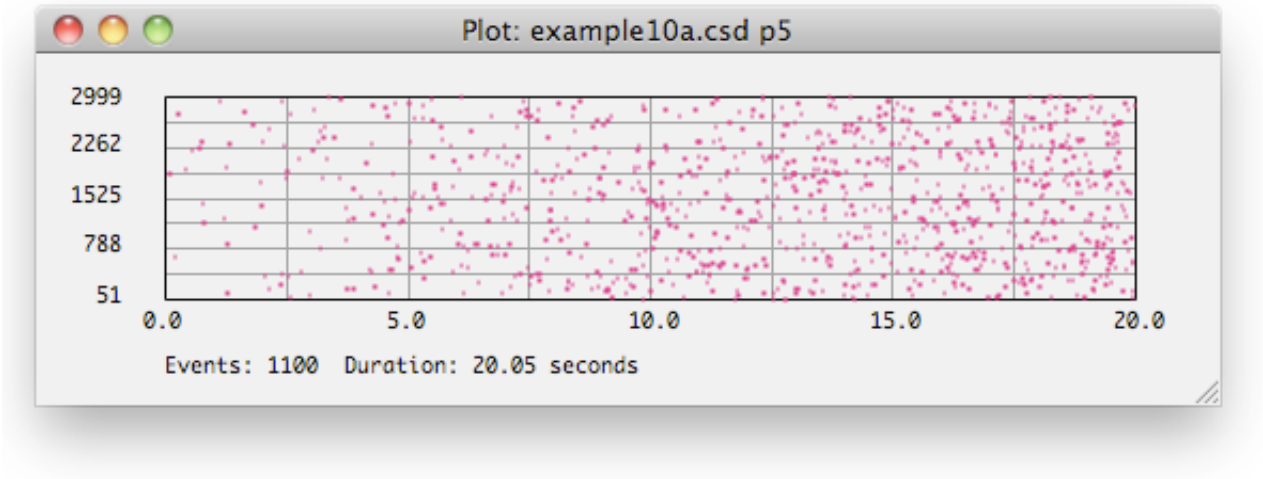
```
Layers:      1
Number:(from-start-times)
Instrument: 1
Start: (density-of-start-times  20  line-shape  10 100)
Duration: (random-value  0.01  0.1)
P4:  (random-value  0.01 0.05)
P5:  (random-value  50 3000)
```

Since the number of values produced by *density-of-start-times* is not known prior to the calculation, the input value for the *Number* parameter should be *(from-start-times)* that

sets the number of events for the score file to be equal to the total number of events generated by the *density-of-start-times* expression.

In *example10a*, 20 seconds are filled. The density increases linearly (following line-shape) between 10 events per second to 100 events per second. Consult the help for *density-of-start-times* in the AC Toolbox for additional information about this tool.

A plot of *P5* (frequency) over time shows the increasing density:



Density-of-start-times can be abbreviated as *density*. The abbreviation *fst* can be used for *from-start-times*. These abbreviations are used in example10b.

Density-shape is a shape that has been generated. It contains 5 points, randomly chosen in the range 1 to 100. The actual range does not matter since the result will be mapped to whatever the density parameters are.

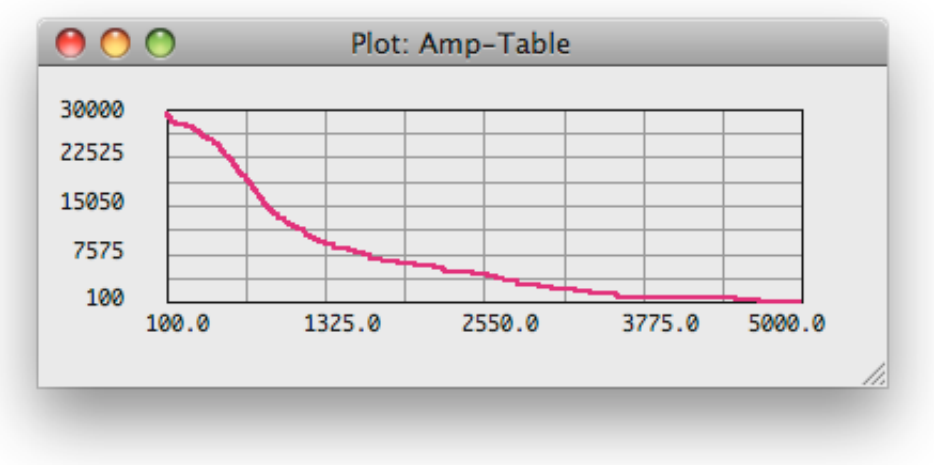
The input for *example10b* is:

```
Layers:      1
Number:(fst)
Instrument: 1
Start: (density 20  density-shape  2 200)
Duration: (random-value  0.01  0.1)
P4:  (random-value  0.01 0.05)
P5:  (random-value  500 2000)
```

Relating two parameters to each other
Example11a uses a different amplitude value (*P4*) depending on the frequency value.

An amplitude curve is drawn: *amp-shape*.

A lookup table is made to associate the values of this shape with frequency values in the possible range of 100-5000 Hz: *amp-table*.



The input to make amp-table is:

```
(convert-to-lookup-table amp-shape 100 5000 0.005 0.8)
```

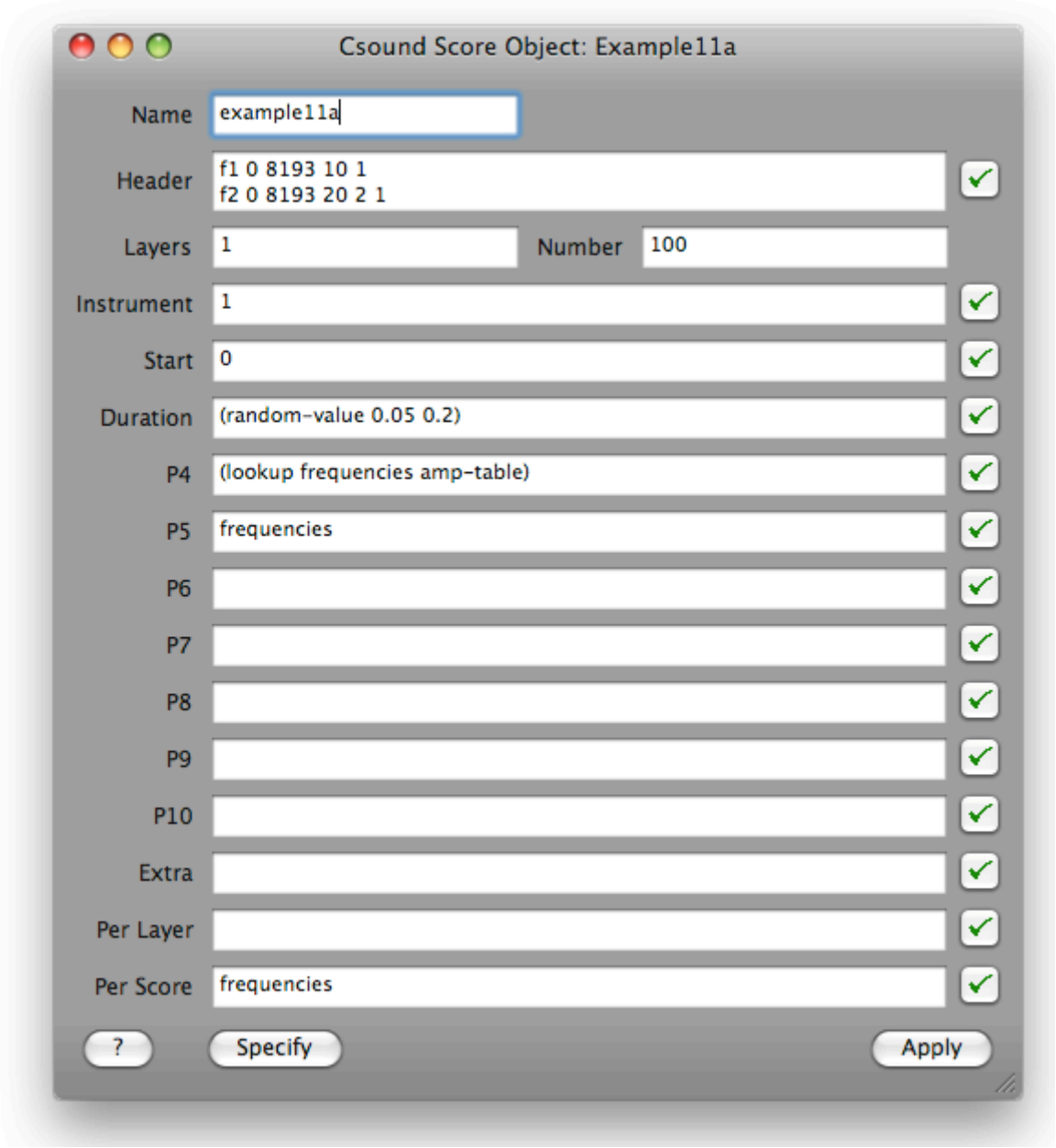
This will create a lookup table with values for frequency between 100-5000 Hz. The associated amplitude values will be 0.005 - 0.8 following the shape of *amp-shape*. The highest value in the shape will correspond to 0.8 and the lowest position with be 0.005.

With the *amp-table* pictured above, low frequencies will have higher amplitudes. The generator *lookup* will access a frequency value in this table and return the corresponding amplitude value. More information on how a lookup table works can be found in the help for *lookup* in the application. *Lookup* can interpolate between values in the table. *Make-lookup-table* can also be used to construct a table.

Values for frequency (*P5*) are generated and saved in a stockpile: *frequencies*. The generator for this stockpile will tend to create low values near 100 Hz and high values near 5000 Hz.

Since these values are defined before the score is applied, they will be available to both *P4* and *P5*. *P5* uses the stockpile for the series of values. *P4* uses the stockpile to scale those same frequency values to appropriate amplitude values.

The disadvantage of this scheme is that the stockpile *frequencies* must be made before the score object can be applied. Unless the stockpile is remade each time the score is applied, the same series of frequencies will be used. This problem can be solved by including the name of the frequency stockpile in the *Per Score* box. Each time the score object is applied, the objects in this box are remade first before the file is made.



Example11b is similar except 10 layers are produced. Two changes must be made to the score object in addition to increasing the value for layers.

First, the amplitudes should be scaled to deal with the increased number of layers:

```
(scale-by-layers (lookup frequencies amp-table) 0.7)
```

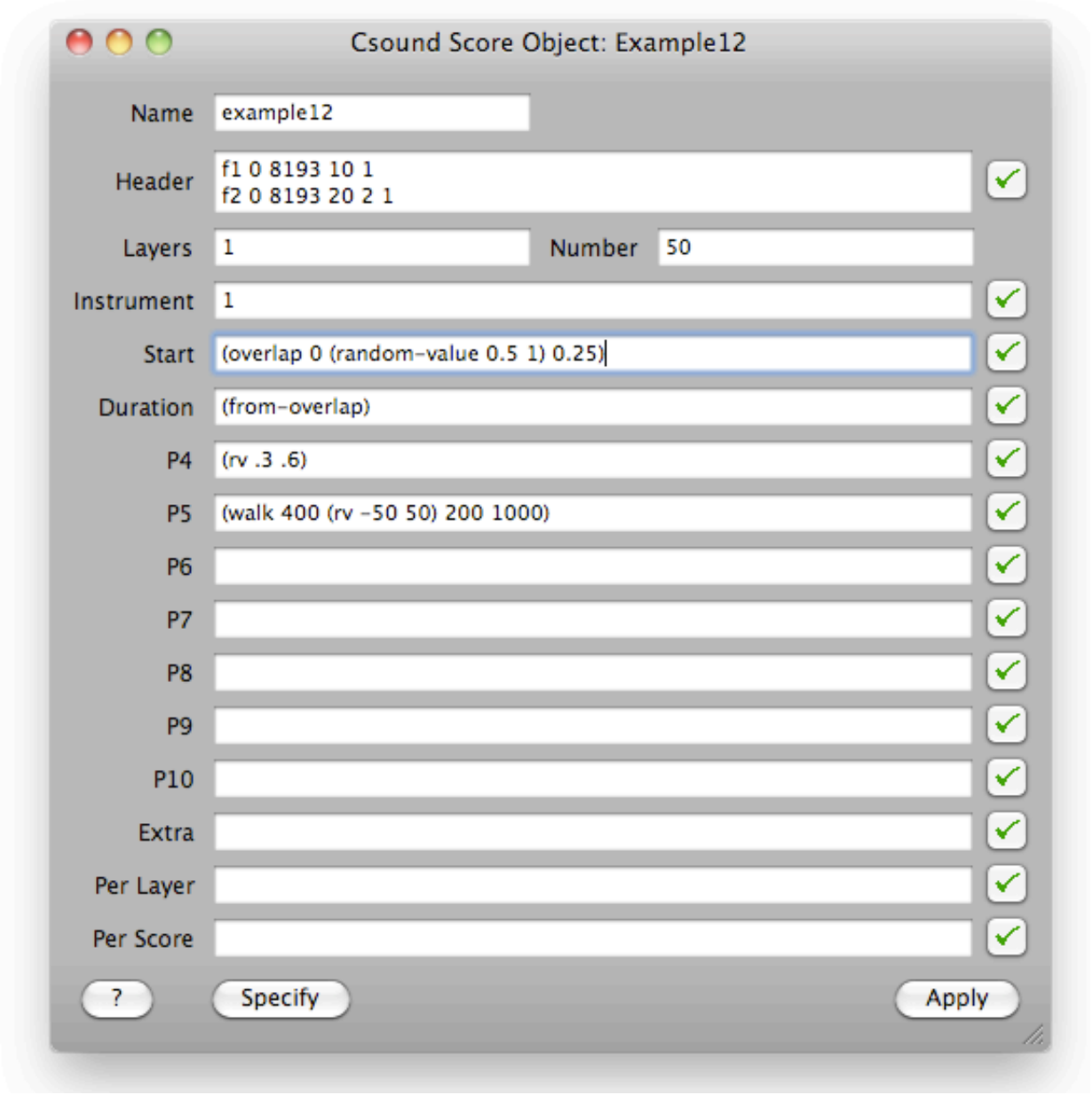
The additional parameter value (0.7) is an optional scaling factor for the number of layers. When many layers are specified, it is usually not necessary to divide the amplitudes by the number of layers. A smaller value will usually suffice. If the division should be by 70 % of the number of layers, then 0.7 can be used.

Secondly, each layer will need a new stockpile of frequencies. Therefore the stockpile should be remade *per layer* instead of *per score*.

Other useful generators for relating parameters include *par*, *act-if*, and *on-the-fly*. *Par* can access the current value of a lower numbered p-field. P5 could refer to P4, P3, P2, or P1. *Act-if* can use a stockpile or parameter value as part of a condition to determine which of a series of generators or values to use each time it is applied. O*n-the-fly* simplifies arithmetic operations using stockpiles and generators. See the help texts in the application for further explanation.

Overlap
If start times should overlap, the generator *overlap* can be used. The amount of the overlap can be specified as an absolute time value or as a percentage of the duration.

Example12 uses an overlap of 250 milliseconds (0.25 seconds). This means that a new event starts 250 milliseconds before the previous one was finished. Note that *overlap* is used in the box for *Start* and *(from-overlap)* is used for *Duration*.



An overlap of 10 percent is expressed as follows:

```
(overlap 0 (random-value 0.1 0.2) 10 :percent t)
```

Identifying layers
If a file has several layers, it is possible for each layer to have a different behavior. For example, each layer could use different boundaries for picking values. Tool *layer-number* returns the number of the current layer while a file is being generated. This information can be used in the Csound score object.

Read-from can read from a list or stockpile using an index.

```
(read-from '(10 20 30) 1)
```

would return the first value in the list: 10.

Layer-number can be used as the index for reading from a list.

Example13a reads one frequency value for each layer, based on the layer number.

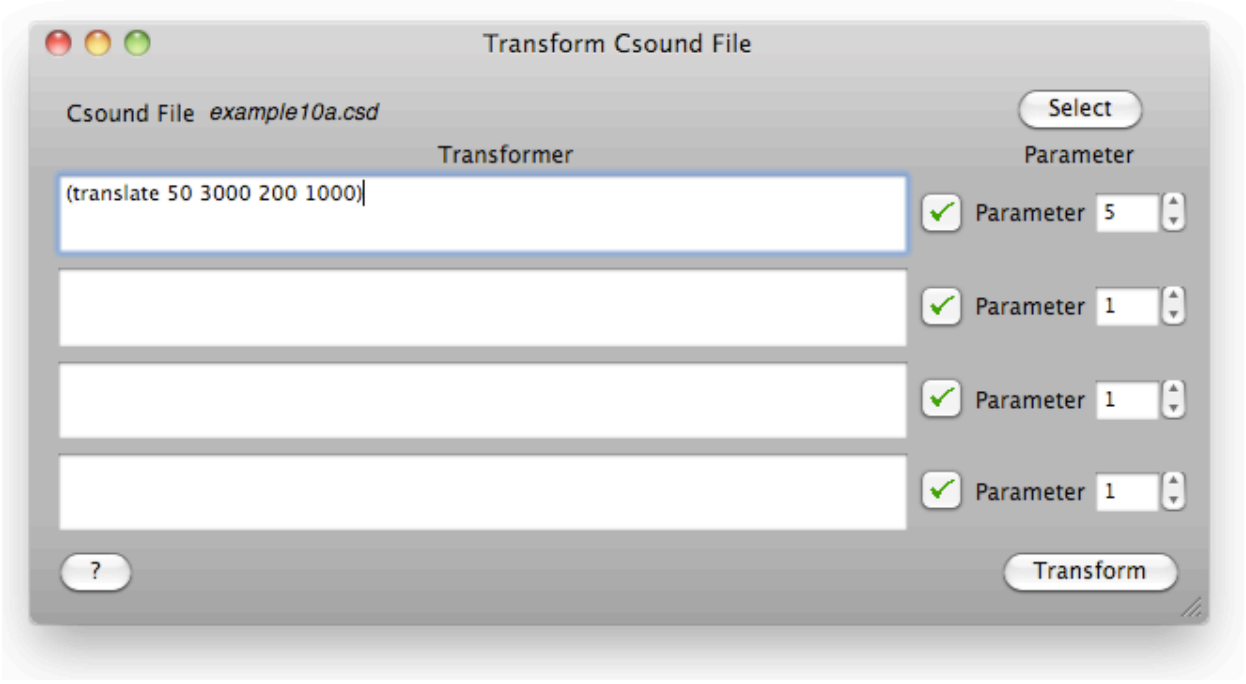
```
Layers:      5
Number:    100
Instrument: 1
Start: (make&sort 100 (random-value 0.0 10))
Duration: (random-value 1.0 3)
P4: (scale-by-layers (random-value 0.01 0.05))
P5: (read-from '(300 750 1700 2760 4000) (layer-number))
```

Example13b picks random frequencies for each layer. The boundaries for the random choices vary per layer.

```
Layers:      5
Number:    100
Instrument: 1
Start: (make&sort 100 (random-value 0.0 10))
Duration: (random-value 1.0 3)
P4: (scale-by-layers (random-value 0.01 0.05))
P5: (rv (read-from '(300 750 1700 2760 4000) (layer-number))
        (read-from '(350.0 800.0 2000.0 3300.0 6000.0) (layer-number)))
```

The first layer will choose values between 300 and 350 Hz. The second will choose between 750 and 800, etc.

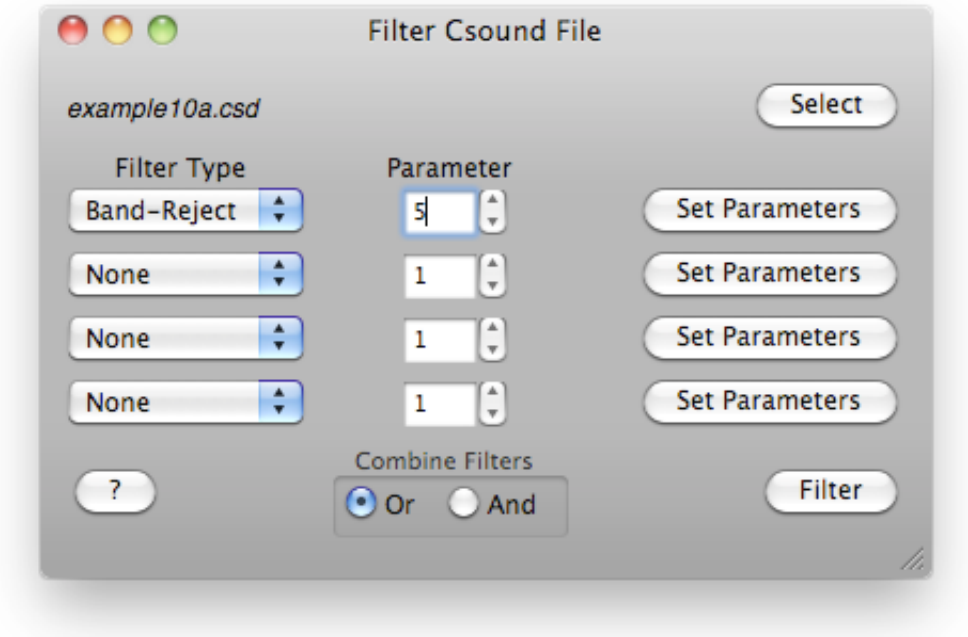
Transforming files
CSD files can be transformed (**Methods>Transform>Csound File)**.



The above dialog will make a new csd file which is a copy of file *example10a.csd*. Parameter P5 will be changed. *Translate* will translate frequency values in the range 50-3000 (the original range of the object) to be in the range 200-1000.

Up to four parameter fields can be transformed at one time. More information can be found in the *Index* item *Csound Files: Transforming* in the application.

Filtering files
CSD files can be filtered (**Methods>Filter>Csound File**).

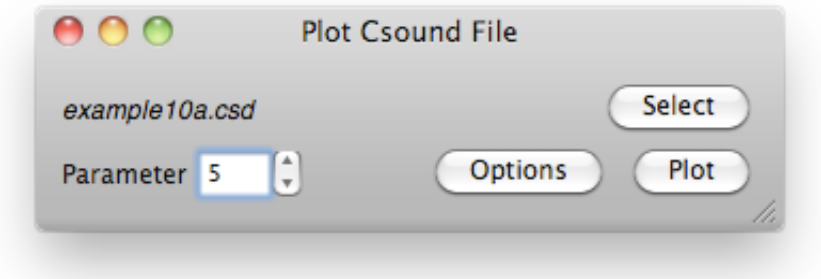


The above dialog will make a new csd file which is a filtered copy of file *example10a.csd*. If the parameters for the band-reject filter are set to 1000 and 2000, events within that range are rejected (and not copied to the new score file). One or more filters can be applied and the results combined with a logical AND or a logical OR. In the first case, an event must pass all the filter tests. In the second case, an event only needs to pass one of the tests. More information can be found in the *Index* item *Csound Files: Filtering* in the application.

Miscellaneous
If an orchestra definition needs the name of a sound or analysis file, *(sf-name "filename")* should be used instead of entering *"filename"* in the dialog box. To use a generator to pick a sound file name, use *generate-sf-name*. If the file is a sound file, it should be in the sample directory specified in the **Csound File Options** dialog. If it is an analysis file, it should be in the analysis directory. The help for sf-name and generate-sf-name in the application give examples of their use. The shortcut for *sf-name* is *sf*. The shortcut for *generate-sf-name* is *gsf*.

A section can be rendered to a file via **Tools>Section->Csound.** The instrument should be chosen in the **Csound File Options**. More information is available in the Index item *Section->Csound.*

A plot of one parameter over time using P3 for the duration can be made via **Tools>Plot>Csound File**.



The above dialog can be used to plot the contents of P5 (which in the examples in this tutorial is frequency) over time.

A histogram of one parameter can be made via **Tools>Histogram>Csound File**. The file chosen for plots and histograms should be a csd file, not an audio file. Note that the representation is of the file data created by the Csound object.

If an audio file will be made with the same name as one open in another program such as QuickTime Player, the open file should be closed first.

If no signal is produced when the csd file is rendered, check that the proper orchestra and directories have been selected. If function statements are needed in the header, make sure that they are appropriate. Check the *Text Output* window for error messages from Csound. If the *Messages* option is not on, turn in on the Csound File Options dialog and render the file again to see possible problems.

To stop the rendering of a Csound file, use **Other>Kill** or CMD-K.

Summary

Menu items
Csound objects, Csound file options, plot>Csound File, histogram>Csound File, filter>Csound file, transform>Csound file

Generators
midi->hz, hz->midi, scale-by-layers, sbl, overlap, generate-sf-name, gsf

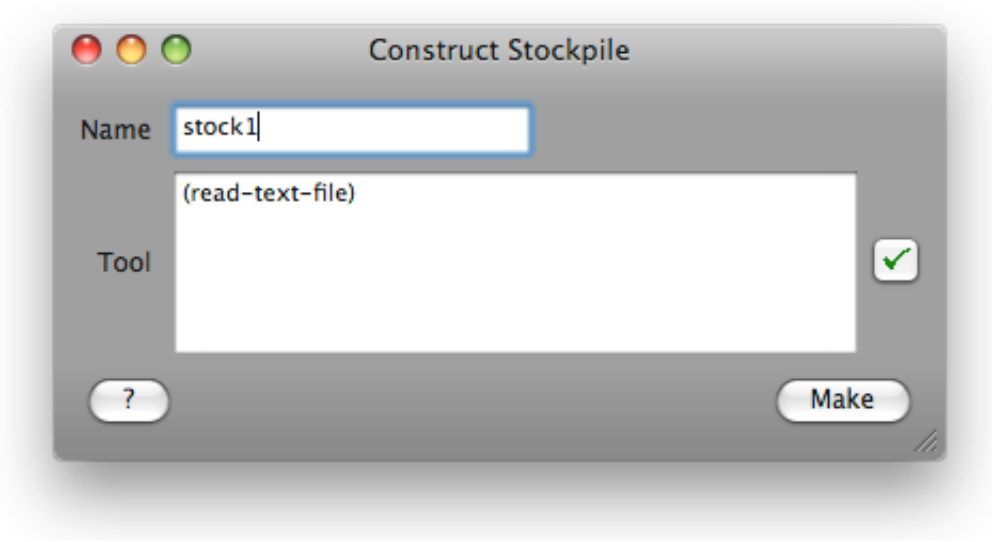
Tools
from-number, make&sort, ms, until-time, ut, density-of-start-times, density, show-midi->hz, sf-name, sf

Tutorial 16

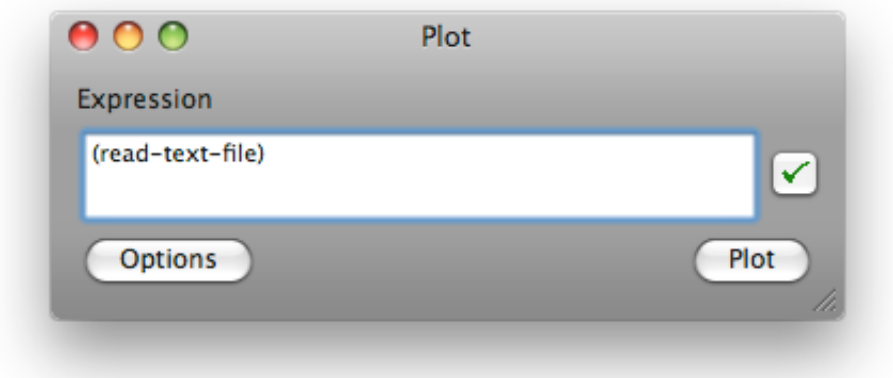
More contact with the outside world: reading and writing text files

Reading

Read-text-file
A text file can be read using *read-text-file*. When evaluated, a file dialog box appears. The values in the selected file are read in and returned in a list. This list can for example be used to construct a stockpile:



or to plot a file that only contains numbers:



To test this function, read in the sample text file, *Tutorial 16 Sample Text File.text*, that has been included in the folder with the other tutorial examples.

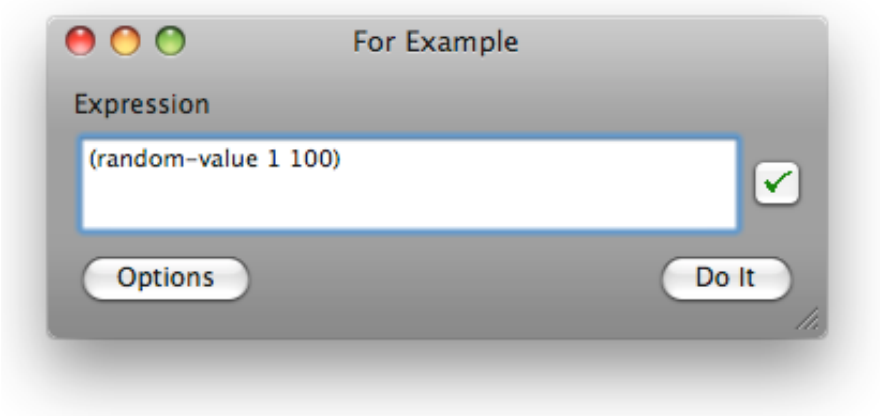
The text file should only contain numbers, letters, and words. No punctuation should be present.

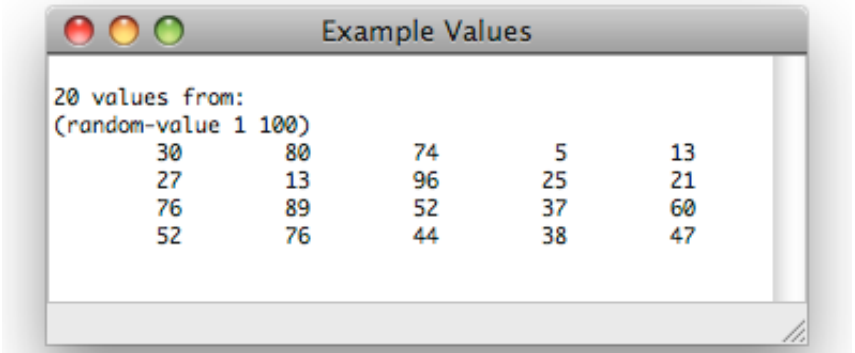
Writing: using menu items

Many programs use text files as input. The generators and tools of the AC Toolbox may be helpful in preparing data for other programs that accept text input. The generation of Csound score files was discussed in Tutorial 15. The Toolbox can also be used to produce text files in other formats for whatever reason.

The generation of binary OSC files is discussed in Tutorial 21.

For example
The **For Example** dialog allows examples of generator output or the results of Lisp expressions to be viewed in a window. To use this to produce a text file, save the window produced by choosing **File>Text>Save**. The data is the window can be edited before saving. The number of generator outputs to print can be controlled via the *Options*.

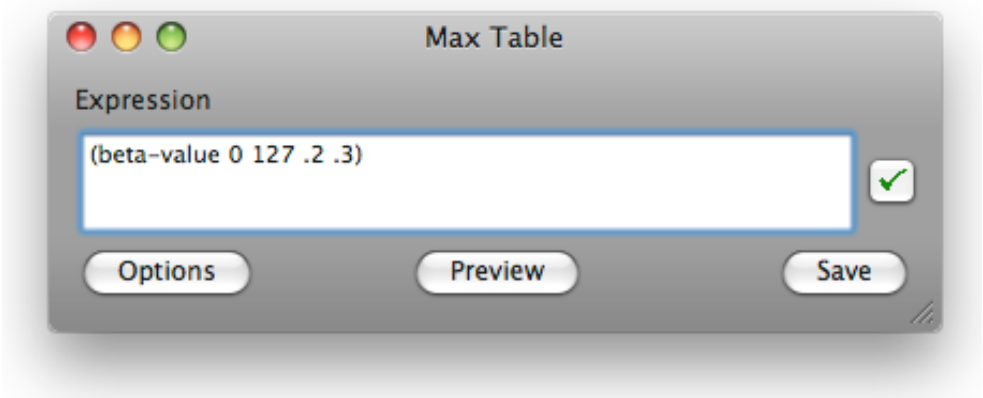




MAX table
The **File** menu has a **Write MAX Table** item that produces a dialog for making table files for the program MaxMSP. Tables of any size can be written. The default size is 128. To change that size, click on the *Options* button and enter a new value.

The table can be created with a generator, stockpile, etc. To see the result before it is saved to a file, click on the *Preview* button. The table values will be displayed. To create different values (assuming a generator with some indeterminacy is used), click on *Preview* again.

When acceptable values have been previewed (or if you do not want to see what is being stored), click on the *Save* button to save the table values to a file.



Writing: evaluating Lisp expressions

Unlike the previous tutorials where a file of objects can be read in and inspected, this part of the tutorial requires Lisp expressions to be evaluated in the Listener or in the editor. The expressions included in this tutorial have been gathered in a file, *Tutorial 16 Lisp Expressions.alsp* in the *Tutorial Examples* folder. This file can be opened with **File>Text>Open**. Each expression can be evaluated by selecting it and then choosing **Evaluate Selection** in the **Edit** menu or by placing the cursor at the end of the expression and pressing the enter key. Note that the enter key is not the same as the return key.

The result of evaluating expressions in the editor appears in the small message area at the bottom of the editor pane.

If expressions are typed and evaluated in the Listener (**Other>Listener**), the result appears under the typed expression.

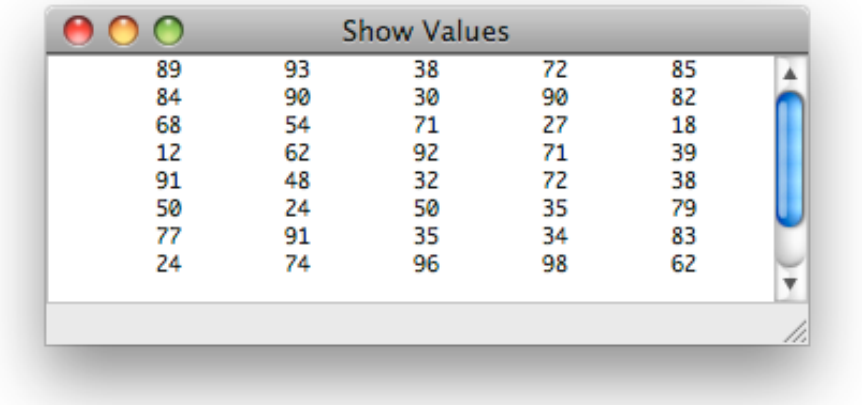
The expressions discussed in this tutorial are used for their side effects, namely printing values in a window or to a file. The result printed in the message area or Listener is of no concern for the task at hand.

To a window (then to a file)
The tool *Show* prints the results of a *number* of applications of a generator in a window. The window can be saved to a file by selecting the window and choosing **File>Text>Save**.

For example, evaluate the following expression in the editor or Listener:

```
(show 50 (random-value  1 100))
```

A window similar to the following should appear, containing 50 random values between 1 and 100:



The number of values printed per line can be specified with the keyword *:number-per-line*.

```
(show 50 (random-value  1 100) :number-per-line  4)
```

The title of the window can be changed using the keyword *:title* and including a string.

```
(show 50 (random-value  1 100) :title "Interesting")
```

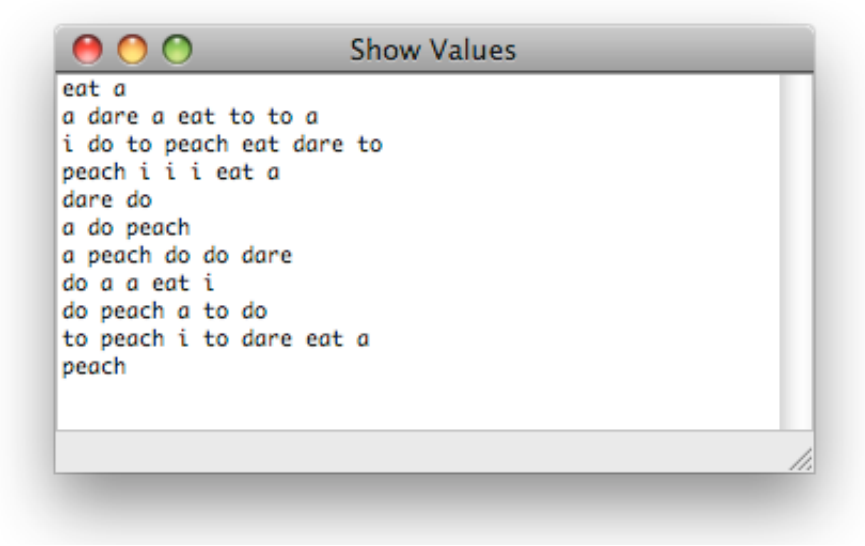
The size of the window can be specified with the keyword *:view-size*. Points are specified in the form '(horizontal vertical). The values for horizontal and vertical range from 0 to the maximum size available on the screen being used.

```
(show 50 (random-value  1 100) :view-size  '(300 300))
```

A generator can be used for the keyword :number-per-line.

```
(show 50 (random-choice '(do I dare to eat a peach))
              :number-per-line  (random-value  2  8)
              :view-size  '(250 250))
```

The above expression could produce an output similar to this:



Additional options can be found in the help text for *show.*

To a text window or file

Create-text-window prints the results of several different generators on one line in a window. The number of lines is specified. Create-*text-file* has the same syntax but prints directly to a file.

A *header* string is printed. A *number* of lines are printed. Each line contains the next value from a list, stockpile, generator, constant, etc., followed by a space, followed by the next value from the next Toolbox object, a space, etc. One value from each Toolbox object used as an argument to this function is included on a line. The file concludes with *foot-string*.

The basic input for the tool is

(create-text-window header-string foot-string number
generator generator ...)

For example, evaluate the following expression in the editor or Listener:

```
(create-text-window  "Some note numbers and durations" "The end" 10
                     (random-choice '(c2 d2 e3 f2 g2 a2 b2))
                     (random-value 1 10))
```

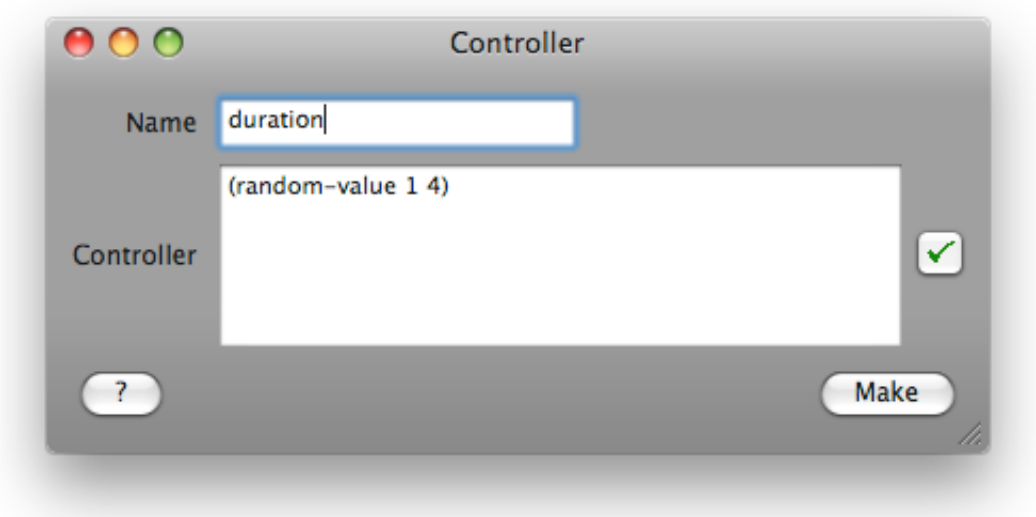
The header string is printed. Ten other lines are printed. Each line contains one result from *random-choice* and one result from *random-value.* The foot string is printed. The output text file should be similar to this:

```
Some note numbers and durations
45 7
36 1
45 3
38 5
43 2
43 6
47 1
52 8
41 1
52 9
The end
```

A similar expression could be used to write directly to a file. In this example, the header and foot are empty strings.

```
(create-text-file "" "" 10
                     (random-choice '(c2 d2 e3 f2 g2 a2 b2))
                     (random-value 1 10))
```

If a controller such as *duration* has been defined, it can be used to prepare data with cumulative start times and the duration of some events.




```
(create-text-window  "" "" 20
                     (walk 0 duration)
                     (generate (get-most-recent   duration)))
```

The arguments to *create-text-file* and *create-text-window* are evaluated left-to-right. We can therefore assume that *(walk 0 duration)* will be evaluated first. It causes the controller *duration* to be applied to produce a value. *Get-most-recent* returns the most recent value produced by a controller. It is encapsulated in a *generate* expression to produce a generator that can be applied to produce a new value each time.

A text window produced with the above specification should be similar to this:

```
0 4
4 1
5 4
9 2
11 1
12 4
16 2
```

Some programs want input to include expressions such as *start(1,10)* . To achieve this (with a few extra spaces), evaluate the following:

```
(create-text-window  "" "" 20
                     (always  "start(")
                     (random-value  10 20)
                     ","
                     (random-value  30 40)
                     ")")
```

Always always returns its input argument. In this context it is useful because it prevents *create-text-window* from looping through the string instead of printing it as one object.

Part of the text produced with the above specification:

```
start( 16 , 30 )
start( 11 , 35 )
start( 20 , 32 )
start( 14 , 35 )
```

Summary

Menu items
For Example, Write Max Table

Tools
read-text-file, show, create-text-window, create-text-file, get-most-recent, generate

Tutorial 17

Expanding the possibilities: writing generators, transformers, tools, and filters in Lisp

Writing a new generator, transformer, tool, or filter for the AC Toolbox requires programming at least one function in Common Lisp. If you do not want to program in Lisp, you should skip this tutorial.

Generators

A generator is a function that returns the next value in some series each time it is used. The bookkeeping necessary for it to know what is the next value is kept within the generator.

The AC Toolbox generator *major* returns a type of function called a closure. When that closure is applied, a value is produced. Each time that closure is applied, the next Midi note number in the major scale will be returned.

In the Listener (accessible via **Other>Listener**), *(major 60)* was evaluated and returned a closure (see below). *(major 60)* was evaluated again, to produce a new closure. This time the closure was used as an argument to *for-example*. This tool will apply the closure 20 times to produce Midi note numbers corresponding to a major scale.

```
CL-USER 1 > (major 60)
#<Closure (GENERATE-SCALE . 1) 20096B5A>

CL-USER 2 > (for-example (major 60))
NIL
```

In an example window, the previous expression prints

```
20 values from:
(major 60)
        60      62      64      65      67
        69      71      72      74      76
        77      79      81      83      84
        86      88      89      91      93
```

The standard scenario is that a function constructs a generator and returns it as its result. This result is then applied by the Toolbox to create values. The generator is returned as a lexical closure (a lambda expression with a function quote). The lambda expression should have no input parameters.

Lisp
The AC Toolbox contains a Common Lisp interpreter. An interpreter means that user-defined functions may run slower or use more memory than if they were compiled in a development environment. With some extra effort, it is possible to compile functions. See later in this tutorial for the possibilities.

The source code for the Lisp examples mentioned in this tutorial is contained in the file *Tutorial 17 Lisp Expressions.alsp*. This file can be opened with **File>Text>Open**. Each expression can be evaluated by selecting it and then choosing **Evaluate Selection** in the **Edit** menu or by putting the cursor at the end of the expression and pressing the enter key. The enter key is different from the return key.

When defining new functions, care should be taken not to use an existing function name. This is tricky since the interpreter does not warn if a name is used or not. The best way to deal with this is to use names that are very unlikely to be already taken, such as *my-random-choice*. Functions that are called by the new function could use even more exotic names to prevent name collisions.

Before defining a new function, you could check to see if the name is already used. This can be done in the Listener or editor window. Pass your new desired name as a string in

capital letters to the function *find-all-symbols*. This should be done before you have evaluated a function with that name.

```
(find-all-symbols "MY-NEW-NAME")
```

Instead of using a string with capital letters, the name could be preceded by a colon.

```
(find-all-symbols :my-new-name)
```

If there is no symbol *my-new-name*, the function will return nil.

You can also just check if a symbol has a function associated with it.

```
(fboundp 'my-new-name)
```

If the AC Toolbox behaves weirdly after a function has been defined, there may be a name conflict. Quit the toolbox and test the name of your new function with *find-all-symbols*.

Editor

A new window for Lisp code can be opened in the editor with **File>Text>New**. This editor will balance parentheses, allow basic Macintosh editing commands (such as cut, copy, paste, find), and includes some Emacs commands. The limited set of available Emacs commands can be seen by clicking on the *Key Bindings* button at the bottom of an editor window. More information about the editor can be found by clicking on the *?* button.

If a Lisp expression is evaluated in the editor, the result appears in the small echo area at the bottom. Functions defined in the editor can be used anywhere in the AC Toolbox.

A simple example

A generator can be expressed as a lambda expression with a function quote. Function *the-same* returns a lambda expression with a function quote. Each time that the returned closure is applied, the same value is produced. Note also that the lambda expression has no input parameters.

```
(defun the-same (value)
   #'(lambda ()
       value))
```

To produce 20 values using the closure produced by *the-same,* evaluate the following expression in the editor or Listener:

```
(for-example (the-same  10))
```

The following will appear in a window:

```
20 values from:
(the-same 10)
        10        10        10        10        10
        10        10        10        10        10
        10        10        10        10        10
        10        10        10        10        10
```

Storing data

A generator must maintain its own bookkeeping information. A locally bound variable should be used to do this. This locally bound variable however must maintain its storage location outside the boundaries of the lambda expression otherwise a new location will be created each time the closure is applied.

Integers produces a closure. When applied, it will produce a series of increasing integers, starting with the value of *n*.

```
(defun integers (n)
   (let ((this-value (- n 1)))
     #'(lambda ()
         (incf this-value))))
```

To produce 20 values in the AC Toolbox Listener using the closure produced by *integers*:

```
(for-example (integers 1))
```

The following will appear in a window:

```
20 values from:
(integers 1)
         1          2          3          4          5
         6          7          8          9         10
        11         12         13         14         15
        16         17         18         19         20
```

If the previous value is stored in a location specified by a locally bound variable within the lambda expression, the new location will be created each time the expression is applied. There is no way to 'remember' what the previous value was. Function *no-good* does this. The resulting closure always repeats the value of *n* instead of increasing it.

```
(defun no-good (n)
   #'(lambda ()
       (let ((this-value (- n 1)))
         (incf this-value))))

(for-example (no-good 1))

20 values from:
(no-good 1)
         1          1          1          1          1
         1          1          1          1          1
         1          1          1          1          1
         1          1          1          1          1
```

My-random-value

A version of the Toolbox generator *random-value* will now be programmed. This version will be called *my-random-value* to distinguish it from *random-value*. The Toolbox tool *alea* will be used. *Alea* is a function that returns a random value between (and including) two limits. If one or more of the limits is a real number, the result will be a real number. If both limits are integers, an integer is returned.

```
CL-USER 1 > (alea 10 20)
11

CL-USER 2 > (alea 10.0 20)
19.324902
```

Note that *alea* is not a generator. Applying *alea* produces a number rather than a closure as the result. Using *alea* as an input for *for-example* results in just one value being printed. If the input to *for-example* is a generator, it is applied several times. If the input is something else, such as a tool that produces one value or a list, just that value or list are printed.

A first version of *my-random-value* can be programmed using *alea*.

```
(defun my-random-value (lower upper)
   #'(lambda ()
       (alea lower upper)))
```

```
(for-example (my-random-value 1 10))

20 values from:
(my-random-value 1 10)
         2         8         9         9         3
         8         8         7         5         5
         1         1        10         9         3
         6         4         8         4         9
```

Usually it is desirable for generators to allow parameters to change over time. This can be done by using the generator *do-the-next-thing* in the closure. *Do-the-next-thing* produces a closure that, when applied, produces the next value. If the input is a constant value, the next value will always be that constant. If the input is a list, the next value will be the next value from the list. If the input is a generator, the next value is the next application of the generator, etc.

```
(for-example (do-the-next-thing '(1 2 3)))

20 values from:
(do-the-next-thing '(1 2 3))
         1         2         3         1         2
         3         1         2         3         1
         2         3         1         2         3
         1         2         3         1         2

(for-example (do-the-next-thing (my-random-value 1 10)))

20 values from:
(do-the-next-thing (my-random-value 1 10))
         7         2         4         5         7
         7         4         9         3         5
         9         7         5        10         2
         4         9         4         8         5
```

When an input parameter of *my-random-value* is enclosed in a *do-the-next-thing* expression, the resulting closure should be applied each time to get the appropriate parameter value.

```
(defun my-other-random-value (lower  upper)
   (let ((lower-function (do-the-next-thing  lower))
         (upper-function (do-the-next-thing  upper)))
     #'(lambda ()
         (alea (funcall  lower-function) (funcall  upper-function)))))
```

```
(for-example (my-other-random-value
                     (line-segment 20 1 10)
                     (line-segment 20 10 20)))

20 values from:
(my-other-random-value (line-segment 20 1 10) (line-segment 20 10 20))
         5     8.659     8.829     5.552    10.044
    10.419    11.871     6.548    10.053     8.463
     9.538    11.882    16.031     9.269     9.783
    12.621    16.737     9.325    17.159    11.309
```

My-random-choice
Generators with the suffix *-choice* choose from a list, a stockpile or several other types of object.

Get-stockpile returns a list of values derived from various types of objects. If the object is a stockpile, a list of elements in that stockpile is returned. If the object is a section, a list of notes (without their start-times) is returned, etc. *Get-stockpile* should be used when writing a *choice* generator to insure that a suitable list is available from which to make the choice.

The generator also needs to calculate the index value of the element to be returned from the stockpile. An element with position *I* can be sought in a list by using *elt*. With *elt*, elements start with 0 instead of 1.

```
(defun my-random-choice  (a-stockpile)
   (let* ((this-stockpile (get-stockpile  a-stockpile))
          (this-stockpile-length (length  this-stockpile)))
     #'(lambda ()
         (elt this-stockpile (random this-stockpile-length)))))

(for-example (my-random-choice '(1 2 4 6 8)))

20 values from:
(my-random-choice '(1 2 4 6 8))
         6         8         2         8         1
         2         8         4         2         6
         6         4         1         6         8
         1         1         6         6         2
```

In *my-random-choice*, the index is calculated using *random* instead of alea since the lower limit is always 0. The extra possibilities of *alea* are not needed in this case.

Adding documentation
Help information can be added for a user-defined generator. This help will only remain until the AC Toolbox is quit. After that it should be reloaded.

To add help, evaluate an *add-new-generator* expression. The syntax is
(add-new-generator name text examples &key basic chaos ...)

Name should be quoted, e.g. *'my-random-choice*, and the text should be a string (i.e. between two double quote symbols, e.g. *"This is my text"*). The examples should also be in one string. If any of the keywords are bound to *t*, the generator help will also appear in the table listing generators of that particular category. All categories found in the popup menu in the Annotated Index may be used as keywords for this tool. More than one keyword can be bound.

To add help for *my-random-choice*, evaluate the following form:

```
(add-new-generator 'my-random-choice
  "(my-random-choice stockpile)

Returns a closure. When applied a random choice is made from a
stockpile.

A stockpile  can be a list, Toolbox stockpile  object, etc (example 1).
"
"
EXAMPLE 1: (for-example (my-random-choice '(1 2 3 6 8)))
" :noise t)
```

Help for *my-random-choice* will be available in the *Index* dialog the next time it is opened.

The brief help that is available in the Annotated Index can be added with
(add-generator-brief name text)
where text is a string. The brief help is available in the category *All* and in the category chosen as a keyword in add-new-generator.

```
(add-generator-brief 'my-random-choice "My version of a random choice")
```

The strings for help in the add-new-generator tool may contain formatting characters.
@p starts and ends a paragraph
@h starts a header (capitals) which is ended with @/

Using formatting characters, the documentation could be expressed in this way:

```
(add-new-generator 'my-random-choice
  "@h(my-random-choice stockpile)@/

@pReturns a closure. When applied a random choice is made from a
stockpile.@p

A stockpile  can be a list, Toolbox stockpile  object, etc (example 1).
"
"
EXAMPLE 1: (for-example (my-random-choice '(1 2 3 6 8)))
" :noise t)
```

Documentation will only become available after the Index or the Annotated Index has been closed and reopened.

Transformers

The transform method in the AC Toolbox allows various kinds of objects to be transformed. Section parameters can be transformed separately (pitch, tempo, attacks, duration, velocity, and channel). *Whatever* allows transformations of other kinds of objects.

The Toolbox does most of the work in regard to transforms. It extracts the appropriate value from the object being transformed, applies the indicated transformer, and constructs a new object using the newly calculated value. All that a user-defined transformer must do is transform the value provided to it by the Toolbox.

Transformers are usually lexical closures (lambda expressions with a function quote) that have one input parameter. These closures are generally constructed by a function. The input parameter is the value to be transformed. The transform method in the Toolbox applies the lexical closure to the appropriate parameter.

My-transpose
A simple definition for *my-transpose*:

```
(defun my-transpose (interval)
   #'(lambda (x)
        (+ x interval)))
```

In the above definition, *x* is the value to be transformed and *interval* is the value added to it. This simple definition will however not work properly if the section being transformed contains chords that are expressed as a list of pitches. It also does not allow *interval* to change over time.

To allow *interval* to change over time, it should be encapsulated in a *do-the-next-thing* expression and the resulting closure applied each time a value is needed. This is the same strategy that was used for time-varying parameters in a generator.

To deal with chords, the transformer should check *x* to see if it is a list. If so, each element in that list should be transformed. If *x* is not a list, then *x* should be transformed.

```
(defun my-other-transpose (interval)
  (let ((interval-function (do-the-next-thing interval)))
    #'(lambda (x)
        (cond ((listp x)  ; if a chord, transform each element
               (let ((value (funcall interval-function)))
                 (mapcar #'(lambda (element)
                             (+ element value))
                         x)))
              (t (+ x (funcall interval-function)))))))
```

Note in the above example that the value to be added to each element in the list is only calculated once. If *interval-function* were to be applied separately for each element in the list, each note in the chord could have a different transposition interval.

Adding documentation
Help information can be added for a user-defined transformer. This help will only remain until the AC Toolbox is quit. After that it should be reloaded.

To add help, evaluate an *add-transformer* expression. The syntax is
(add-new-transformer name text examples &key basic chaos ...)

Name should be quoted, e.g. ‘my-other-transpose, and the text should be a string (i.e. between two double quote symbols, e.g. “This my text”). Examples should also be in one string.

To add help for *my-other-transpose*, a form such as this could be used:

```
(add-new-transformer 'my-other-transpose
"@h(my-other-transpose interval)@/

@pReturns a closure that adds a numerical INTERVAL to the value being
transformed (example 1).@p

@pINTERVAL may vary over time using a generator, a list, a stockpile,
 etc. (example 2).@p
"
"
EXAMPLE 1: (show-transformation (my-other-transpose 12)
                                '(60 62 64 65))
EXAMPLE 2: (show-transformation
              (my-other-transpose (random-value -5 5))
                                  '(60 60 60 60 60))
 ")
```

The brief help that is available in the Annotated Index can be added with
(add-transformer-brief name text)
where text is a string.

Formatting characters can be used with add-new-transformer as described for generators.

Tools

Tools are 'regular' Lisp functions that return a value, or a list, etc. There are no special requirements for defining tools. The limitations described below are particularly relevant when writing tools.

Documentation can be added with *add-new-tool.* It behaves similarly to *add-generator*.
(add-new-tool name text examples &key basic chaos ...)
The available keywords are the categories present in the Annotated Index.

The brief help that is available in the Annotated Index can be added with
(add-tool-brief name text)
where text is a string.

Formatting characters can be used as described for generators.

Filters

Filters are usually lexical closures (a lambda expression with a function quote) that have one input parameter. These closures are generally constructed by a function.

A user can define filters that would be used as an OTHER filter when using the AC Toolbox filter method.

If a filter returns nil, a value if removed. If it returns a nonNIL value, the value is retained. The AC Toolbox passes the value to be filtered through the input parameter of the filter.

A simple example of a low-pass filter:

```
(defun low-pass-example (threshold)
                #'(lambda (input)
                        (<= input threshold)))
```

This filter could be used as an 'other' filter.

If the threshold for the low-pass filter should change over time:

```
(defun low-pass-example-tv (threshold)
        (let ((threshold-function (do-the-next-thing  threshold)))
                        #'(lambda (input)
                        (<= input (funcall  threshold-function)))))
```

If a section is used with the *Whatever* parameter in a filter, a list containing the data received from the filter method should be returned.

Documentation for a filter uses *add-new-tool*.

Loading definitions

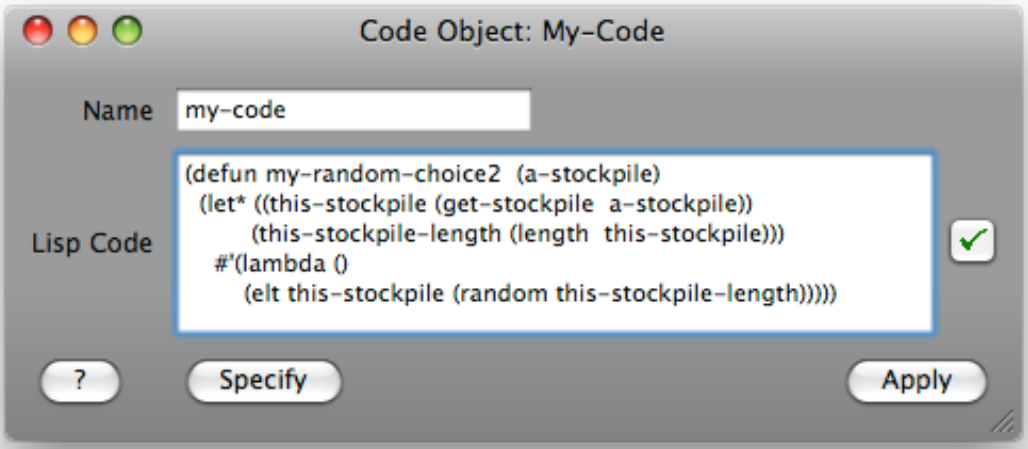
A file containing user-defined generators, etc. can be loaded using the **Load Lisp File** menu item in the **File** menu.

A file can be automatically loaded when the AC Toolbox is launched by selecting it with **File>Choose Init File.** Note that this only loads the file when it is started, not at the moment it is selected with this menu item.

The file could contain definitions and/or instructions to load other files. If the form
`(load "my-definitions.lsp")`
is included in the init file, the file *my-definitions.lsp* will be loaded when the application is launched. To load a file that is not in the current folder, the path to that file should be given, e.g.: "/Users/paulberg/Documents/my-definitions.lsp".

Code Objects

Lisp code can also be stored in a *code object*. This is an AC Toolbox object that acts as a repository for Lisp code. It can contain any number of Lisp expressions or definitions. When the *Apply* button is clicked, the code is evaluated and can be used in other expressions and objects. The advantage of a code object is that it is stored with other AC Toolbox objects when the environment is saved.



When an environment containing a code object is loaded, the code object must be evaluated (by clicking on the *Apply* button) before the function(s) can be used.

Limitations

The AC Toolbox is written in LispWorks Common Lisp. A Lisp interpreter is available in the AC Toolbox. The interpreter may produce code that is not efficient enough for the task at hand. E.g. recursive functions may cause a stack overflow. Performance can be improved by explicitly compiling functions: *(compile 'function-name)*.

```
(defun fact (n)
           (if (= n 0)
               1
             (* n (fact (- n 1)))))
```

Fact will probably produce a stack overflow when the following expression is evaluated:

```
(fact 10000)
```

Compiling the function may solve this problem:

```
(compile 'fact)
(fact 10000)
```

Another possibility if many functions should be compiled is to compile them in a compatible version of LispWorks. The LispWorks Personal Edition may be adequate for the task. Compatibility can be determined by trial and error or by evaluating

```
(lisp-implementation-version)
```

in the Listener of both the AC Toolbox and LispWorks.

The compiled file can be read in with the menu item **File>Load Lisp File**.

Summary

Menu items
evaluate selection, load lisp file, choose init file, code object

Tools
get-stockpile, add-new-generator, add-new-transformer, add-new-tool, add-generator-brief, add-transformer-brief, add-tool-brief

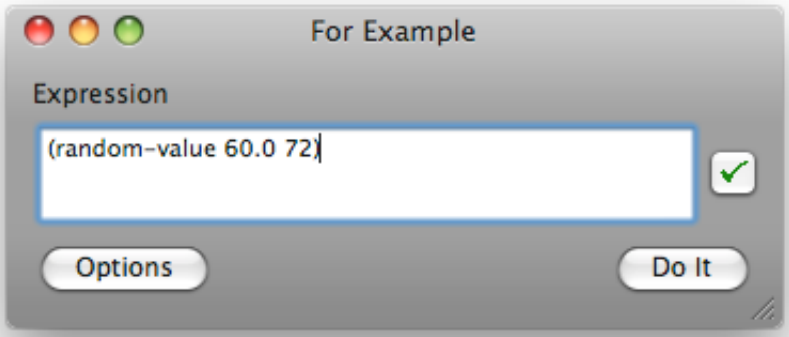
Lisp
compile, find-all-symbols, fboundp

Tutorial 18

Expressing microtones

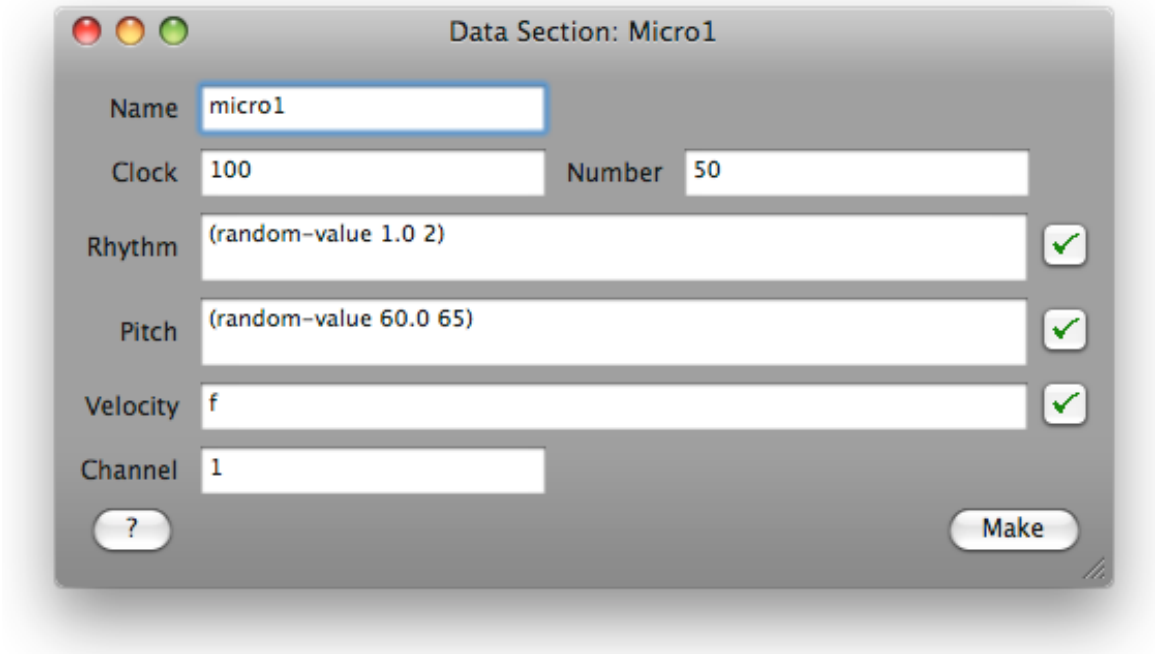
Microtones are expressed in the AC Toolbox as floating-point numbers. *60* is c4. *60.5* is a quarter-tone higher. Any value after the decimal point is possible but may not be meaningful.

To generate floating-point numbers, it is sufficient for most generators or tools to use a floating-point number for one or more of the input parameters.




```
20 values from:
(random-value 60.0 72)
    66.721    68.725    62.715    62.269    65.900
    66.359    65.817    70.301    64.544    68.518
    65.597    61.459    64.127    60.856    70.845
    67.592    60.495    63.282    61.496    64.020
```

A section can be defined using microtones for pitch values. Currently, microtones can only be played using Quicktime or a Capybara/Pacarana. For all other Midi output destinations, including Midi files, pitch values will be rounded to integer Midi note numbers.



The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 18 Examples* in the *Tutorial Examples* folder. Load these objects by selecting **Load Examples**

in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

• Make section *micro1* and listen to it using QuickTime as the Midi destination (chosen in Midi Setup).
• Change the pitch expression to *(random-value 60 65)* and notice that fractional pitch values are no longer produced.

To produce a floating point value when using symbolic pitch notation, enclose the symbol in the expression *float*:

```
(random-value (float c4) f4)
```

Only quarter-tones
From is a tool that produces a list of numbers between two limits. A step size can be specified, for example to produce quarter-tones:

```
(from 60 72 :step .5)
```

This list could be used as an input to a generator ending with *-choice* to make choices from this set of quarter-tones.

```
Name            micro2
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (random-choice (from 60 72 :step .5))
Velocity        f
Channel         1
```

Another way to limit microtones to certain intervals is to use the generator *round-off*. Input values are rounded to be multiples of some specified unit. The unit can be any number, including .5, .25, etc.

```
(round-off (random-value 60.0 72) .5)
```

Many generators have a keyword *round* which does the same thing as *round-off*. Floating-point values are rounded to a specified quantization unit. In section *micro3,* the random floating-point values for pitch are rounded to the nearest multiple of .5.

```
Name            micro3
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (random-value 60.0 72 :round .5)
Velocity        f
Channel         1
```

• Make and play section *micro3* using a different rounding unit, such as .25.

Several generators that use optional parameters instead of keywords, e.g. *exponential-value*, have an optional parameter for *round*. *Series-value* needs to work with discrete values to prevent repetitions. It has a keyword *step* which can be used to define a set of pitches with microtonal intervals.

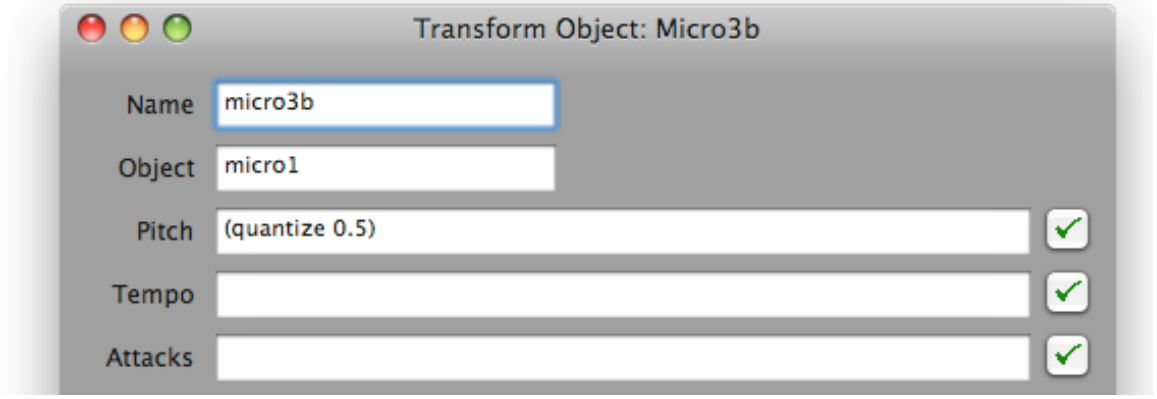
```
(series-value 60 72 :step 0.5)
```

The Transform method can be used to quantize the pitches of a section made with floating-point values. Pitch values can be quantized to quarter-tones with:

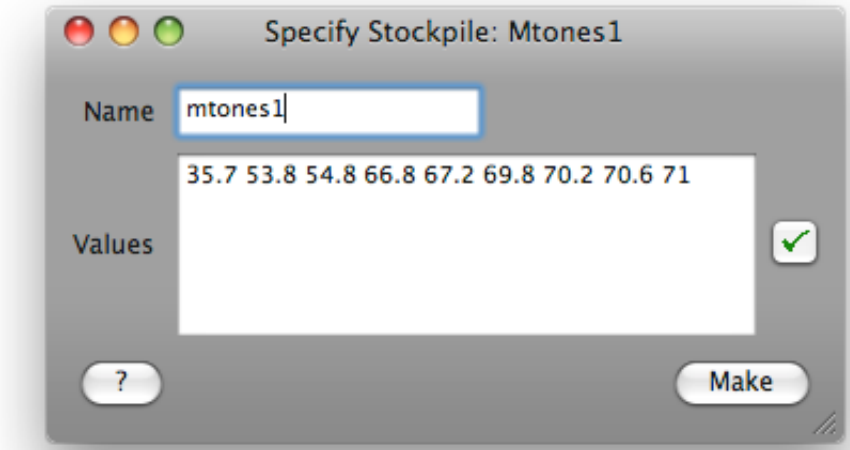
```
(quantize .5)
```

To transform a section with floating-point values to one with only integer values, pitch can be modified with:

```
(quantize 1)
```



Using a stockpile
A set of microtonal values can be specified in a stockpile and used for making selections with various generators.




```
Name            micro4
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (random-choice mtones1)
Velocity        f
Channel         1
```

Using a mask
A mask can be converted into fractional values for pitch and if desired, rounded to quarter-tones or any other microtonal unit.

• Make the example *mask1* or design your own mask and name it *mask1*.

This mask can be converted using *convert*. If the value for either *low* or *high* is a floating-point number, the result will be a floating-point number.

```
(convert mask1 50 60.0 72)
```

Such an expression can be used in a section.

```
Name            micro5
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (convert mask1 50 60.0 72)
Velocity        f
Channel         1
```

To limit results converted from the mask to quarter-tones (or any other unit), *round-off* can be used.

```
Name            micro6
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (round-off (convert mask1 50 60.0 72) .5)
Velocity        f
Channel         1
```

Another way to limit the results of converting a mask is to use tool *convert2*. It is similar to *convert* but has a keyword for *round*.

```
Name            micro6b
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (convert2 mask1 50 60.0 72 :round 0.5)
Velocity        f
Channel         1
```

If a mask should be used to read an arbitrary set of microtones (that are not spaced at equal steps from each other), *read-from* could be used. The stockpile (or list) being read is spread on the Y-axis and the mask is used to read from those values. Mask values are considered positions on the Y-axis expressed as percentages (from 0 - 100 %).

```
Name            micro7
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (read-from mtones1 mask1 50)
Velocity        f
Channel         1
```

Scale
Various generators and tools exist to create scales. They could be used to created microtonal scales, for example to fill a stockpile. This could be done with *define-scale*, *walk*, and *generate-scale* among others.

Generate-scale will start at a base-pitch and successively add interval values to the preceding pitch.

```
20 values from (GENERATE-SCALE 36 '(1 2 2.5 1)):

        36          37          39      41.500      42.500
    43.500      45.500      48.000      49.000      50.000
    52.000      54.500      55.500      56.500      58.500
    61.000      62.000      63.000      65.000      67.500
```

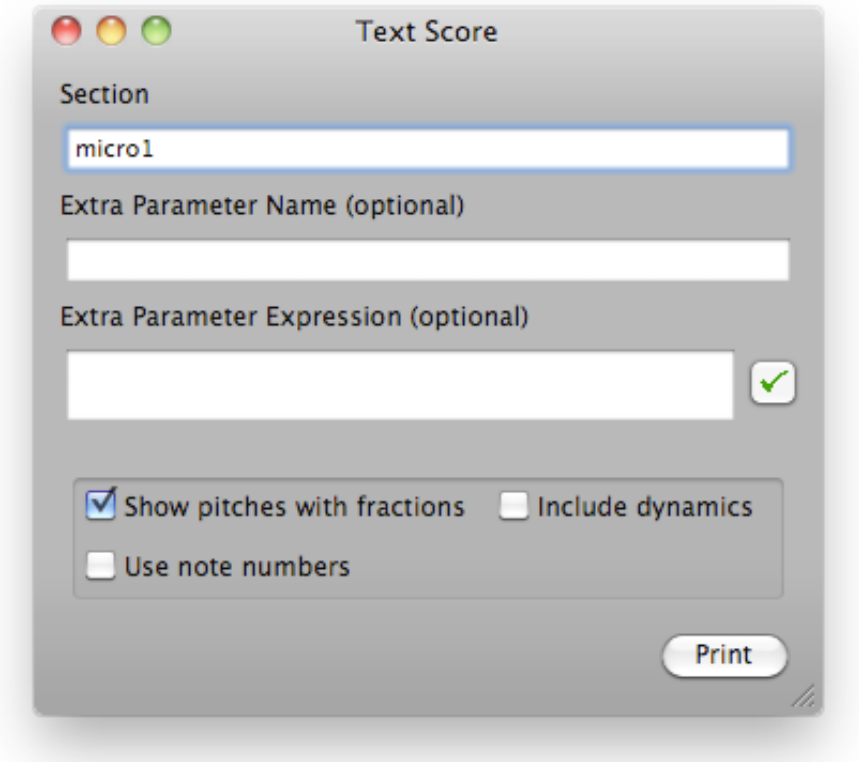
• Make stockpile *mtones2*.

Section *micro8* reads from the stockpile *mtones2* that contains a microtonal scale.

```
Name            micro8
Clock unit      100
Number          50
Rhythm          (random-value 1.0 2)
Pitch           (read-from mtones2 mask1 50)
Velocity        f
Channel         1
```

Printing microtones as text

The menu item **Tools>Text Score** allows a very simple text score to be printed representing pitch names, start times, durations, and optionally dynamics and an additional parameter. In addition to the pitch name, it is possible to print the fractional part of the pitch in parentheses after the name. There is a check box to enable this option. If this box is not checked, pitches are rounded.



The following is a portion of the text score for *micro1*.

```
   START        END   DURATION    PITCH

   0.000      1.280      1.280    c4 (.285)
   1.280      3.040      1.760    c#4 (.217)
   3.040      4.660      1.620    c4 (.365)
   4.670      6.640      1.970    e4 (.103)
   6.640      7.880      1.240    d#4 (.688)
   7.880      8.940      1.060    d4 (.592)
   8.940     10.120      1.180    d4 (.698)
  10.130     11.790      1.660    d4 (.192)
```

The fractional values shown are added to the represented pitch to realize the calculated pitch value. C4 (.285) would be 60.285

A text score representation may be helpful in notating microtones in a conventional score.

In an edit window (which can be opened with the Edit button in the Objects dialog), microtonal pitches are printed as floating point values.

Miscellaneous
All sections can store microtonal pitch values. All microtonal values will be saved in the environment when one is saved. Midi files however cannot use these fractional values. Pitch values will be rounded when a Midi file is written.

QuickTime and Capybara/Pacarana are currently the only two Midi output destinations that will perform microtones in the AC Toolbox. The pitch resolution of the Capybara/Pacarana is greater than that of QuickTime.

Summary

Generators
generate-scale

Tools
convert, convert2, from

Tutorial 19

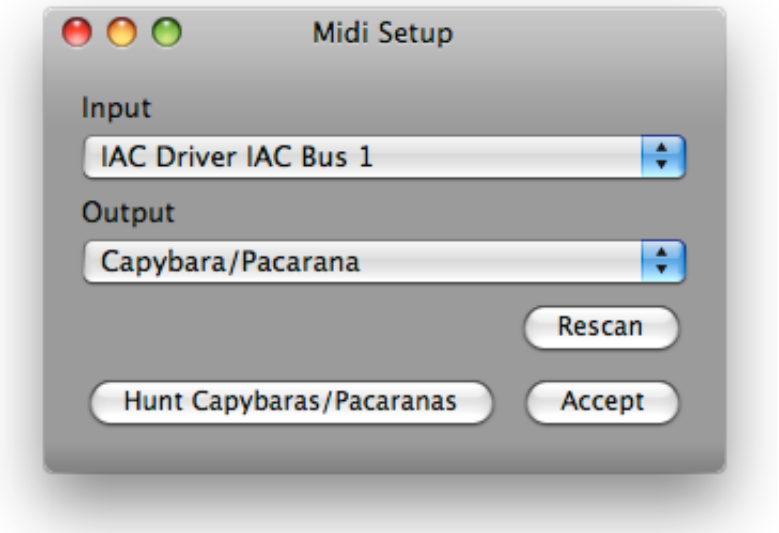
Controlling a Capybara/Pacarana

A Capybara is a sound computation engine produced by the Symbolic Sound Corporation and a rodent that some South Americans barbecue. A Pacarana is also a rodent and a sound computation engine from the same company. This tutorial concerns the sound computation engines. Although they are separate devices, they use the same software and are interchangeable for the purpose of this tutorial. Therefore the term Capybara/Pacarana is used in this set of tutorials.

Midi output from the AC Toolbox can be routed to a Capybara/Pacarana via a FireWire interface. This allows Midi data to be expressed as floating-point values and is particularly interesting for pitch and controller data. Secondly, this allows a connection between the control mechanisms of the AC Toolbox and the sound design capabilities of the Capybara/Pacarana.

Midi setup
The Midi Setup dialog is available via the menu item **File > Midi Setup**.
Capybara/Pacarana may appear as one of the possible destinations in the Midi output popup menu. If not, connect and/or turn on the Capybara/Pacarana and hit the button **Hunt Capybaras/Pacaranas**. This does what the name says. If a Capybara/Pacarana is found, it is added to the popup menu and can be selected as the Midi output destination. All subsequent Midi output will then be sent to the Capybara/Pacarana.

It is up to the user to ensure that the Capybara/Pacarana does something with this Midi data. The AC Toolbox mechanisms for producing Midi data to send to the Capybara/Pacarana are the same mechanisms used for producing other Midi data in the Toolbox. In fact, the objects could be sent to other Midi destinations. Earlier tutorials can be consulted for information on making data sections, density sections, controller objects, program change objects, etc.



Kyma
This tutorial assumes that the user is familiar with Kyma, the language for programming the Capybara/Pacarana. Kyma sounds containing Midi commands such as `!Pitch`, `!KeyVelocity`, `!cc01` should be downloaded by Kyma to the Capybara/Pacarana to allow for control by the AC Toolbox. The sounds can be directly downloaded or loaded from a timeline. In both cases, the user should ensure that the appropriate Midi channels are available.

Kyma offers extensive possibilities for sound design. Sound design is however outside the scope of this tutorial. A few simple examples using a Capybara/Pacarana as Midi destination will be presented to demonstrate the control mechanism.

The Kyma sounds used in this tutorial can be found in *Tutorial 19 Kyma sounds* that is in the *Tutorial 19 Capybara_Pacarana* folder in *Tutorial Examples*.

• Load Kyma, open the file *Tutorial 19 Kyma sounds*. Compile, load, and start the sound *25 sines*. This sound will produce 25 sine wave oscillators with a simple envelope. Amplitude is controlled with faders on the Virtual Control Surface that are labeled *MixAmp* and *Gain*. They may need to be adjusted depending on the number of oscillators used in the various examples.

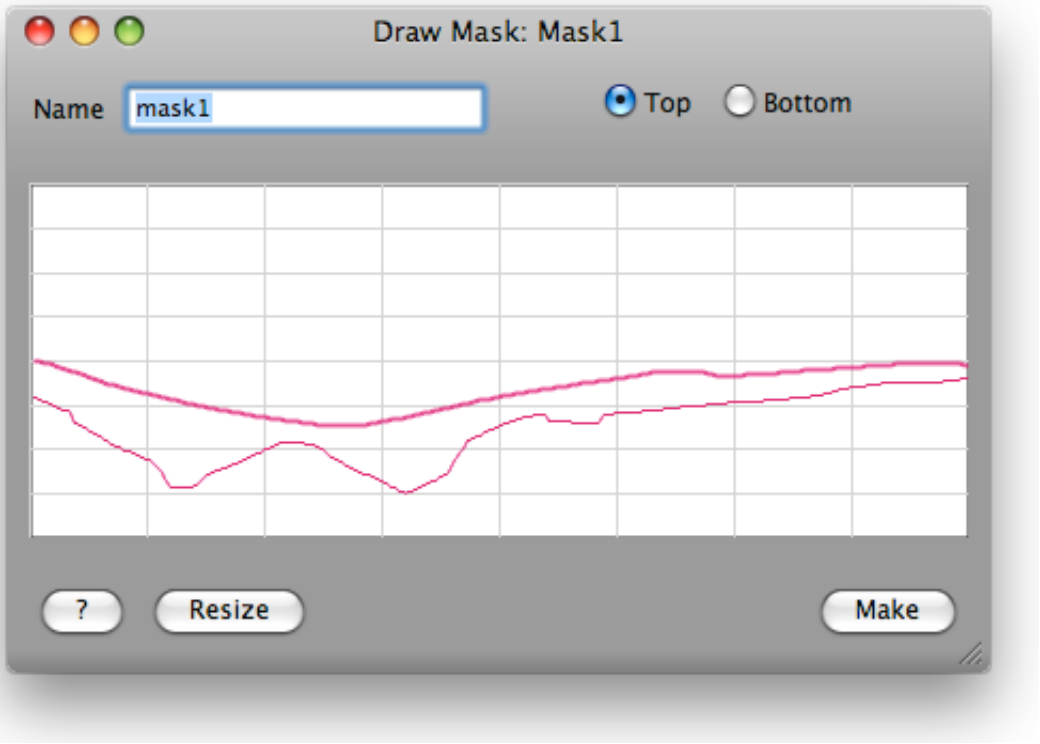
• Set the default Midi channel to 1 in the **DSP > Configure Midi** menu item in Kyma.

Note numbers

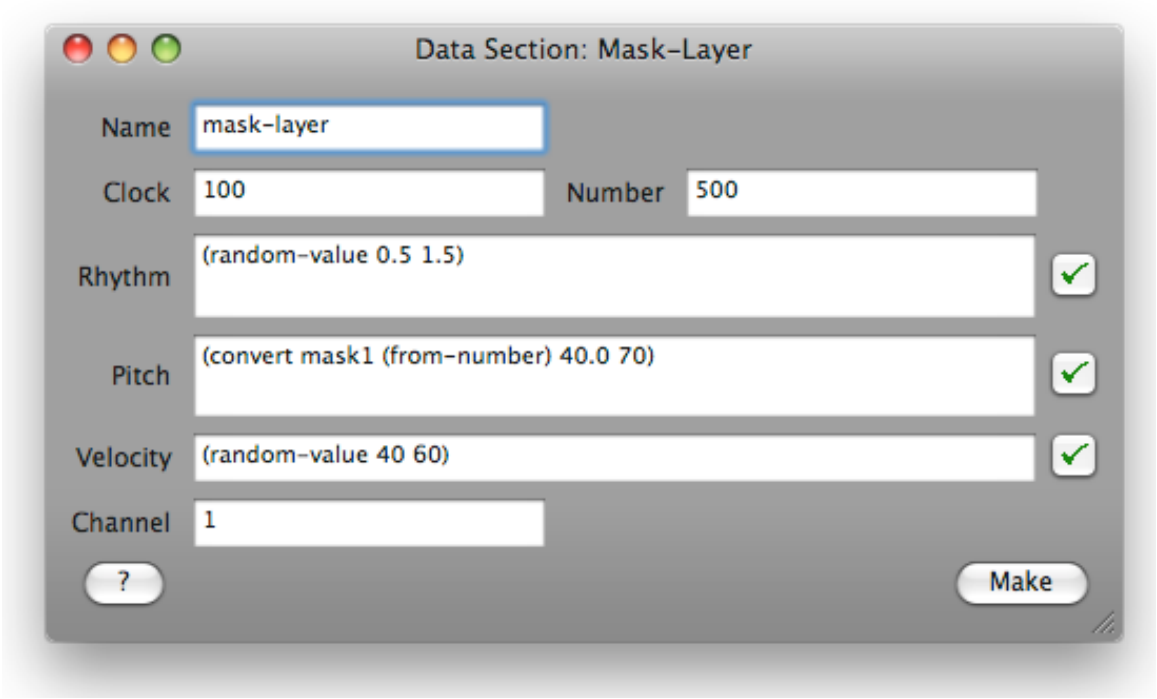
Masking sine waves
After the Kyma sound *25 sines* has been started and Capybara/Pacarana has been chosen as the Midi output destination (in Midi Setup), Midi note on and note off data generated by the AC Toolbox will be sent to this instrument of 25 sines.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 19 Examples* in the *Tutorial 19 Capybara_Pacarana* folder. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

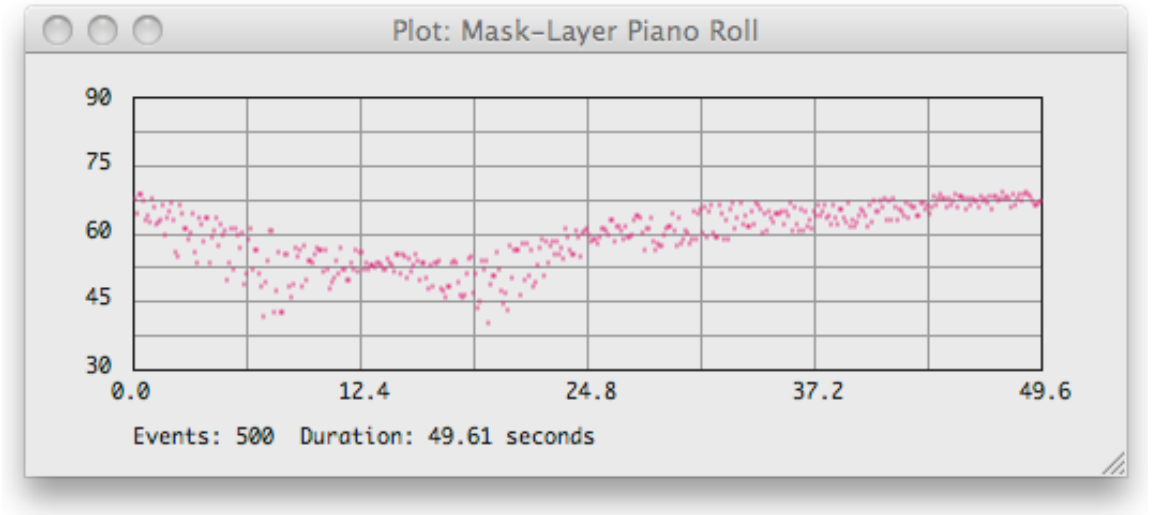
• Click on *mask1* in the list of examples to see a possible mask.



Section *mask-layer* defines a data section with one layer of output produced by converting *mask1* to pitch data. This conversion contains a floating-point number for the parameter low causing the results of convert to be floating-point numbers.



• Make section *mask-layer* and play it. A series of sine waves with frequencies corresponding to Midi note numbers between 40.0 and 70 will be produced. (If for some reason, no sound is heard from the Capybara/Pacarana, recheck the Midi Setup and make sure that the choice for Capybara/Pacarana is accepted. Make sure that Kyma is expecting Midi input on channel 1 as is specified in the object *mask-layer*. If that does not help, restart the AC Toolbox.)



In *mask-layer*, the mask was mapped to be values between Midi note numbers 40 and 70. If a user would rather express values in Hz, generator *hz->midi* should be used to map the result. If the mask in section *mask-layer* should be converted to values between 100 and 1000 Hz, the following expression could be used for the pitch parameter:

```
(hz->midi (convert mask1 (from-number) 100.0 1000))
```

The currently active Kyma sound allows the use of up to 25 sines. If more sines are desired, the polyphony in Kyma should be increased. The AC Toolbox can handle up to 100 Kyma voices.

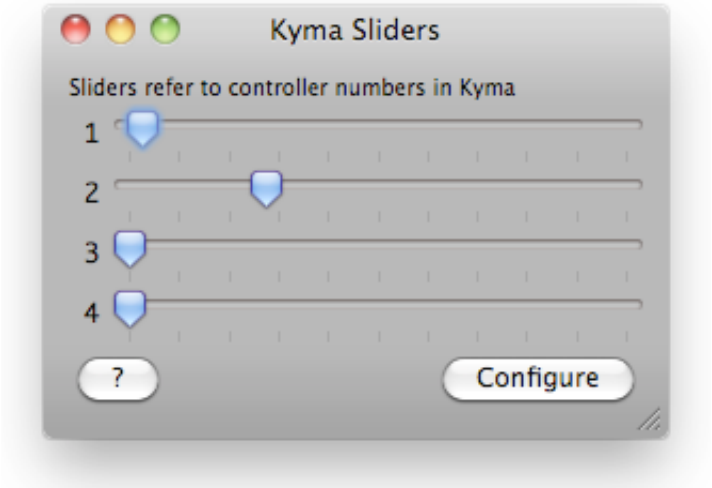
Section *many-mask-layers* produces 25 layers that are variants of section *mask-layer*. This means that the input for *mask-layer* is used to produce 25 new sections that are joined in

parallel. The offset is the time in seconds from the beginning of the parallel section before each of these layers start.

The amplitude or gain faders in Kyma may need to be adjusted to prevent distortion.

Kyma sliders
Once a Kyma sound has been loaded in a Capybara/Pacarana, the user can work in the AC Toolbox without having to deal with Kyma at all. Making adjustments to the Virtual Control Surface in Kyma requires jumping back and forth from the Toolbox to Kyma.

A window with Kyma sliders in the AC Toolbox (**Other>Kyma Sliders**) is intended as a convenience to limit the amount of jumping back and forth that is necessary.

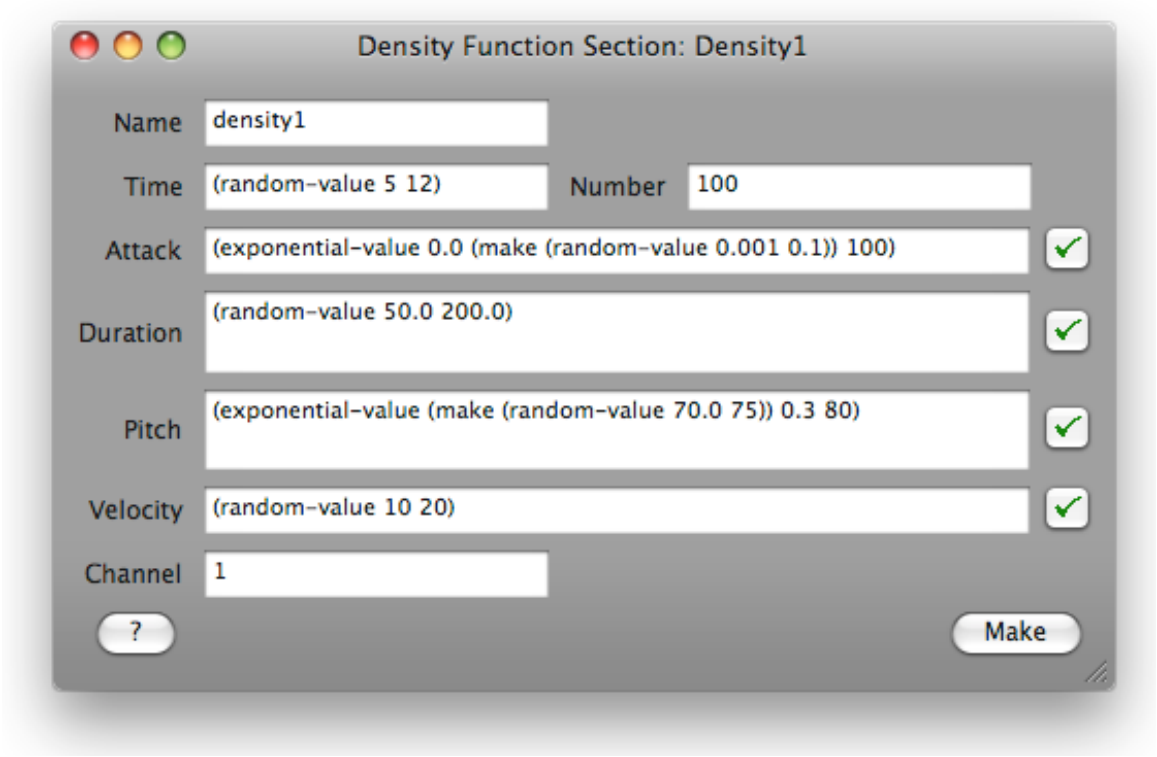


The AC Toolbox can send Midi controller data to a Kyma sound that has a continuous controller (such as `!cc01`) as a parameter. Controller 1 in the Toolbox would go to `!cc01` in Kyma.

The Kyma sliders can be configured to send data to various controllers in Kyma. In the Kyma sliders window pictured above, four sliders are available: sliders 1-4. The Kyma sliders in the Toolbox do not need to be consecutive values: controllers 1, 7, 10, 23, and 45 could be specified in the configuration.

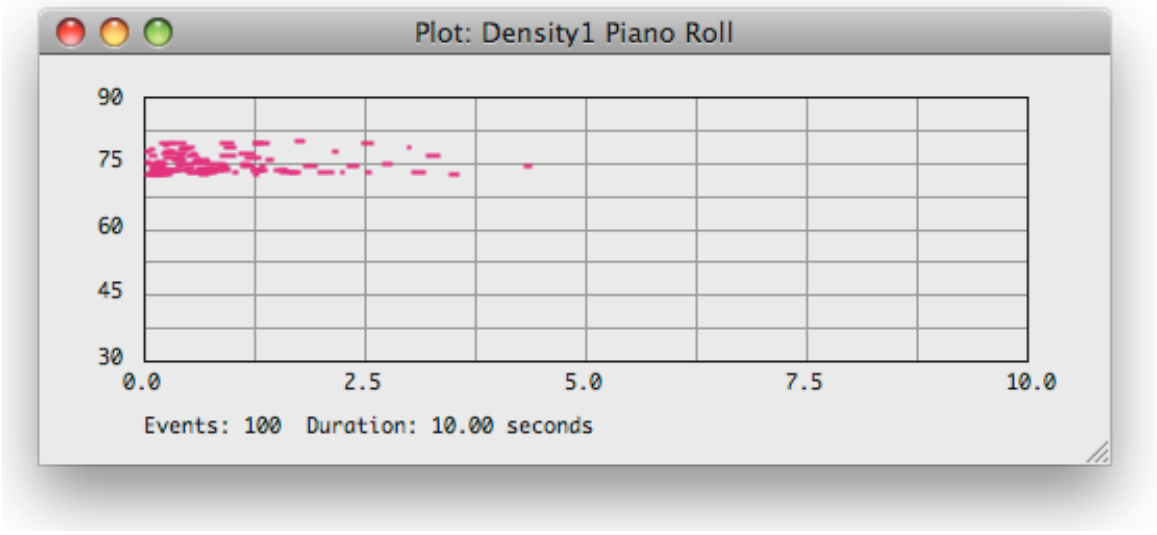
Kyma sound *25 sines* uses `!cc01` for MixAmp and `!cc02` for Gain. Once an object has been played, the values for the continuous controllers in Kyma can be controlled from within the Toolbox with the Kyma sliders.

Controlling density
Example *density1* is a density section where the length of time for the section is chosen with a generator to be between 5 and 12 seconds. In that time, 100 events will be created, starting at times chosen with the generator in the box for *Attack*. This generator should produce values between 0 and 100 percent of the duration of the section. At each of these points, a new event will be started. (Density sections are discussed in more detail in Tutorial 8).



Density1 uses an exponential generator for the attack percentages. This generator will tend to produce values near 0 (the beginning of the section). The spread is determined by the gamma parameter that in this example is made once per section as a random value in the range .001 to .1. The first value means a rather large spread of values. The second limit would focus things near the beginning of the section.

Durations are random in the range 50-200 milliseconds. Pitches are floating-point numbers (since the pitch threshold is a floating-point number in the range 70.0 - 75).

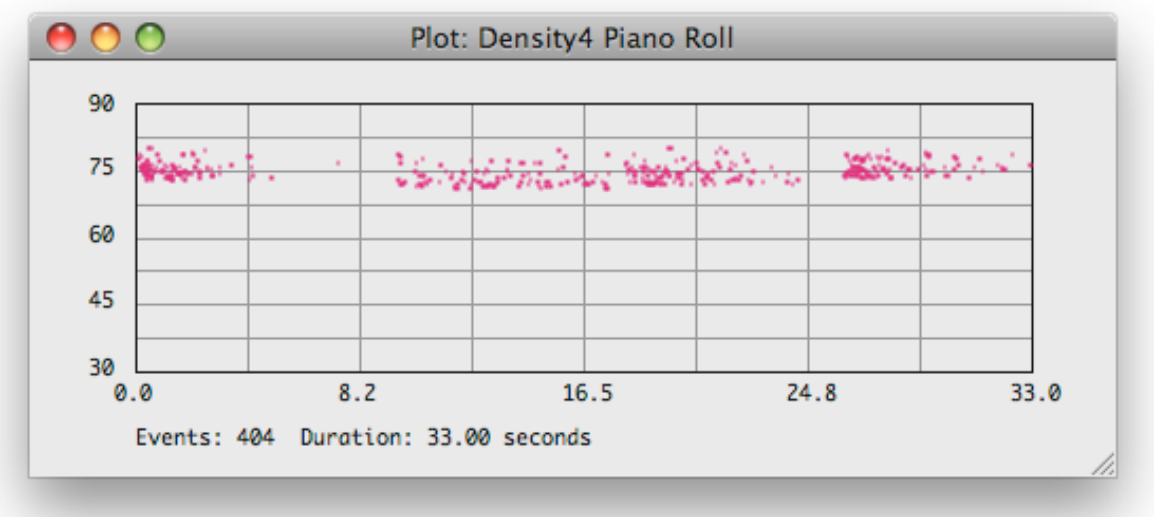


Since Kyma sound *25 sines* allows no more than 25 oscillators at a time, if 25 oscillators are already sounding and the Toolbox needs another one, it will steal the oscillator that has been on the longest.

Several variants of section *density1* can be generated and joined in succession. This is done with *Generate Sequential Section*. Section *density4* contains 4 variants of *density1* with an offset time in seconds between them. This offset time is chosen with the generator *random-choice*.

• Make and play *density4*.

A representation of a possible outcome of making *density4*:



Midi controllers

Midi controller values sent to Kyma can be floating-point numbers in the range 0-127. Currently, the controllers in Kyma must be named `!cc01`, `!cc02`, etc. In the AC Toolbox this would be controller 1, controller 2, etc.

• Compile, load, and start the Kyma sound *for controller examples*.

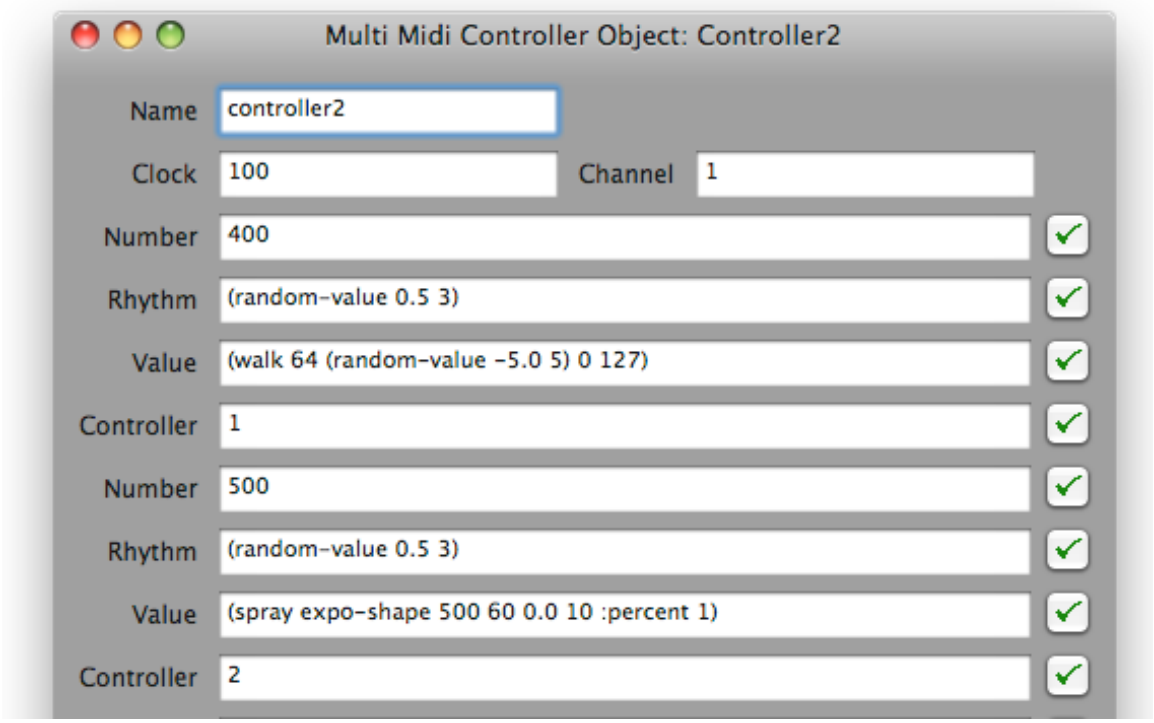
The example Kyma sound is a SampleCloud with a few minor changes. Some hot values have been replaced with controller values such as `!cc01`. `!TimeIndex` that controls the read point in the sample file was replaced with `!cc01`. `!GrainDur` that controls the duration of the grain was replaced with `!cc02`. `!Amp` became `!cc03`. These values can be controlled in the Toolbox using Kyma sliders.

Midi controllers in the AC Toolbox are controlled by Midi Objects. These objects are controller density, controller data, multi-midi-controller-object, etc. These objects should not be confused with the AC Toolbox controller object that does something different. See Tutorial 11 for a discussion of Midi Objects in the Toolbox.

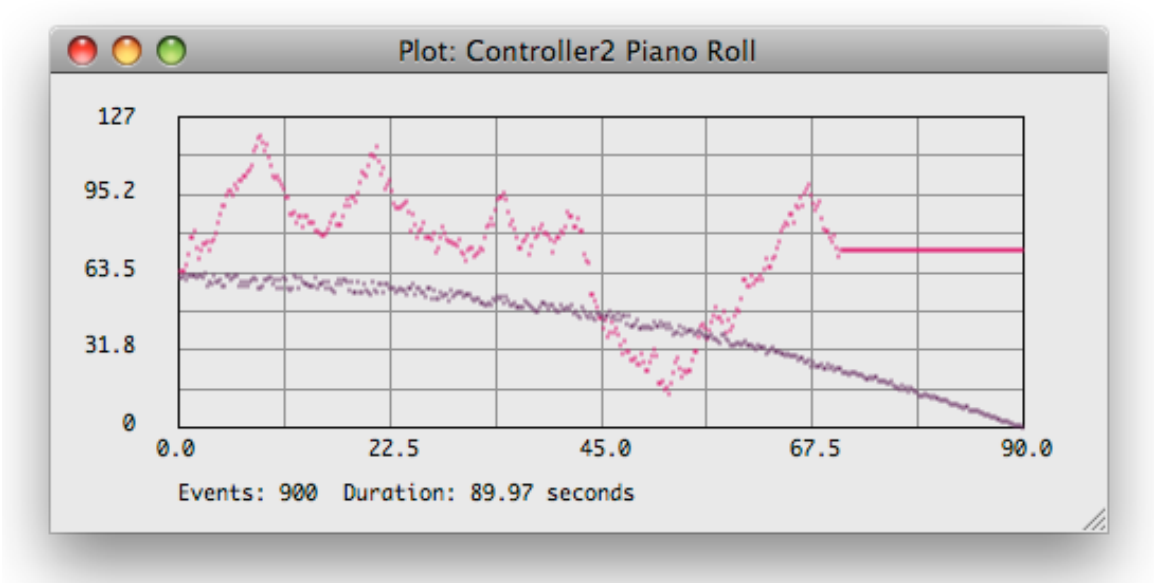
In the examples, object *controller1* will send 400 values to controller 1 (which is `!cc01` in the Kyma sound). The clock time is 100 milliseconds meaning that a value will be sent to controller 1 every 50-300 milliseconds as determined by the random generator for the rhythm parameter. The controller values are determined by a random walk.

• Make and play object *controller1*. Note that the Kyma sound does not stop after the performance of the object. This will be dealt with later. The end can be seen when the *Stop* button returns to *Play*.

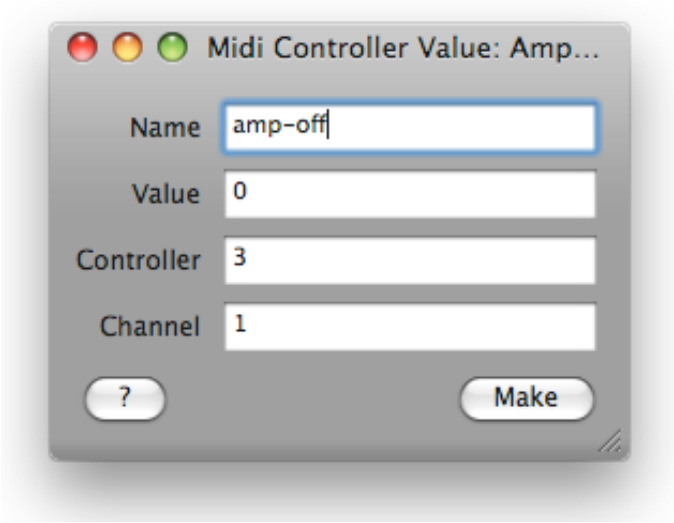
Controller2 expands on what happened in *controller1*. In addition to sending values to controller 1, values are also sent to controller 2. This controller receives values deviating randomly from a descending exponential curve.



• Make and play object *controller2*.

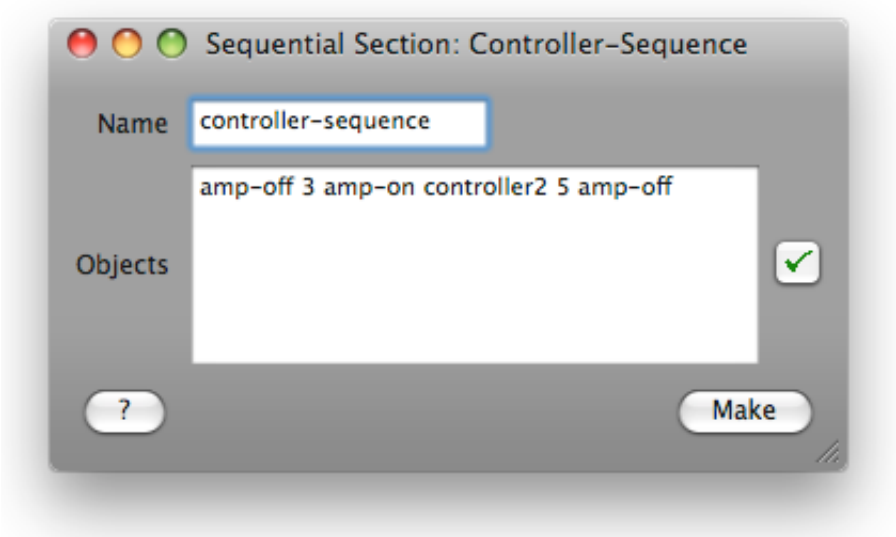


In order to set the amplitude of the Kyma sound to 0, controller 3 should be sent a 0. This can be done with a Controller Value object.



Similarly, an initial value can be set with the object *amp-on*.

The objects *amp-off, amp-on, and controller2* are joined in sequence in a sequential section named *controller-sequence*. A time in seconds is included to indicated a delay before performing the next object. The delay is optional.



Playing *controller-sequence* sends a 0 to the amplitude controller in Kyma, waits 3 seconds, sends a value 64 to turn on the amplitude, plays object *controller2*, waits 5 seconds to allow the last controller value a time to sound, and then sets the amplitude to 0.

Program changes

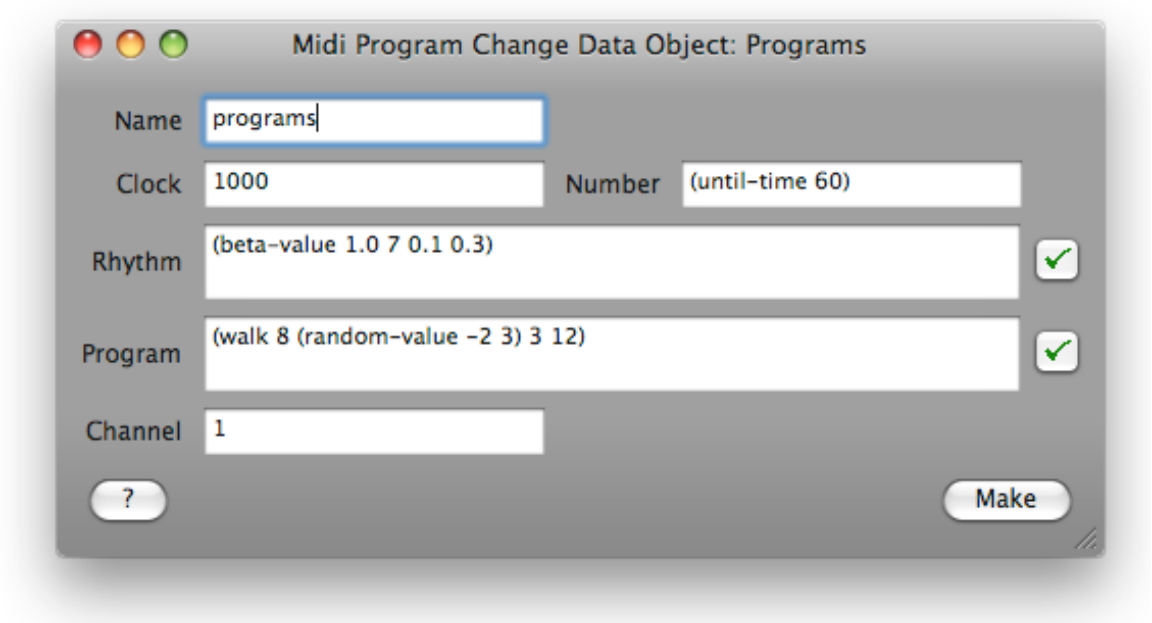
In the AC Toolbox, program changes are specified with Midi Objects such as program change, program density, and program data. A discussion of their usage is found in Tutorial 11.

In Kyma, program changes can be used to select presets of the virtual control surface. The examples in this tutorial limit themselves to this use.

• In Kyma, the VCS Program Change Channel should be set to 1. It is found via the **DSP>Configure Midi** menu item.
• In Kyma, compile, load, and start the sound *for program change examples*.

The sound *for program change examples* is a GrainCloud with a VCA to control the total amplitude with controller `!cc03`. The virtual control surface contains a number of presets. Choices will be made in the Toolbox from program numbers 3 - 12 that correspond to Kyma presets 3 - 12.

A program data object such as *programs* allows the choice of presets and the speed at which that occurs to be algorithmically controlled. In *programs*, rhythm is controlled by *beta-value* that will tend to produce values near 1 second and near 7 seconds. Program numbers are chosen with a random walk bounded by 3 and 12 that are the limits of the presets to be chosen in Kyma. Program numbers are chosen until 60 seconds have been filled.



Object *programs* is combined with the *amp-on* and *amp-off* objects in a sequence called *program-sequence.* As is the case with *controller-sequence*, an amp-off is sent, followed by a delay of 3 seconds, an amp-on, *programs*, a delay of 2 seconds and an amp-off.

• Make *programs* and *program-sequence*. Play *program-sequence*.

Programs2 is a faster version of *programs*. It is included in *program-sequence2.*

• Make *programs2* and *program-sequence2*. Play *program-sequence2*.

Summary

Menu items
Midi setup, data section, kyma sliders, density section, generate sequential section, generate parallel section, controller data, multi controller, program data

Tutorial 20

Reading spectral data from other sources

The AC Toolbox does not have any functions for doing spectral analysis of sound files. It can read in two types of spectral data from other programs. This data can then be manipulated or examined in the AC Toolbox.

The first spectral type is a composite (static) spectrum. This spectrum provides one chord or set of pitches derived from the analyzed sound. Data for this type is read with the tool *read-spectrum-file*. It can be made with the programs SPEAR, Hetro, Audacity, or Amadeus.

The second type of data contains amplitude and frequency tracks. This provides values that change over time but are also more difficult to manage. This type of data can be read with the tool *read-tracks-file*. It can be made with SPEAR or Hetro.

SPEAR is an analysis/resynthesis program that allows the selection, editing, and manipulation of partial tracks. These tracks can be exported as *text – partial* data to a text file and used in the AC Toolbox. The analyzed sounds can be quite long.

Hetro is the heterodyne filter analysis program that is part of *Csound* and accessible via the **Csound File Options** dialog if Csound5 has been installed. The analysis file will contain frequency and amplitude tracks for a number of partials. This file can be no longer than 32 seconds because of limitations in the Hetro file format.

Audacity is an open source audio editor. A spectrum can be exported in text format for use in the AC Toolbox.

Amadeus is a shareware program. A spectrum analysis can be saved in the format *TEXT (not importable).*

These options provide means to explore spectral data as a possible source of musical material. Both are insufficient for a fully-automated, mechanized approach to composition that inputs a sound file and expects a finished musical composition as the output. Instead these two approaches assume the data will be crafted, explored, and transformed as part of a compositional process.

Spectrum files

An analysis of a bassoon sample will be used in this tutorial. The sound file for this bassoon sample can be found in the *Support/FileExamples* folder.

A good source of other instrumental samples for analysis can be found at the University of Iowa Electronic Music Studios.

Read-spectrum-file uses *spear* as the default file type. The AC Toolbox will reduce the SPEAR track data to one pitch per track.

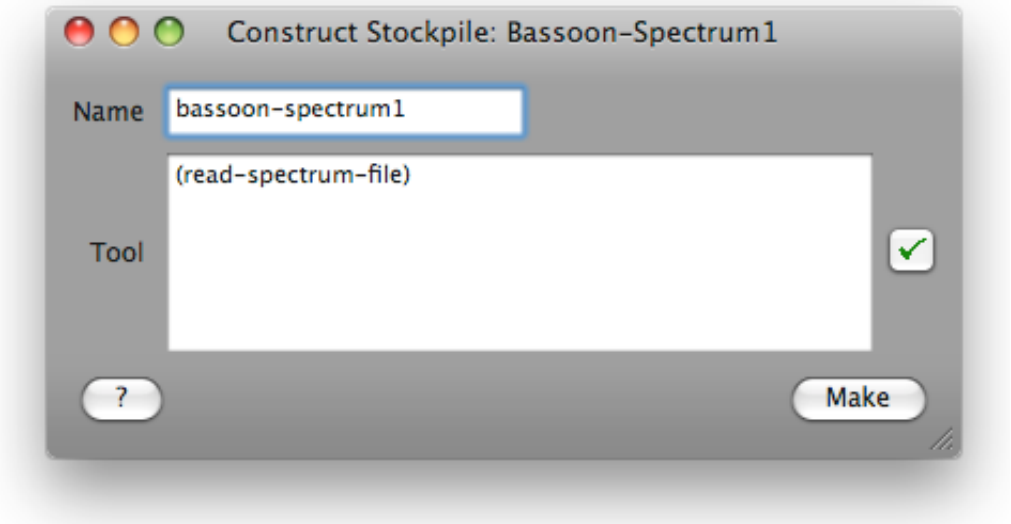
To produce data with SPEAR that is suitable for use in the AC Toolbox:

Open a (mono) sound file and make an analysis
Select tracks with the mouse while holding down the shift key
Select File>New From Selection
Select File>Export Format>Text – Partials
Select File>Export

It is important the SPEAR data is exported as a text file using the 'text – partials' format.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 20 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

The file *test spear bassoon.txt* from the *Support/FileExamples* folder should be read to a stockpile so it is available for frequent use. Object *bassoon-spectrum1* does this. *Read-spectrum-file* imports the text. The frequency data is converted to floating-point Midi values and by default is not rounded to semitones, quarter-tones, etc.

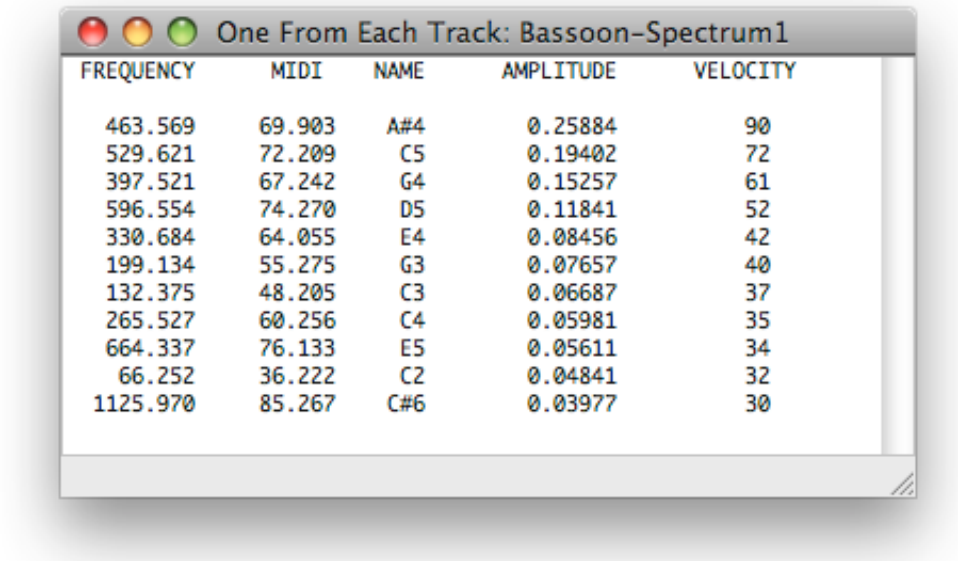


Read-spectrum-file assumes that the file being read was made with SPEAR. If Audacity is the source, the form should be:

```
(read-spectrum-file :type 'audacity)
```

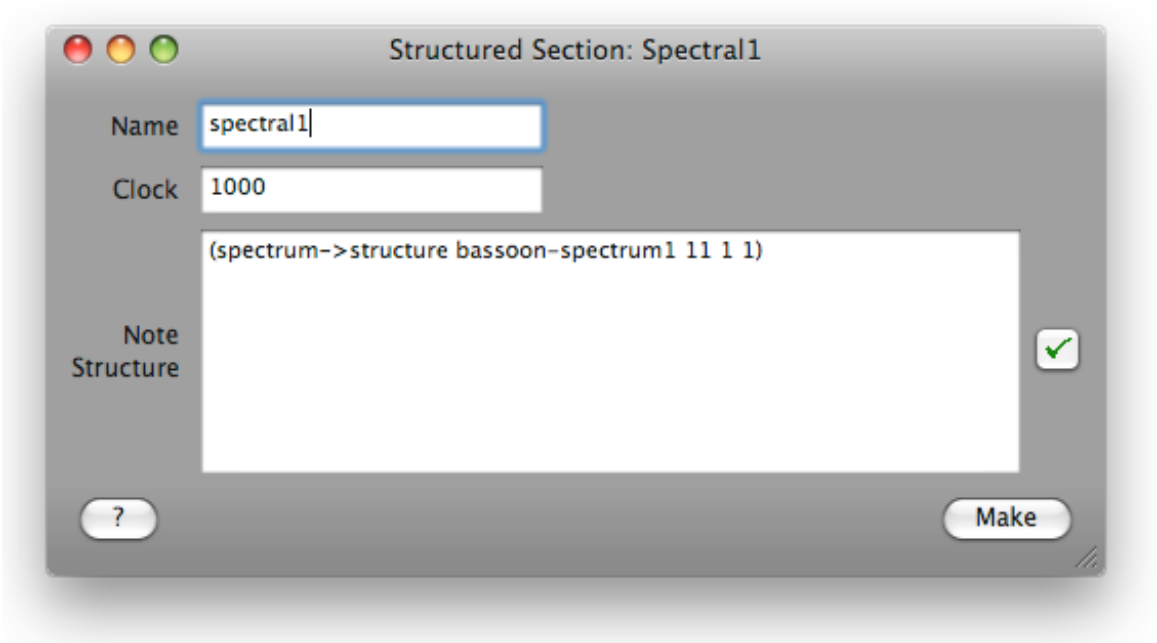
The spectral data and its translation can be written to a window by typing in the Listener (**Other>Listener**):

```
(spectrum->window bassoon-spectrum1)
```



Chords

Spectral1 is a section with one chord. Each pitch value and its corresponding amplitude value from the spectral peaks stored in *bassoon-spectrum1* are included in one chord.



The parameters for the tool *spectrum->structure* are

```
spectral-data number duration channel &key low high low-amp high-amp
```

Spectral data in this case is the name of the stockpile with the data. *Number* is the number of peaks to use. It should be <= the number of peaks in the stockpile. *Duration* is a rhythmic value. If *Channel* is one value, all peaks are sent to the same Midi channel. If it is a list, generator, etc., each peak goes to a different channel.

The pitches are expressed as fractional Midi note numbers. If the Midi output destination is QuickTime or Capybara/Pacarana, the Midi fractions can be heard. For all other destinations, playback will round off the Midi note numbers to integers.

Bassoon-spectrum2 reads the file again, but rounds the pitch data to semitones.

```
(read-spectrum-file :round 1)
```

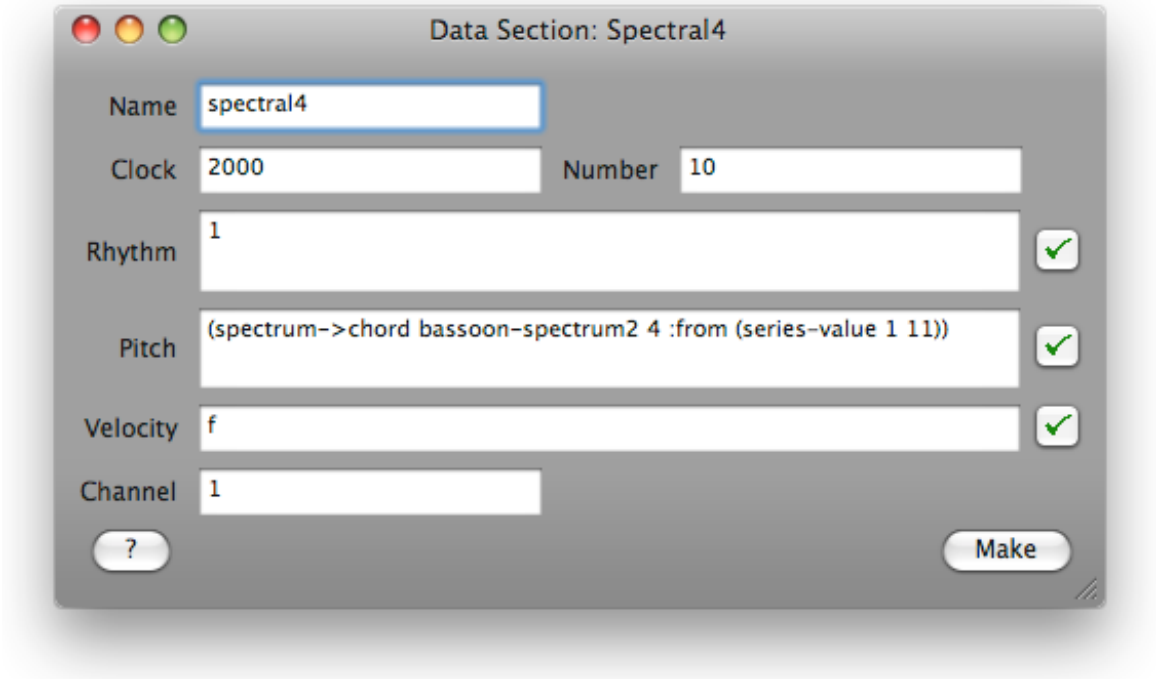
Spectral2 produces a chord using this rounded stockpile. Note that spectral data can only be rounded in *read-spectrum-file*, not in the various tools that use this data.

Spectral3 limits the number of peaks to 5.

Spectrum->chord is a generator that returns a chord each time it is applied. It only returns pitch values and not amplitude values. The input parameters for this generator are:

```
spectral-data number &key from name sort low high peaks
```

Spectral4 produces 10 chords using *spectrum->chord*. 4 pitches are chosen for each chord. The peak numbers included in each chord are determined with the generator *series-value*. Peak numbers refer to amplitude peaks. The peak with the largest amplitude is peak 1.



Spectrum->chord can also map the pitches in a spectrum to a different range. The new range can be specified by binding values to the keywords *low* and *high*. *Spectral5* maps the pitches to the range c4-b5. Note that since the entire spectrum is mapped to the new range and each chord only contains part of the spectrum, the entire range may not be present in one chord.

```
(spectrum->chord bassoon-spectrum2 11 :low c4 :high b5)
```

When pitches are mapped to a new range, they are rounded using the same unit used with *read-spectrum-file*. In this case, the output values are rounded to semitones.

Spectral6 chooses the lower limit by applying a generator once per section to determine a value.

```
(spectrum->chord bassoon-spectrum2 11 :low (make (random-value c3 c4))
:high b4)
```

Spectral7 generates a sequential section consisting of 5 variants of *spectral6.* The 11 peaks of the bassoon spectrum will be mapped to different ranges in the successive chords.

Pitches

Spectrum->pitch returns a list of pitch values from the spectrum. The parameters are:

```
spectral-data &key name sort low high peaks
```

Spectral8 uses this tool to produce the pitches in the section.

```
(spectrum->pitch bassoon-spectrum2)
```

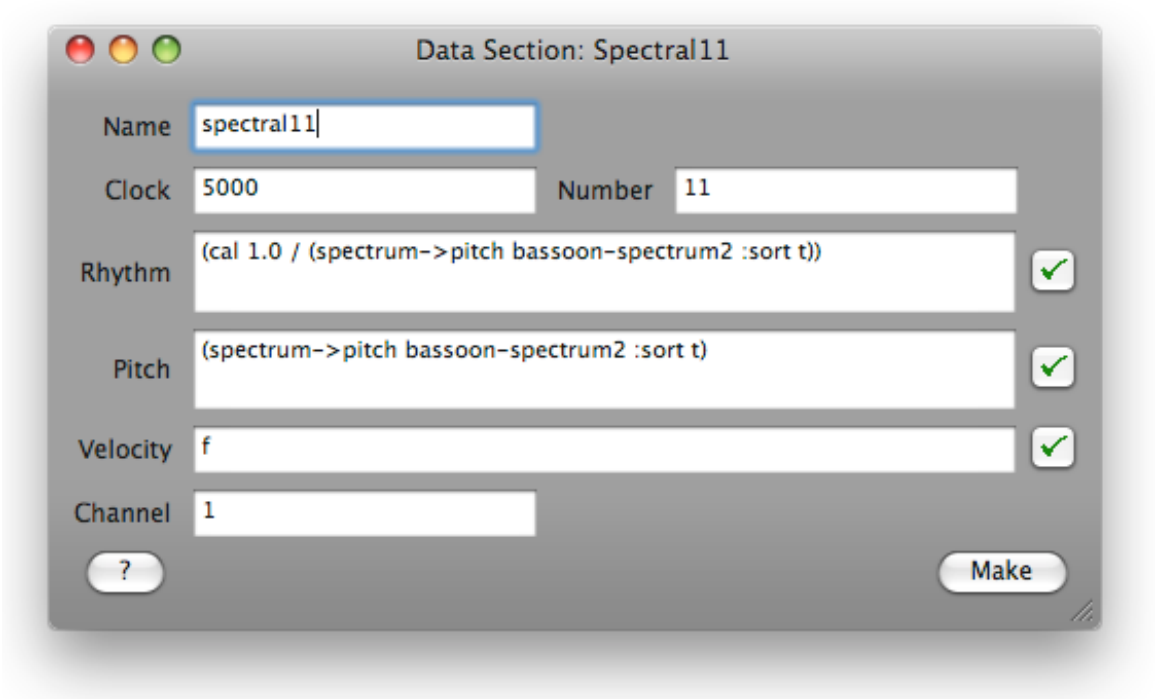
The pitches come one after another instead of being a chord. The order of the pitches is from those with the highest amplitude values to the weakest values. If the keyword *sort* is bound to t, the pitches occur from the lowest pitch to the highest. *Spectral9* does this.

```
(spectrum->pitch bassoon-spectrum2 :sort t)
```

The list produced by *spectrum->pitch* could also be used as the stockpile for a *choice* generator. In *spectral10*, a random-choice is made from the available pitches.

```
(random-choice (spectrum->pitch bassoon-spectrum2))
```

Spectral11 is a simple example of relating spectral data to rhythm. Rhythmic values are calculated to be the inverse of the fractional Midi note number. This is done with the tool *cal* that allows calculations using AC Toolbox data to be specified from left to right (infix notation).



The result of using cal for *spectral12* is:

```
1.0/pitch1, 1.0/ pitch2, 1.0/pitch3, ...
```

Spectral data may have duplicate peaks depending on the choices made in SPEAR or the result of rounding the pitches. *Voice1* should be used to read file *test spear voice.txt* found in the *Support/FileExamples* folder. When this data is printed using

```
(spectrum->window voice1)
```

duplicate pitch values can be seen. To eliminate duplicates when reading the spectrum file, bind the keyword *unique* to *t*. This is done in *voice2*. In the Text Output window, you can see that were 24 unique pitch values found from the initial 30 that were read.

If the number of peaks are forgotten (for any reason) and the Text Output window is no longer available, the tool *peaks?* can be used in the Listener to return the number.

```
(peaks? voice2)
```

Other file formats

Text files containing composite spectral data (a table of frequency peaks with corresponding amplitude values) can also be produced in Audacity or Amadeus.

In Audacity, select the sound to be analyzed. A spectrum can be produced via the menu item **Analyze>Plot Spectrum**. This spectrum should be exported (click on the **Export** button) and saved as a text file. In the *Support/FileExamples* folder, an analysis of the bassoon spectrum using a Hanning window of size 2048 has already been made. This text file *test audacity bassoon.txt* can be opened for viewing in the AC Toolbox (File>Text>Open) and will show a text table of frequency and amplitude values. It can be read into the AC Toolbox with object *Audacity1.*

```
(read-spectrum-file :type 'audacity)
```

By default, 10 peaks will be returned. They will be the 10 peaks with the highest amplitude values. To read more peaks, bind the keyword peaks to a different number. This can be done with *Audacity2*.

```
(read-spectrum-file :type 'audacity :peaks 20)
```

When reading an Audacity or Amadeus file, the default is to return 10 peaks. For SPEAR and Hetro files, the default is to return all peaks.

In Amadeus, the menu item **Analyze>Spectrum** can be used to produce the analysis. This file should be saved as format *TEXT (not importable).* It can be read with

```
(read-spectrum-file :type 'amadeus)
```

The number of peaks can also be specified with the keyword *peaks.*

The preparation of data with the Csound utility Hetro will be discussed later in this tutorial.

Tracks

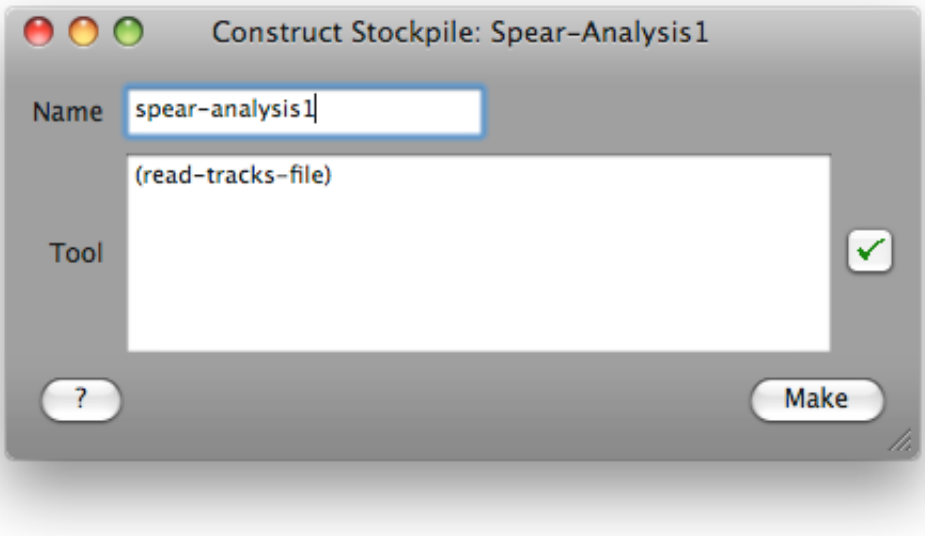
Some sound analysis programs produce tracks containing amplitude and frequency trajectories. The data from two of these programs, *SPEAR* and the Csound analysis utility *Hetro* can be read in the AC Toolbox and mapped to melodic, rhythmic, and dynamic values. The programs use different methods to produce the data, therefore the results differ as well. The data is read and manipulated with the same tools.

SPEAR
The analysis data from SPEAR has no fixed maximum length. This means rather long sound files can be analyzed and exported to the AC Toolbox. Instructions on preparing SPEAR data for use in the AC Toolbox can be found earlier in this Tutorial.

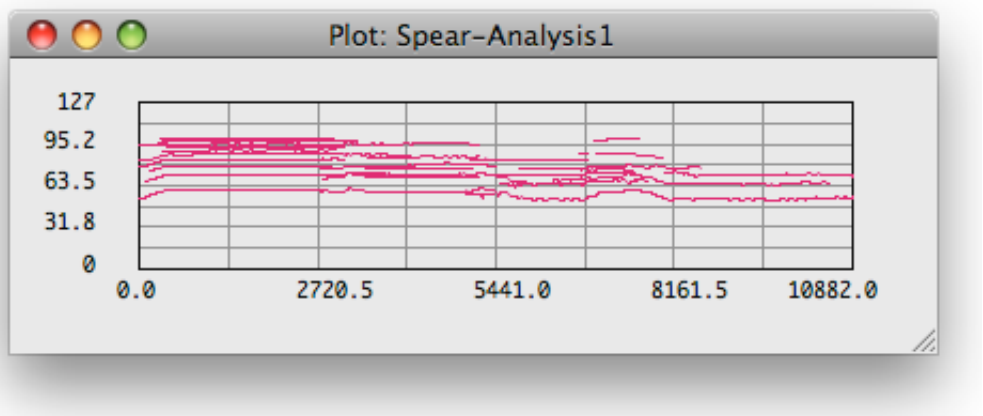
The *Support/FileExamples* folder contains a sound file with a fragment sung by Roscoe Holcomb. A portion of the SPEAR analysis of this file is called *test spear voice.txt* and can be used for the following examples. It can also be opened in SPEAR and resynthesized to allow comparisons with the Midi data that is produced.

Spear-analysis1 is a stockpile containing the data from an analysis file. The pitch data has not been rounded. The default file type for *read-tracks-file* is *spear*. Choose the file *test spear voice.txt*. The number of tracks read from the file will be printed in the Text Output window.

```
(read-tracks-file &key type round filter base average)
```



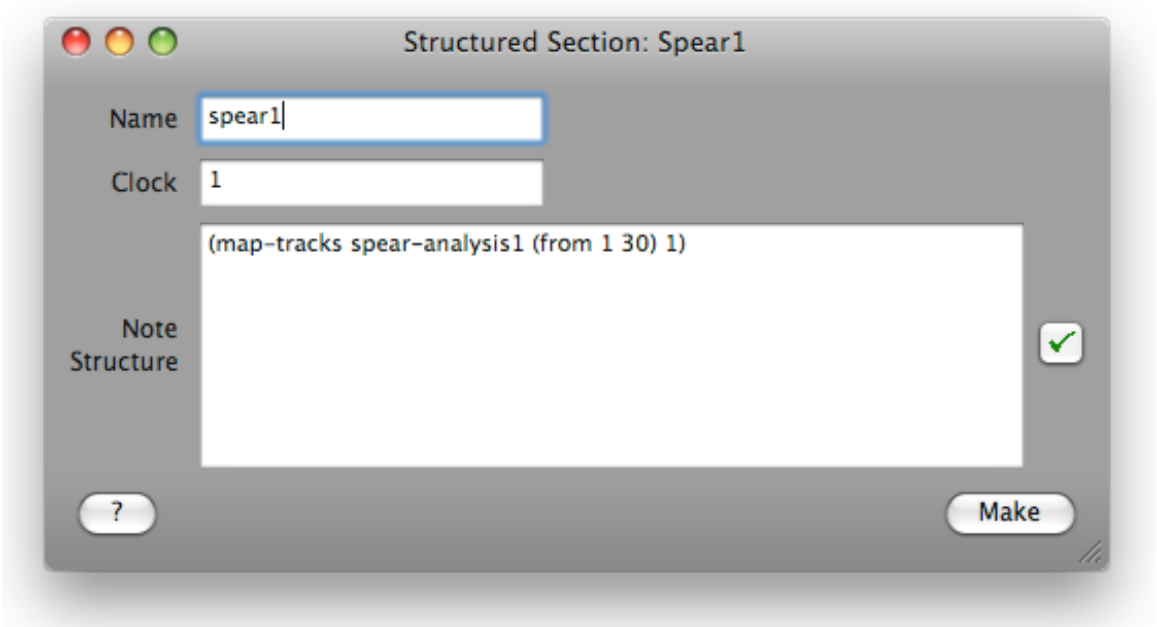
The stockpile *spear-analysis1* can be plotted via the Objects dialog.



Tracks can be selected from this data and mapped to Midi values. Tracks can be sent to the same Midi channel or each track can be assigned a different one. This is done with *map-tracks*. The parameters are

```
(map-tracks analysis tracks channel &key min-vel max-vel filter)
```

Spear-analysis1 is the name of the stockpile with the data. *Tracks* can be a list of chosen tracks. Tracks 1 –30 are chosen with `(from 1 30)`.Tracks 1, 3, 5, 7 could be chosen with the list `'(1 3 5 7)`. If all tracks should go to the same Midi channel, a constant value for channel can be given. Otherwise a list of channels should be provided.



Spear1 uses the fractional Midi values that were returned by *read-spectrum-file.* They can be heard using QuickTime.

Spear-analysis2 reads the file again, rounding the pitch values to semitones.

```
(read-tracks-file :round 1)
```

Spear2 uses this analysis stockpile with *map-tracks.*

Some tracks have very low velocity values. *Map-tracks* will map the values to be between *min-vel* and *max-vel*. The default Midi values for those parameters are 20 and 64. When only a few tracks are selected, it may be necessary to map the velocity to higher values.

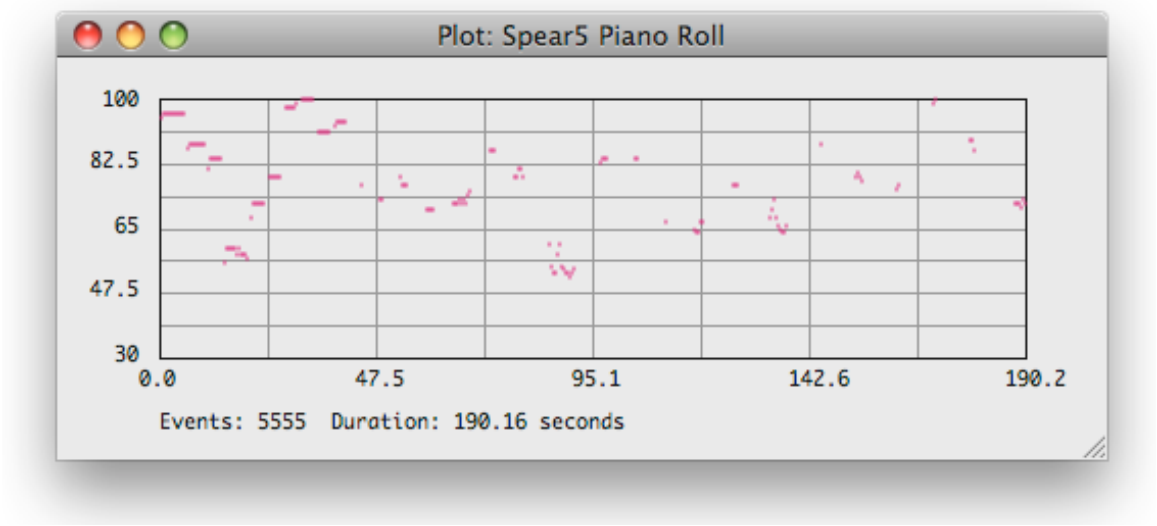
Spear3 uses tracks 1, 3, 5, 10, 20, and 30 and increases the minimum velocity value to 40.

```
(map-tracks spear-analysis2 '(1 3 5 10 20 30) 1 :min-vel 40)
```

To help in choosing tracks, it may be useful to listen to each track, one at a time. This can be done by defining a controller named *tracks* with the values from 1-30. A discussion of controllers can be found in Tutorial 13. Structured section *spear4* will take one value from *tracks* each time it is evaluated. The first time, the section should be *specified* (by holding the Command key while clicking on *Make*) instead of just *making* it. By specifying the section instead of making it, no value is taken from the controller meaning that the next section will start with track 1.

```
(map-tracks spear-analysis2 (take-one tracks) 1 :min-vel 40)
```

Finally *spear5* can be made. It creates a sequential section consisting of 30 variants of *spear4*. This will present the tracks one at a time.



The tool *pitch-track* will retrieve a list of pitches from one or more tracks of spectral data. Only the pitches are returned.

Stockpile *Pitches* uses this tool to make of list of the pitches from tracks 1, 3, 5, and 10. Pitch repetitions within a track are filtered.

```
(pitch-track spear-analysis2 '(1 3 5 10) :filter t)
```

The pitches from each track are appended one after another. *Spear6* is a section containing a random-choice from this pitch stockpile.

Read-tracks-file also has keywords *filter* and *average*. They will be discussed in the following section with the Hetro analysis examples.

To query a stockpile containing track data for the number of tracks, use the tool *tracks?*. In the Listener, type

```
(tracks? spear-analysis2)
```

Heterodyne

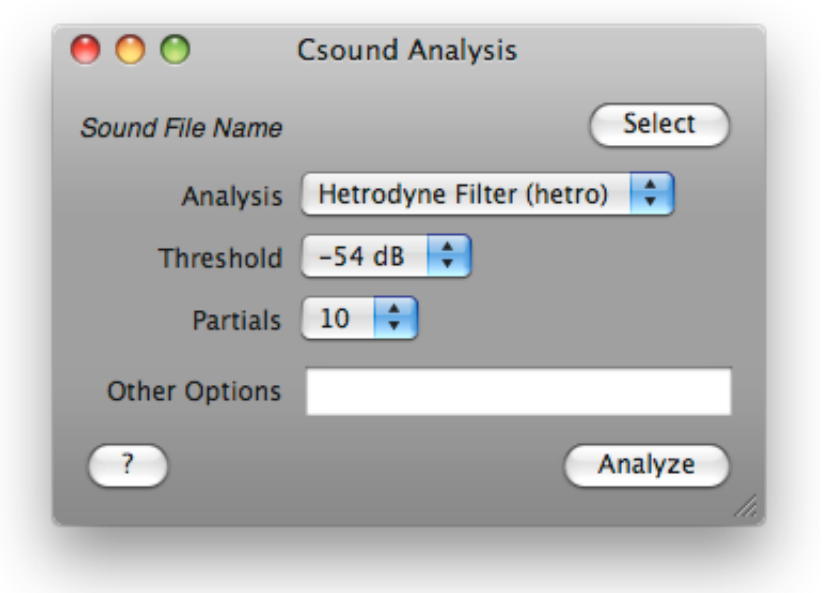
Heterodyne Filter analyses can be made with Csound. In Csound, the purpose of the analysis is to create frequency and amplitude envelopes for a number of harmonics in the sound file. Using an additive synthesis model, these envelopes can be used to resynthesize (part of) the analyzed sound.

In the AC Toolbox, each set of frequency and amplitude envelopes is mapped to fractional Midi note numbers and velocity values in the same way that the SPEAR data is mapped.

The heterodyne filter analysis in Csound is limited to a sound not longer than 32 seconds. This limitation comes from a feature of the Csound analysis file format. By default, *Hetro* (the name of the heterodyne filter analysis program for Csound) will return 10 harmonics. There are options for returning a different number of harmonics as well as various other possibilities. The Csound documentation should be consulted for these options. To use Hetro, Csound5 must be installed.

The tracks in the analysis made by Hetro tend to be continuous throughout a file. The tracks in SPEAR tend to be of a shorter duration.

Making a heterodyne filter analysis in Csound is a rather slow process. The file to be analyzed should be a mono file. In the **Csound File Options** dialog, click on the **Analysis Programs** button. The **Csound Analysis** dialog will appear. Choose *Heterodyne Filter (hetro)* in the pop-up menu. Adjust the amplitude threshold and partials if desired. **Select** the file to be analyzed (remember the 32-second limit) and then select the **Analyze** button.



The sample heterodyne analysis file called *test hetro (intel).het* or *test hetro (ppc).het* can be used for the following examples. It is found in *Support/FileExamples*. The file that corresponds to the processor being used should be chosen. It is currently necessary that the analysis be made on a computer with the same processor type (intel or ppc) as the one on which the AC Toolbox is running.

Hetro-analysis is a stockpile containing the data from an analysis file. The type should be specified as *hetro*. Pitch is rounded to quarter-tones.

```
(read-tracks-file :type 'hetro :round 0.5)
```

Map-tracks will map and select tracks from this stockpile in the same way as it did with data from the spear analysis. The difference is the nature of the analysis data.

Hetro1 maps tracks 1-10.

Hetro2 selects just one track (track 1) and adjusts the minimum velocity value.

```
(map-tracks hetro-analysis 1 1 :min-vel 60)
```

Hetro3 selects track 2 and adjusts the maximum velocity as well.

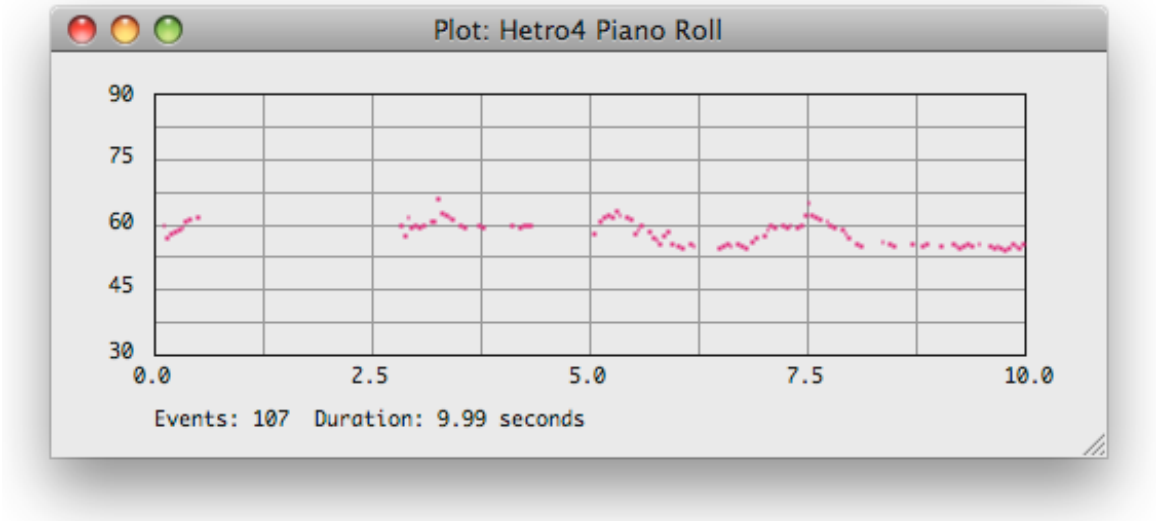
```
(map-tracks hetro-analysis 2 1 :min-vel 60 :max-vel 80)
```

There are still a large number of notes being produced. This can be seen by looking at the *Object Info* dialog (select *hetro3* in the Object dialog and then click on the *Info* button).

Hetro4 removes successive repetitions of the same note by binding the *filter* keyword to t.

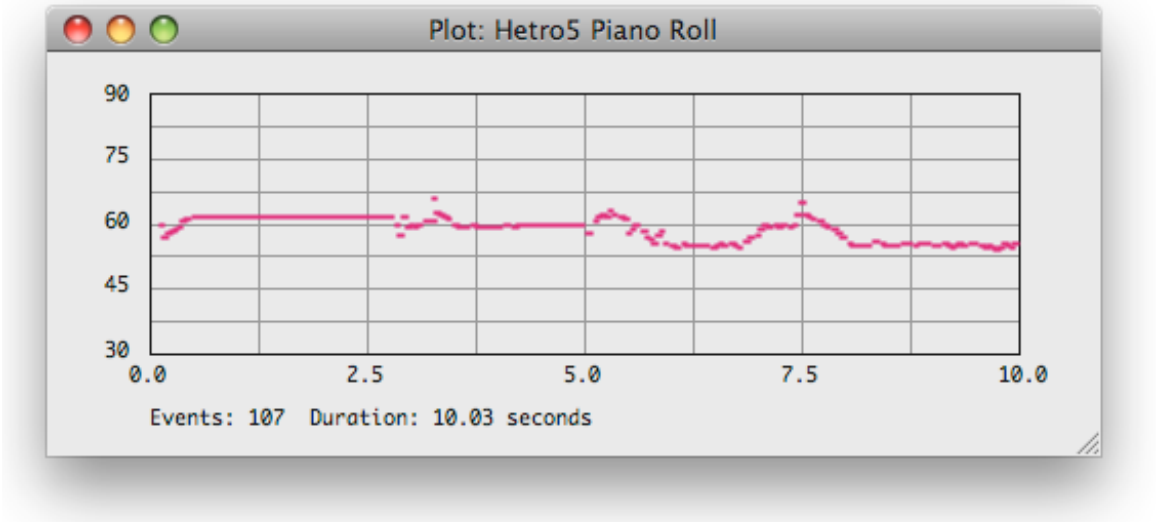
```
(map-tracks hetro-analysis 2 1 :min-vel 60 :max-vel 80 :filter t)
```

Rests now occur where previously there were repetitions.



These rests can be filled with a held note using the transformer *fill-gaps*. *Hetro5* uses the expression *(fill-gaps hetro4)* for the *duration* parameter to do this transformation.

Hetro5 now contains a melody corresponding more or less to one track of the heterodyne analysis.



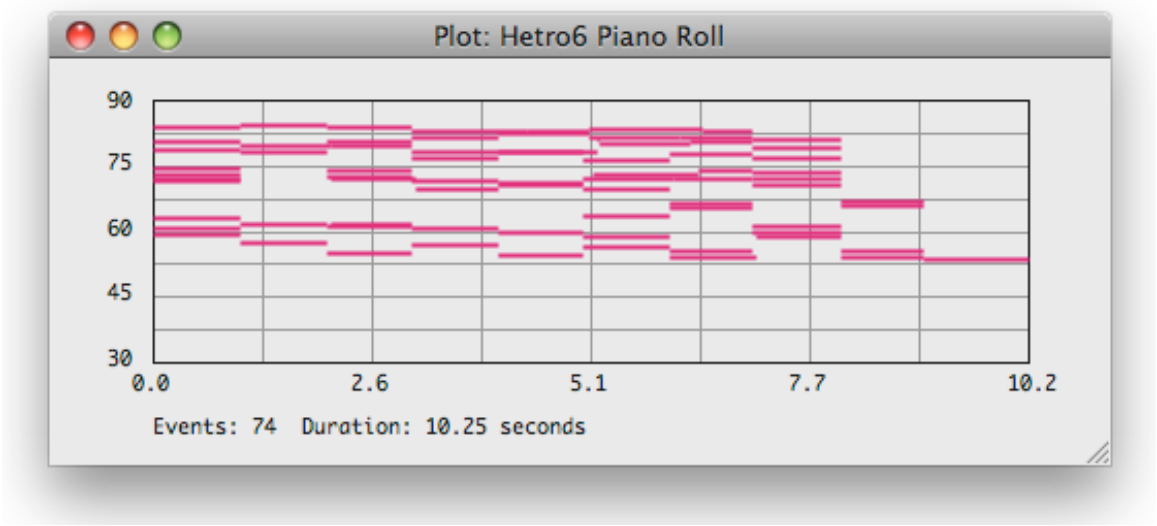
Track data from SPEAR or Hetro can be averaged within a specified time unit with keyword *average* in the *read-tracks-data* tool. Time is specified in milliseconds. The next averaged value is assigned the first start time within that unit. This reduces the amount of data produced. *Hetro-analysis2* averages the track data (per track) within 1000 milliseconds.

```
(read-tracks-file  :type 'hetro :round 0.5 :average 1000)
```

Averaging with a unit of 1000 milliseconds does not mean that every track will produce a value every 1000 milliseconds. If the next value in a track begins after 1110 milliseconds, that is when the new average value is returned.

Hetro6 uses this data with *map-tracks*. The pitch data is also filtered. Filtering data in *map-tracks* rather than in *read-tracks-file* is advantageous when averaging because it will also filter duplicate pitches in chords.

```
(map-tracks hetro-analysis2 (from 1 10) 1 :min-vel 60
:max-vel 80 :filter t)
```



Summary

Generators
spectrum->chord

Tools
read-spectrum-file, spectrum->window, spectrum->structure, spectrum->pitch, peaks?, read-tracks-file, map-tracks, tracks?, cal

Transformers
fill-gaps

Tutorial 21

Generating binary OSC files

SuperCollider is a language for sound synthesis with a separate synthesis server. The language sends messages to the server using OSC (OpenSound Control) – a popular protocol for communicating between computers and programs.

One of the features of the SuperCollider server is that it can take a binary file of OSC messages and use it to produce an audio file in non-realtime. In the language application, a synthesis definition (*synthdef*) is made, compiled, and written to disk. After that, the language application is no longer necessary. Another program such as the AC Toolbox can call the server, specify various options for the audio file and tell the server to produce an audio file based on OSC messages in a binary file.

In this way, the parameters of a sound model described as a synthdef can be controlled by a binary file of OSC messages.

With the AC Toolbox, it is possible to generate a binary file of OSC messages to control a sound model, have the SuperCollider server render these messages to an audio file, and then open the audio file for performance.

The syntax for creating binary OSC files in the AC Toolbox is similar to that for generating Csound score files. The difference is primarily due to the different data requirements of the two formats.

This tutorial does not provide information on how to program in SuperCollider. For that, the documentation for the language application should be consulted. This tutorial assumes that SuperCollider has been installed on the user's computer. The Toolbox will first look in the Applications folder for it. If not found there, the user will be asked where SuperCollider is.

SuperCollider can be downloaded from:
http://supercollider.sourceforge.net

Synthdefs
This tutorial will use four synthdefs that should be compiled in the SuperCollider language application and saved to disk.

The file containing the four synthdefs is called *test osc synthdefs.rtf* and can be found in the folder *Support/FileExamples.* If these synthdefs have not previously been compiled on the user's computer, the file should be opened in SuperCollider, and each synthdef written to disk. The language application can be closed after this.

Synthdefs intended for use with the AC Toolbox should have a duration argument. It does not matter what this argument is called but it should free the synthdef node after some amount of time. This can be done for example by specifying *doneAction: 2* for an envelope.

Options
In the **Other** menu in the AC Toolbox, there is an item for **OSC File Options**. This dialog allows various options to be specified. The options basically fall into two categories.

One category deals with rendering the OSC binary file into audio. The audio header format, the sample format, the number of channels, and the sample rate for the output file can be specified. An audio input file (used by the UGen SoundIn) can also be chosen. Files that will be read into buffers do not need to be specified in the *OSC File Options* dialog.

The second category deals with the way OSC files are named. It is possible for file names to be automatically generated, starting with a prefix supplied in this dialog and followed by consecutive numbers, e.g. *actSC1.osc*, *actSC2.osc*, etc. The rendered audio file will named *actSC1.aiff* etc. if *Aiff* is chosen as the audio header. The user will be asked for a folder for

writing the files. The choice of folder can be changed via *Output Directory*. The numbering of the files begins at 1 for each session.

If the files are not named automatically, the user will be asked for the file name. The audio file will use that name with an extension corresponding to the audio header type.

When an audio file is rendered, it can be opened in another application for playback. The default situation is to use the program that the system uses to open a file of that file type. For AIFF and WAV files, this is probably iTunes. It is also possible to specify another application to use to open the file by selecting *Choose Application* in the popup menu for *Open Audio File with* in the OSC File Options.

The *Render File* button will render an existing OSC file that was made with the AC Toolbox. This allows rendering an OSC file with different options.

Two ways to generate OSC data

There are two approaches to generating data for a binary OSC file in the AC Toolbox. The first way is to take a section that has already been made and convert the rhythm, pitch, and velocity data to OSC messages to control one synthdef. The second way is to create an OSC score object using generators etc. to create the parameter values for the synthdef. In this case, the OSC score object can be saved with the environment, combined in sequence and parallel with other OSC score objects, transformed, and filtered in a fashion very similar to Csound objects.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 21 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make a binary OSC file from the object, click on *Apply* in the dialog box.

Converting a section to an OSC file
When a section is converted to an OSC file, the OSC data is not saved as a separate object.

• Make *section1* from the *Tutorial 21 Examples* file This is a data section and can be played and plotted within the Toolbox in the usual way.

This section will be converted using synthdef *sine1* that should have been written to a file in SuperCollider. This synthdef is a simple sine generator with a three-part envelope: attack, steady state, and decay. The attack and decay are expressed as a factor of the total duration in the range 0-1.

The arguments for the synthdef can be seen in its definition:

```
(
SynthDef("sine1",{ arg freq = 440, amp = 0.2, dur = 1.0, attack = 0.25,
decay = 0.25;
  var ss = 1 - attack - decay;
  Out.ar(0,
          SinOsc.ar(freq, 0,amp)
                * EnvGen.kr(Env.linen(attack,ss, decay,1), timeScale:
dur, doneAction: 2)
  )
}).writeDefFile;
)
```

• After *section1* has been made, select menu item **Tools>Section->OSC File**.

• *Section* is the name of the section to be converted.

• *Synthdef* should be *sine1* for this example.

• *Include* can be left blank (it's use will be explained later in this tutorial).

• *Amp Name* is the name of the amplitude argument in the synthdef. For *sine1* this is *amp*.

• *Max Amp* is the value that Midi velocity 127 should receive. This is not the total amplitude of all layers, but the value each event will receive if it uses velocity value 127.

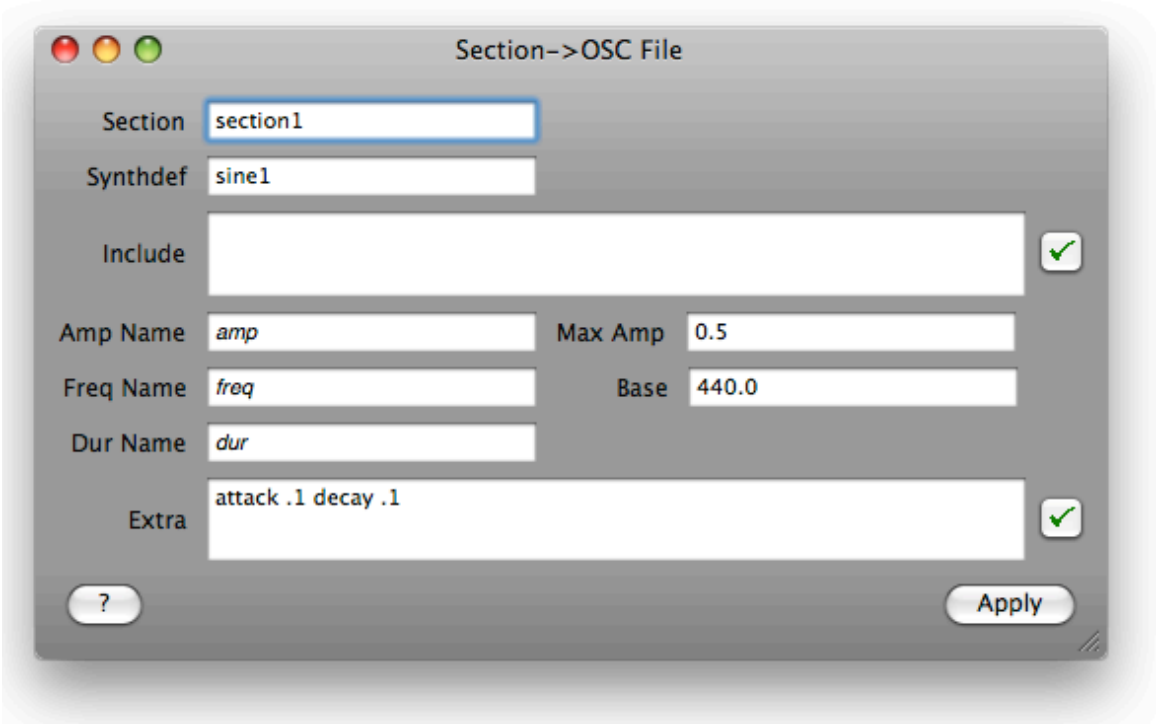
• *Freq Name* should be the name of the frequency argument in the synthdef. For *sine1*, it is *freq*. The AC Toolbox will convert the Midi note numbers stored in the section data to frequency values for this parameter. In the Toolbox, Midi note numbers can be floating point numbers.

• *Base* is the frequency value for *a4* (note 69) and is used for the frequency calculations.

• *Dur Name* should be the name of the duration argument. For *sine1*, it is *dur*.

• *Extra* provides room for extra pairs of arguments and values to be entered. It can be left blank or filled with as many pairs of argument names and argument values as desired. To limit the attack and decay of the envelope each to 10% of the duration, enter:

```
attack .1 decay .1
```



To calculate the OSC messages and write them to a binary file, select *Apply*. When the file is written, it will render the file to an audio file, and optionally open the audio file.

If a new audio file is to be generated with the same name as one that is open in an application, the open file should be closed before generating the new one.

The SuperCollider server returns messages concerning the rendering. These messages will be printed in the Text Output window. It is possible to turn off the printing of these messages in the *OSC File Options* dialog.

Using an OSC Score object
An *OSC Score* object makes an algorithm that can be used to write a binary OSC file. The score object does not save the data generated, just the input specification for making the file. This is similar to what happens with Csound objects.

If an OSC score object is *specified*, the input is saved and the object name added to the environment. When an OSC score object is *applied*, the input is saved, the object is added to the environment, OSC messages are generated and written to disk, the binary file is rendered to audio, and an application is opened to play the audio file.

If a score is applied a second time, a different set of OSC messages could be generated if one or more random generators was used in the specification.

Name is the name of the OSC score object to be made.

Synthdef is the name of the desired synthdef. It should be in the default synthdefs folder of SuperCollider.

Layers is the number of parallel applications of the input.

Include is for specifying additional bundles with OSC messages. An example of this usage will be given later. It can be left blank if it is not needed.

Number is the number of messages to be generated. Each message will contain a start time, a duration value, and argument names and values as specified in this dialog.

Start is the start time of an event. If this value is a constant, it is the time of the first event. Subsequent events will be the previous start time plus the duration of the previous

event. If *Start* is a generator, list, etc. it will be used to calculate all the necessary start times, regardless of their duration.

The columns for *Argument Name* and *Argument Value* can be filled with argument names from the synthdef and something such as a generator to calculate values for those arguments. The argument names must be the same names as in the synthdef. Argument names are case sensitive.

Duration: the name of the duration argument and something to generate its value *must* be entered. Duration values are expressed in seconds.

Other argument names and values may be left blank or filled as desired. If an argument name is entered, an argument value must be entered.

Extra can be left blank or contain as many additional name and value pairs as desired. The names should be argument names from the synthdef. The usage is the same as in converting a section to a OSC file.

Per Layer can be left blank or contain the name of one or more objects that should be remade before each layer is generated.

Per Score can be left blank or contain the name of one or more objects that should be remade each time this score is applied.

Examples with sine1

Wsine1 will produce 50 events for the synthdef *sine1*. The first event starts at time 0. The next event will start when the first event is finished. *dur* is the name of the duration argument. It will receive a random-value in the range 0.05 to 0.1 seconds. *amp* is the parameter for amplitude. It is assigned the value 0.3. Note that if an synthdef argument is not used in the OSC Score object, the synthdef argument will use it's default value. *freq* is the name of the frequency parameter. It will receive values returned by generator *shake*. In this object, *shake* will fluctuate around 500 Hz. The range of deviation is ± 1 %.

If *Specify* is selected, the input specification is saved, but the binary file is not written. If *Apply* is selected, the file will be written. Depending on the options chosen in the *OSC File Options* dialog, the file may also be rendered, and the resulting audio file opened.

If a duration parameter is a negative value, a rest is made with a length corresponding to the absolute value of the duration. Any other data (such as for frequency) that would be sent at that time is discarded. To prevent frequency data from being discarded, use the generator *skip-rests*. See the help in the application for a description.

Layers
Wsine2 produces 10 layers of a specification similar to that of *wsine1*. Layers mean that 10 applications of the input in this dialog will be added together and start at the same time. Each generator will be evaluated and start from the beginning again. A generator will not remember what it did in the previous layer.

```
Synthdef:  sine1
Layers:    10
Include:
```

```
Number:     50
Start:      0
Duration:   dur         (random-value 0.05 0.1)
            amp         (scale-by-layers 0.3)
            freq        (shake (make (rv 100.0 2000)) (rv -1.0 1) percent
t)
Per Layer:
Per Score:
```

For frequency, *shake* chooses a center frequency once per layer. The expression *(make (rv 100.0 2000))* applies generator *rv* (a shortcut for *random-value*) once each layer. If *make* was not used, *rv* would pick a new center frequency for each message.

When several layers are added, it is necessary to scale the amplitude. This can be done with *scale-by-layers* that scales the amplitude by the number of layers or by an optional factor times the number of layers. The shortcut for *scale-by-layers* is *sbl*.

Layers are calculated consecutively. The tool *layer-number* will return the number of the current layer being calculated. This information could be used to choose different parameter values for different layers. An example can be found in the help for *layer-number*.

Filling time

Instead of specifying a number of events to be generated, it is possible to specify an amount of time in seconds to be filled. This can be done with *until-time* and works the same way it does with other objects. Durations will be chosen until the desired amount of time is filled. Each event is allowed to finish, so the total time of the object may exceed the requested duration. *Until-time* can only be used in an OSC Score object if *start* is a constant value. If another form needs to know how many events were generated with *until-time*, the form *(from-number)* can be used to get the number.

Wsine3:

```
Synthdef:   sine1
Layers:     1
Include:
Number:     (until-time 50)
Start:      0
Duration:   dur         (random-value 0.05 0.1)
            amp         0.3
            freq        (masks&values 1000.0 100 (rv 50 100)
                          (rv 500 2000.0) (rv 100.0 1000))
            attack      0.1
Per Layer:
Per Score:
```

Fifty seconds will be filled. Frequency uses *masks&values* to generate continuously changing masks and select values within these masks. Note that the *attack* argument has been adjusted to be 10 % of the total duration. *decay* still uses the default value.

The shortcut for *until-time* is *ut*. The shortcut for *from-number* is *fn*.

Using a generator for start times

In the previous examples, *Star*t has had a constant value, meaning that each following note will start when the previous one stopped. It is possible to use a generator for *Start*. In this case, the generator will be applied a number of times to create a start time for each event. The distribution of start times will then reflect the characteristics of the generator.

Wsine4:

```
Synthdef:  sine1
Layers:    1
Include:
Number:    100
Start:     (exponential-value 0.0 0.1 20)
Duration:  dur        (random-value 0.3 1)
           amp        0.05
           freq       (exponential-value 440.0 0.01 1000)
           attack     0.01
Per Layer:
Per Score:
```

Exponential-value tends to create values near its threshold, in this case 0. If values occur above a certain value, they will be discarded. In this expression, the limit is set at 20 seconds. Most start times will be near the beginning of the score and no event will begin after 20 seconds. When specifying a generator for start times, it is important to remember the difference between producing integer values and real values. If integer values are produced, events would only start on integer time units, e.g. seconds 0,1,2, or 3. Real values could start at any point in the available space.

Frequency in this OSC score object also uses an exponential distribution. Most values will be near 440 Hz and never above 1000 Hz. *attack* has been shortened to 1% of the total duration.

If *Start* is a generator and masks, shapes, or anything in which the order of the results is important are used in other arguments, *make&sort* should be used to calculate the start values to ensure that the values for the other arguments are in the correct order and do not follow the possibly random start times of the *Start* generator.

```
(make&sort 100 (exponential-value 0.0 0.1 20))
```

The shortcut for *make&sort* is *ms*.

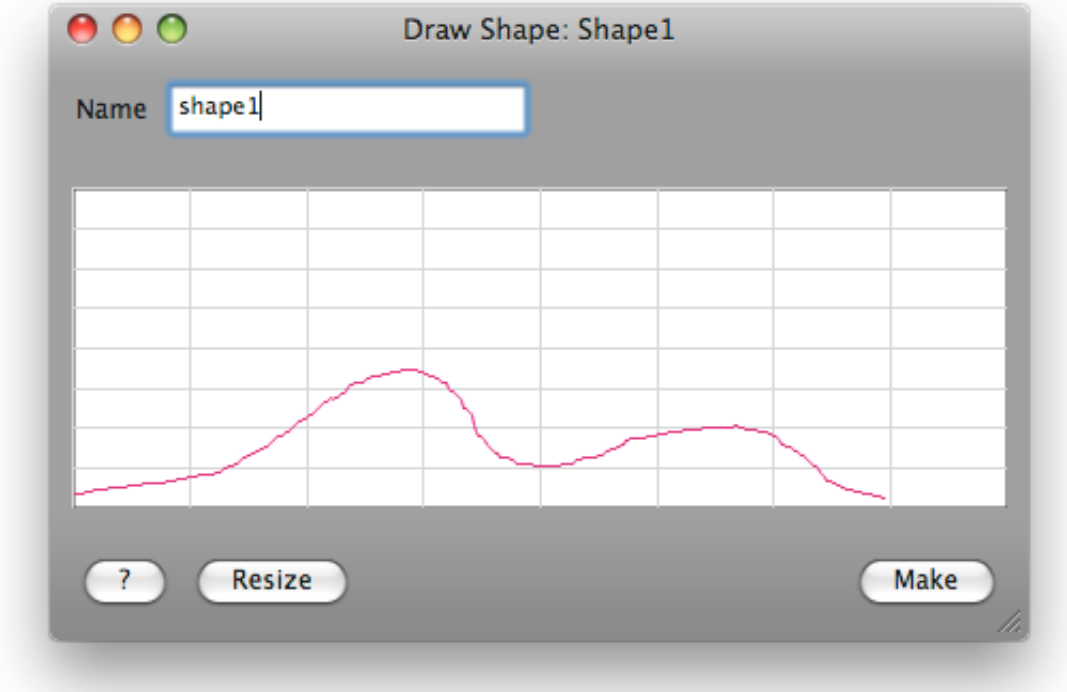
Calculating densities

A tool for calculating start times is *density-of-start-times*. Arguments are the amount of time in seconds to be filled, a shape controlling density, and the minimum and maximum density values (per second).

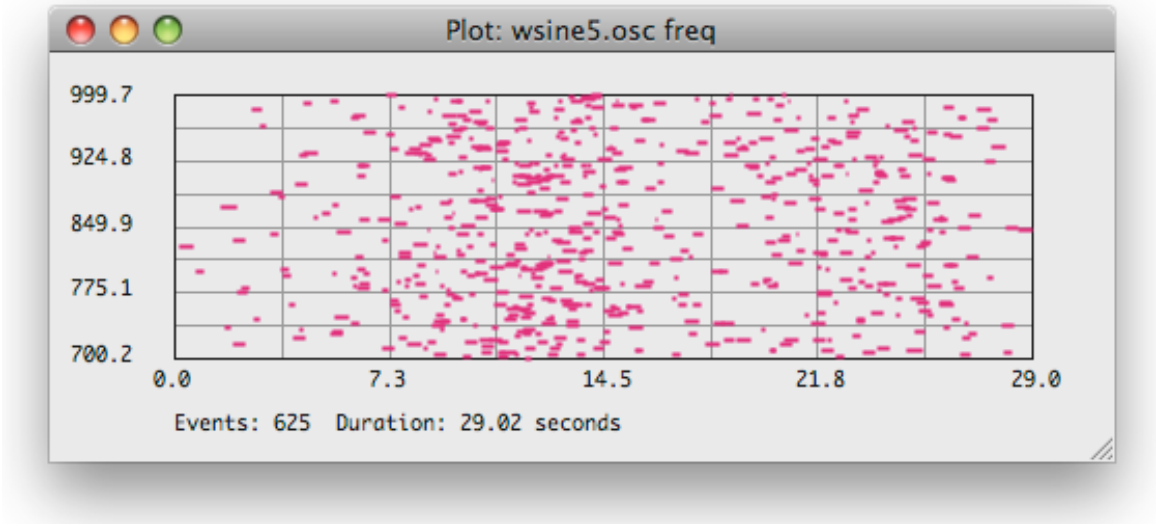
Wsine5:

```
Synthdef:  sine1
Layers:    1
Include:
Number:    (from-start-times)
Start:     (density-of-start-times 30 shape1 0.5 50)
Duration:  dur        (random-value 0.1 0.5)
           amp        0.05
           freq       (rv 700.0 1000)
           attack     0.01
Per Layer:
Per Score:
```

Density will vary according to the form of *shape1*.



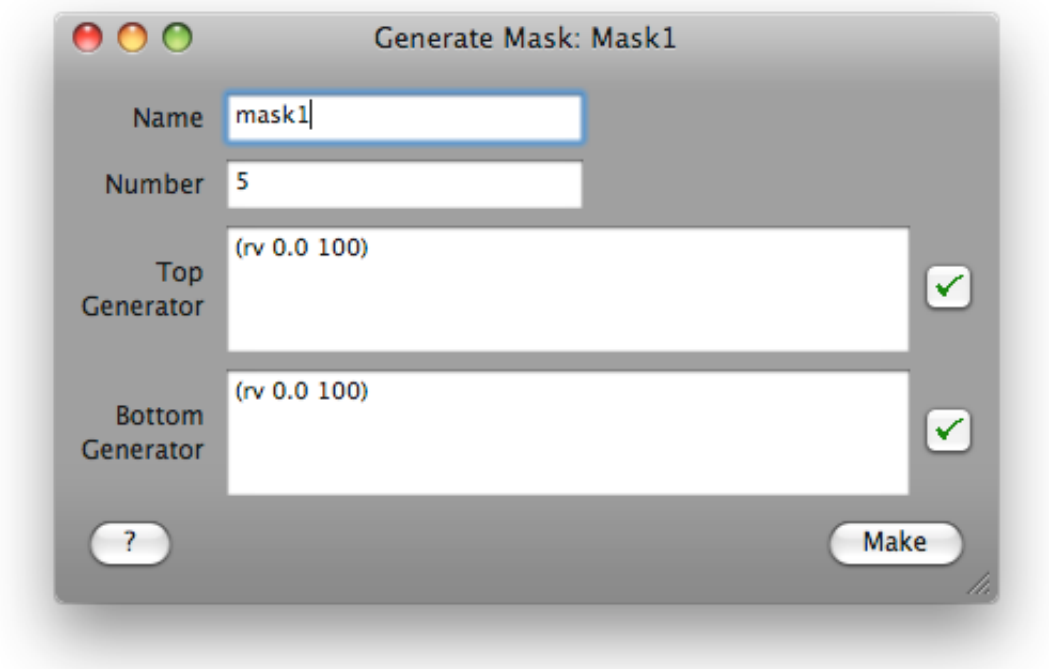
The lower density is 0.5 which amounts to one event per two seconds. The upper limit is 50 events per second. *Density-of-start-times* will calculate as many start times as it needs to realize the specified densities. The number of values needed to do this is only available after the calculation has been made. The number of events must be derived from the generated start times. This is done with *(from-start-times).*



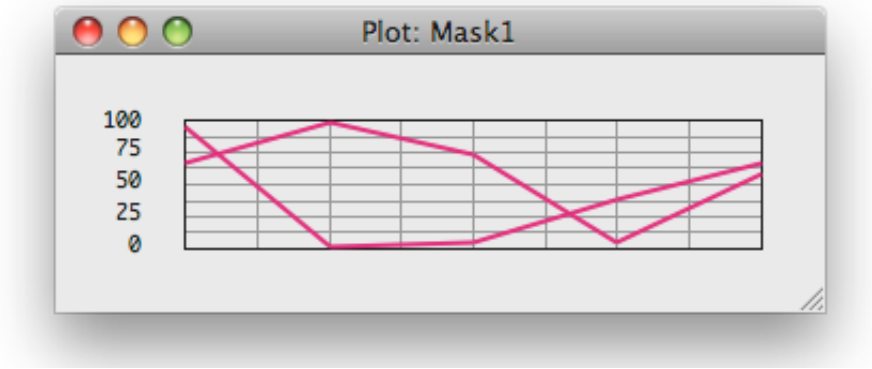
Since all of these events are happening in the same layer, it is necessary for the user to pick a very small amplitude value to prevent distortion. Here, 0.05 is used.

The shortcut for *density-of-start-times* is *density*. The shortcut for *from-start-times* is *fst*.

Remaking objects per score
A mask can be made using a generator for the upper limit and another one for the lower limit. *Mask1* will generate such a mask consisting of 5 points describing each limit.



Each time the mask is made, a new mask will result. One possible result could look like this:



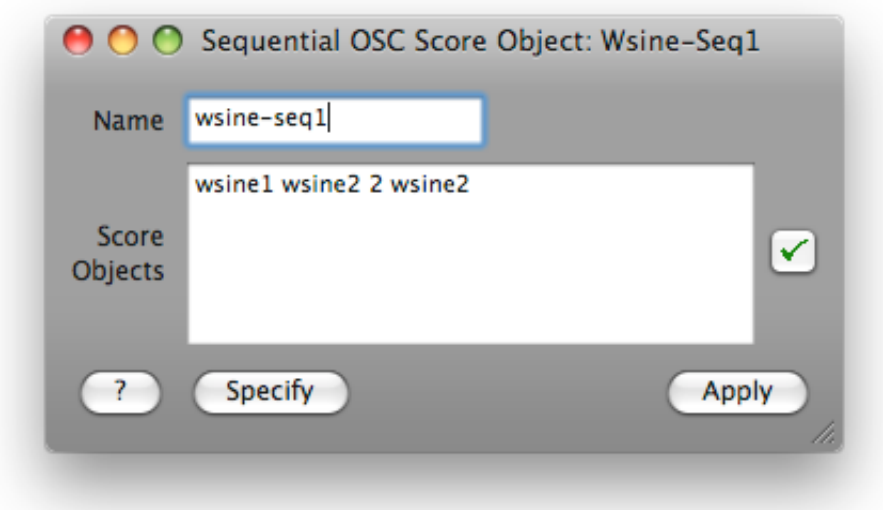
If the name of the mask is entered in the *Per Score* item, it will be generated each time the score is made.

Wsine6:

```
     Synthdef:  sine1
     Layers:    1
     Include:
     Number:    100
     Start:     0
     Duration:  dur        (rv 0.05 0.1)
                amp        0.1
                freq       (convert mask1 (from-number) 100.0 1000)
     Per Layer:
     Per Score: mask1
```

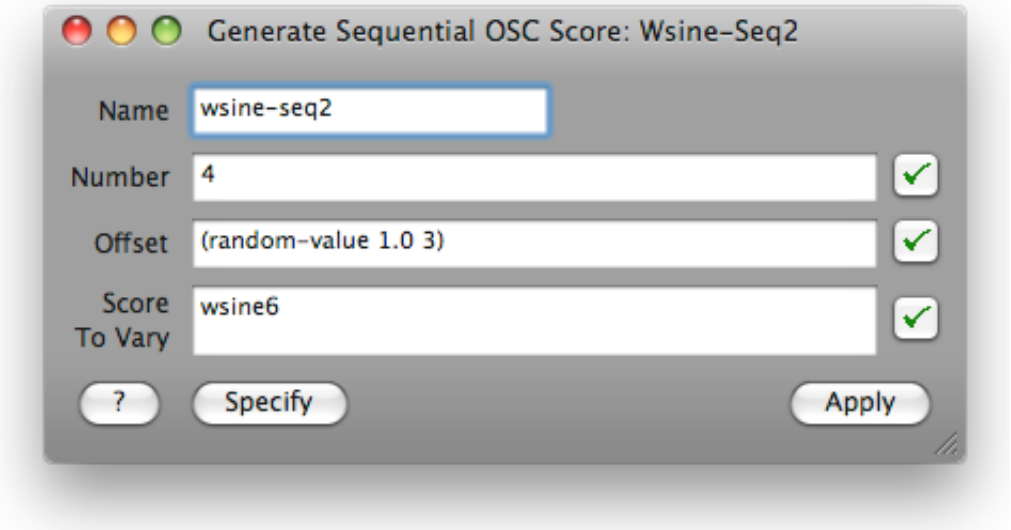
Sequential Scores

Scores can be combined in sequence with an *OSC Sequential Score* object. The names of the objects to be combined should be entered. They can be typed or dragged from the Objects dialog. It is possible to specify a time in seconds between the objects. Each time a score object is combined, it is made again. This means that the result could be different each time, even within the same sequential score.



In *wsine1-seq*, an output made with the algorithm of *wsine1* is followed by an output made with *wsine2,* followed by two seconds of silence, and then a new output made with *wsine2*. The two applications of *wsine2* will produce different results.

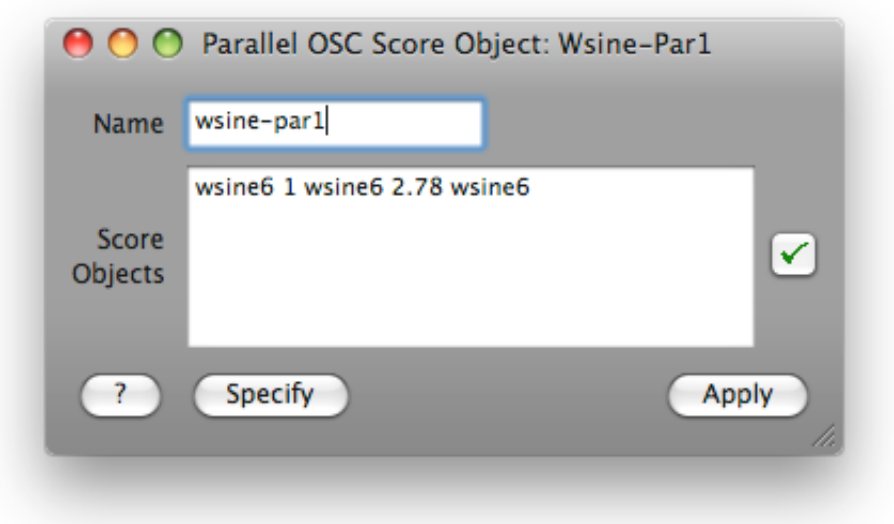
A sequential score can be generated using the menu item **Generate Sequential Scores**. The order of scores to use in a sequential score object can be determined with a generator such as *(random-choice '(wsine1 wsine2)).* The name of just one score can also be used, in which case a number of variants of that score will be joined in sequence. *Offset* is the time in seconds between scores in the sequence. It can be a constant value, generator, etc. If it is 0, there is no gap between scores.



In *wsine-seq2*, four variants of *wsine6* are calculated, with a random gap of 1 to 3 seconds between them. A new mask is generated for each variant since *mask1* is generated per score in the specification of *wsine6*.

Parallel Scores

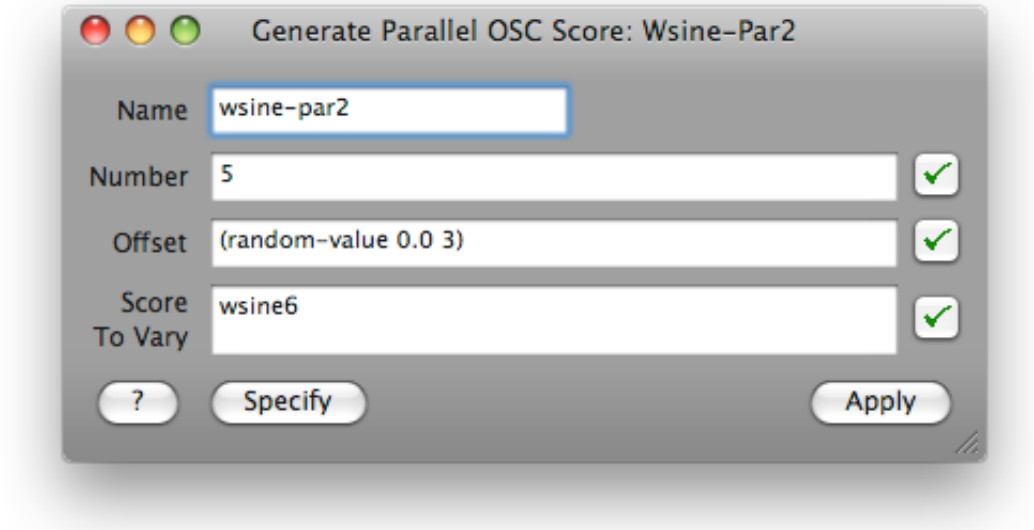
Scores can be combined in parallel with an *OSC Parallel Score* object. The names of the objects to be combined should be entered. They will all start at the same time. It is possible to specify a time in seconds between the objects. This will cause the second object to start later. The start time is relative to the beginning of the score.



In *wsine-par1*, the score will start with an instance of *wsine6*. One second after the start of the score, another instance of *wsine6* will start. 2.78 seconds after the start of the score, a third instance of *wsine6* starts. Each instance of *wsine6* generates a new mask and creates different frequency values based on the mask.

When scores are joined in parallel, attention must be paid to the amplitudes of each score.

A parallel score can also be generated using the menu item **Generate Parallel Scores**. *Number* is the number of scores to combine in parallel. *Offset* is the start time from the beginning of the score for each layer. The first score starts at time 0. Each subsequent score uses a calculated offset time. *Score To Vary* is either the name of a score or a generator that can choose the names of scores.



Objects and layers
If a score uses an object such as a mask or stockpile that is generated once per score, all layers of that score will share the same object. The object should be specified in the box for *Per Score*. Several objects can be specified in this box.

Wsine7 generates a mask once per score and uses the same mask for all 5 layers.

```
Synthdef:  sine1
Layers:    5
Include:
Number:    100
Start:     0
Duration:  dur        (rv 0.1 0.2)
           amp        0.1
           freq       (convert mask1 (from-number) 100.0 1000)
Per Layer:
Per Score: mask1
```

If each layer should have a new mask, it should be specified in *Per Layer*. This box can contain the name of several objects.

Wsine8 generates a new mask for each layer:

```
Synthdef:  sine1
Layers:    5
Include:
Number:    100
Start:     0
Duration:  dur        (rv 0.1 0.2)
           amp        0.1
           freq       (convert mask1 (from-number) 100.0 1000)
Per Layer: mask1
Per Score:
```

Relating parameters
A convenient way to relate two parameters is to calculate one of the parameters as a stockpile. In the score definition, the stockpile can be used for one parameter and an expression using that stockpile can be entered for the other parameter.

In any case, the parameter that is to be examined must exist before the score is made. A stockpile is a convenient way to do this.

Freqs is a stockpile of more or less ascending frequencies calculated using *spray* that produces values with a random deviation around a shape.

In *Wsine9*, *freqs* is used for the *freq* argument. The durations in this OSC score are the inverse of the frequencies in *freqs* multiplied with 100. This is done with the tool *cal*. The durations become shorter as the frequencies become higher.

```
Synthdef:  sine1
Layers:    1
Include:
Number:    100
Start:     0
Duration:  dur        (cal 1.0 / freqs * 100)
           amp        0.1
           freq       freqs
Per Layer:
Per Score: freqs
```

A lookup table is another way to relate parameters. The generator *lookup* can read from a table made with the tool *make-lookup-table* that can relate keys to values, generators, etc. *Act-if* can check various conditions to determine a value for another parameter.

Updating parameters
While an event is sounding, additional OSC commands can be sent to that event to change parameter values. For example, in an event of 10 seconds, frequency values could change every second.

To do this, the tool *inside-osc* should be used. It will update parameters inside a sounding OSC event.

Inside-osc has two parameters, *rhythm* and *value*. *Rhythm* is a time in seconds to wait before making the next update. It can be a constant, generator, etc. *Value* should return the next value for the update. It can be a generator, stockpile, etc. When an event is started, the first update value is sent. After *rhythm* seconds, the next update is sent. This continues until the event is finished.

Wsine10 produces 10 events. During each event, the frequency value is updated every .01 seconds. Frequency values are generated with *generate-line* that interpolates from a start to end value in N steps, then picks a new value for N and interpolates to a new value for END in that many steps. The duration parameter alternates events and rests (negative numbers).

```
Synthdef:  sine1
Layers:    1
Include:
Number:    10
Start:     0
Duration:  dur       (take 1 (rv 1.0 10) 1 (rv -1 -1.5))
           amp       0.1
           freq      (inside-osc 0.01
                       (generate-line 400
                           (rv 100 1000)
                           (rv 50 3000)
                           :exponential
                           t))
Per Layer:
Per Score:
```

Using a buffer

Filteredbuf is one of the synthdefs included in the file *test osc synthdefs.rtf*. It reads a buffer starting at a frame position and sends the result through a bandpass filter. If not already compiled to the folder *synthdefs*, this should be done before trying the examples that use *filteredbuf*.

```
(
SynthDef("filteredbuf", {arg out = 0,  bufnum = 0, amp = 0.5, dur = 1,
attack = 0.1, decay = 0.1, freq = 800, rq = 0.5, position = 0;
 var soundout, ss = dur - attack - decay;
 soundout = PlayBuf.ar(1,bufnum, 1, startPos: position);
 soundout = BPF.ar(soundout,freq,rq) * EnvGen.kr(Env.linen(attack, ss,
decay,amp), doneAction: 2);
 Out.ar(out, soundout);
}).writeDefFile;
)
```

freq is the center frequency of the bandpass filter. *rq* is the reciprocal of *q*. The smaller the value of *rq*, the more narrow the bandwidth. *attack* and *decay* are expressed in seconds. *position* is the position to start reading the buffer. Position is expressed in frames. In a mono buffer, each value is a frame. In a buffer with more channels, a frame contains a value for each of the channels.

Filling a buffer
To use a buffer in an OSC score object, an OSC message allocating a buffer and reading a file into that buffer is needed. OSC messages can be expressed in the AC Toolbox with the *message* tool. A message should be enclosed in a *bundle* that contains a time for sending a message and the message itself. Each bundle may contain only one message.

Bundles, such as those needed for allocating buffers, should be entered in the box for *Include*. If no bundles are needed, *Include* can be left blank. Otherwise one or more bundles can be entered.

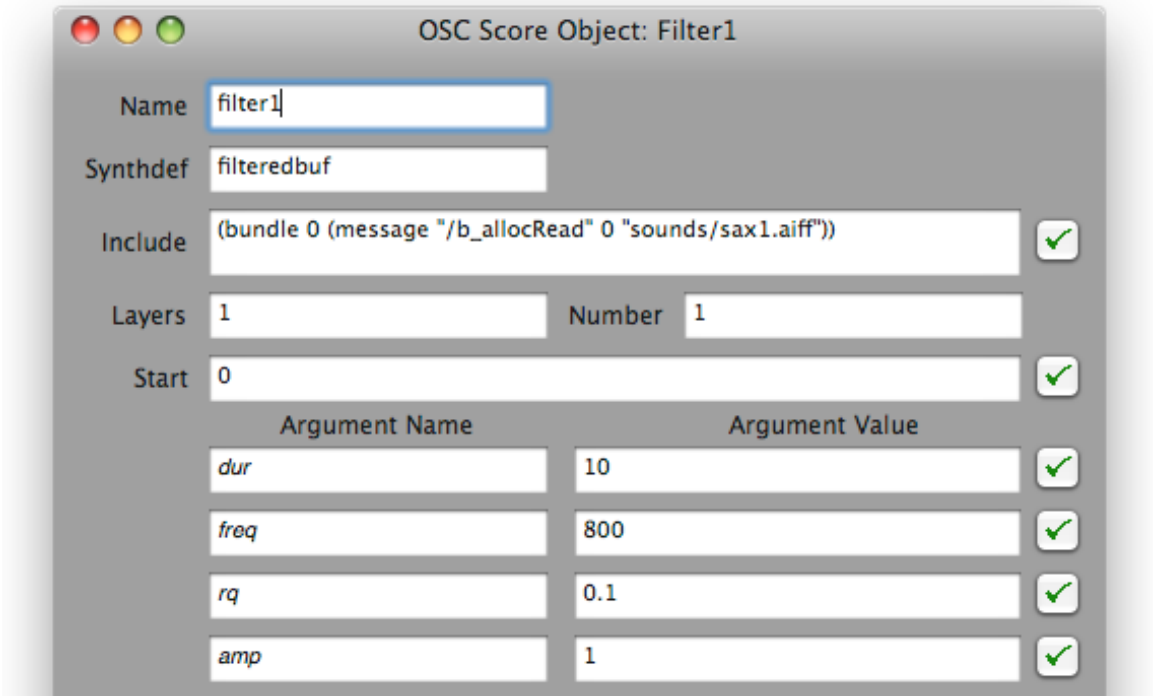
OSC messages that are understood by the SuperCollider server are described in the SC documentation in the *Server Command Reference*. A server command should be entered as a string.

A bundle to allocate and read a file into a buffer is:

```
(bundle 0 (message "/b_allocRead" 0 "sounds/sax1.aiff"))
```

This bundle sends the message at time 0. The message contains a server command "/b_allocRead", a buffer number 0 and a file name. In this case the file is found in the sounds folder within the SuperCollider folder.

Filter1 fills buffer 0 with a file. The user should replace the file name with an available mono audio file with a length of at least 10 seconds. An 11-second mono file called *Roscoe.aiff* is available in the *Tutorial 20 Spectra* folder.



In the score, 10 seconds of the buffer is played. The center frequency for the filter is 800 Hz.

Calculating frame position

A time value in seconds can be converted to a frame position using the generator *osc-frame*.

(osc-frame 1) returns the frame position for a point that is 1 second into the buffer. The argument for *osc-frame* can be a constant, a generator, etc. This generator is used in *filter2* to allow reading the buffer from random positions that are generated in seconds and converted to frames.

```
Synthdef:   filteredbuf
Layers:     1
Include:    (bundle 0 (message "/b_allocRead" 0 "sounds/sax1.aiff"))
Number:     20
Start:      (gaussian-value 5 10 0 20)
Duration:   dur         (random-value 3.0 7)
            freq        (random-value 400.0 2000)
            rq          0.01
            amp         1
            position    (osc-frame (random-value 0.0 4))
            decay       (random-value 1.0 3)
```

```
Per Layer:
Per Score:
```

Filter2 also uses a generator for *Start*. Twenty events will start at times calculated with a Gaussian distribution with a mean value of 10 seconds and an upper limit of 20 seconds. *rq* has been decreased from the value in *filter1*.

Bunch of Bundles
If a bundle is used frequently in different scores, it may be convenient to save it in an OSC Score Bunch object. The object can be selected with the menu item **Bunch of Bundles**. This object can contain one or more bundles. The name of the object can be entered in an *Include* box but also used in sequential and parallel scores.

To have a bunch available for use in another object, it should be *specified*. Applying the bunch would cause a binary OSC file to be written. This might be an unnecessary step.

If several different buffers are used in a score, there should be a bundle for each buffer allocation. These bundles can be gathered in an OSC Score Bunch.

It is also possible for an OSC score bunch object to contain several bundles that would start some synthdef, send it data messages, and then turn it off though this is probably easier to do in the SuperCollider language application.

Filter3 uses the name of *bundle1* instead of the bundle description in the *Include* box. *Include* should either use a OSC Score Bunch object or one or more bundles. The two methods of input should not be mixed.

```
Synthdef:   filteredbuf
Layers:     1
Include:    bunch1
Number:     (random-value 8 30)
Start:      (gaussian-value 5 5 0 (make (rv 10.0 20)))
Duration:   dur         (random-value 3.0 7)
            freq        (random-value 400 2000)
            rq          0.1
            amp         1
            position    (osc-frame (random-value 0.0 4))
            decay       (random-value 1.0 3)
Per Layer:
Per Score:
```

Filter3 uses a generator to determine the number of events in the score. It also uses a generator to determine the maximum value for the start times. When used in a sequential score, different applications of *filter3* would produce different results.

Filtered-seq1 produces 4 variants of *filter3*.

Using b_gen commands
Buffers in SuperCollider can be filled with *b_gen* commands. A buffer should be allocated and then filled with a wave fill command such as *sine1* or *sine2*. The first argument in those commands concerns flags. If they should all be on, the value 7 is used. The rest of the data for the wave fill commands should follow.

Object *bunch2* contains a bundle to allocate a buffer and one to fill it with wave fill command *sine1*. This object should be *specified* (not applied).

```
(bundle 0 (message "/b_alloc" 0 512 1))
(bundle 0.1 (message "/b_gen" 0 "sine1" 7 1 0 0 0.5))
```

Object *b_gen-example* uses *bunch2* to allocate and fill a buffer. This object uses synthdef *SimpleOsc*.

```
Synthdef:  SimpleOsc
Layers:    1
Include:   bunch2
Number:    50
Start:     0
Duration:  dur        (rv 1.0 2)
           amp        0.3
           freq       (walk 400.0 (rv –50.0 50))
Per Layer:
Per Score:
```

Using an audio input file

In the *OSC File Options* dialog, there is check box **Render with an Audio Input File**. If checked, an audio file can be selected. This audio file provides audio input for the *SoundIn* UGen if it is used in a synthdef. A new file can be selected with the *Audio Input* button in the options dialog.

Synthdef *soundin* from file *test osc synthdefs.rtf* should be written to the synthdefs folder of SuperCollider.

```
(
SynthDef("soundin",{ arg out=0, dur = 1, attack = 0.1 , decay = 0.1,
amp = 0.5 ;
 var ss = dur - attack - decay;
 Out.ar(out,
        SoundIn.ar(0) * EnvGen.kr(Env.linen(attack,ss, decay,amp),
doneAction: 2)
 )
}).writeDefFile;
)
```

If **Render with an Audio Input File** is checked, the audio input for this synthdef is provided by that file.

soundin just reads an input and multiplies it by an amplitude envelope. This example is intended to show how an input file can be specified.

soundin-example reads an input file for 10 seconds. If no audio input file has been specified in the options, there will be 10 seconds of silence.

```
Synthdef:   soundin
Layers:     1
Include:
Number:     1
Start:      0
Duration:   dur         10
            attack      0.1
            decay       0.5
Per Layer:
Per Score:
```

The audio input file should not be confused with the use of a buffer. It is not necessary to check **Render with an Audio Input File** in order to allocate and read files to a buffer.

Miscellaneous

Once a synthdef has been compiled and written to the synthdefs folder, it is not necessary to have the SuperCollider language application open. Subsequent sessions with the AC Toolbox will use the previously compiled synthdefs. This is an example of the *compile once, use often* principle.

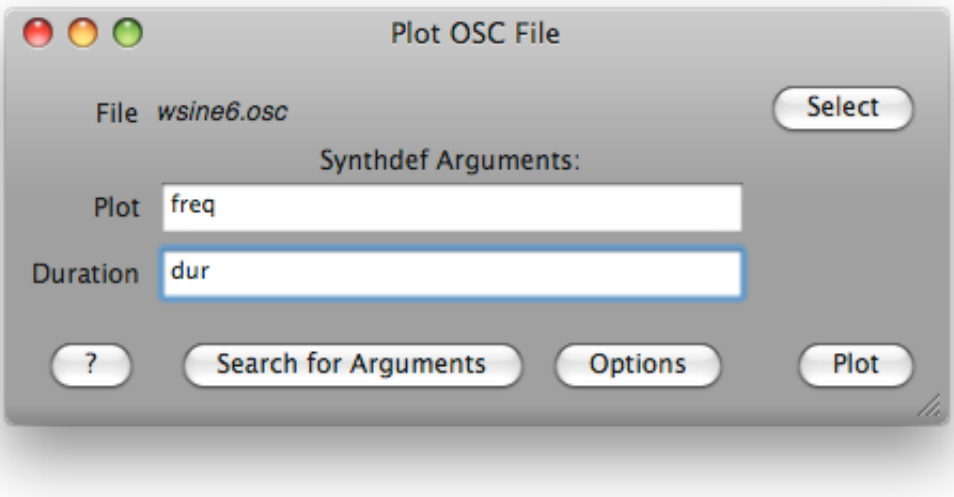
The synthdefs should be in the default folder (as specified by SuperCollider). Compiling the synthdefs in the language application should write them to the proper location.

Additional examples of using OSC scores can be found in the AC Toolbox *Index*. Look for *OSC: Score Objects*.

If an audio input file is used, SuperCollider might give a warning concerning the number of input channels. This warning can be ignored.

If the audio file reports errors on opening, examine the Text Output window for possible error messages. If the Toolbox did not print the results from SuperCollider, select the option in the *OSC File Options* dialog to **Print results from SuperCollider** and apply the object again. If too many events are specified for the binary OSC file, it may be that it is not rendered properly within the AC Toolbox. In that case, the file could be rendered within the SuperCollider language application (see the SuperCollider help for non-realtime synthesis).

An argument from an OSC file can be displayed via **Tools>Plot>OSC File.** A histogram can be made via **Tools>Histogram>OSC File**. Note that these representations are of the file data created by the OSC score object and not a representation of the object itself.



An OSC file can be filtered (**Methods>Filter>OSC File**) or transformed (**Methods>Transform>OSC File)**. These methods are described in Index items in the application (OSC Files: Filtering and OSC Files: Transforming).

A separate tool, *at-osc-time*, can be used to transform events in an OSC file between specified start and end times. The tool can be evaluated in the Listener (**Other>Listener**) or in a Lisp editor window.

An existing binary OSC file can be rendered to an audio file (if the corresponding audio file is no longer available or different options are desired) by evaluating (render-osc) in the Listener or in an editor window. There is also a button in the OSC File Options to do this. The file will be rendered using the options currently specified in the OSC File Options.

Summary

Menu items
OSC scores, OSC file options, generate sequential scores, generate parallel scores, bunch of bundles, plot>OSC file, histogram>OSC file, filter>OSC file, transform>OSC file

Generators
osc-frame

Tools
bundle, message

Miscellaneous
specify, apply

Tutorial 22

Notating music: FOMUS

Translating computer data to a form that can produce useful results in a music notation program such as Finale, Sibelius, or LilyPond can be tricky. In particular, the notation of rhythmic values can be problematic. Opening a Midi file in a notation program may give surprising results, particularly in regard to rhythm.

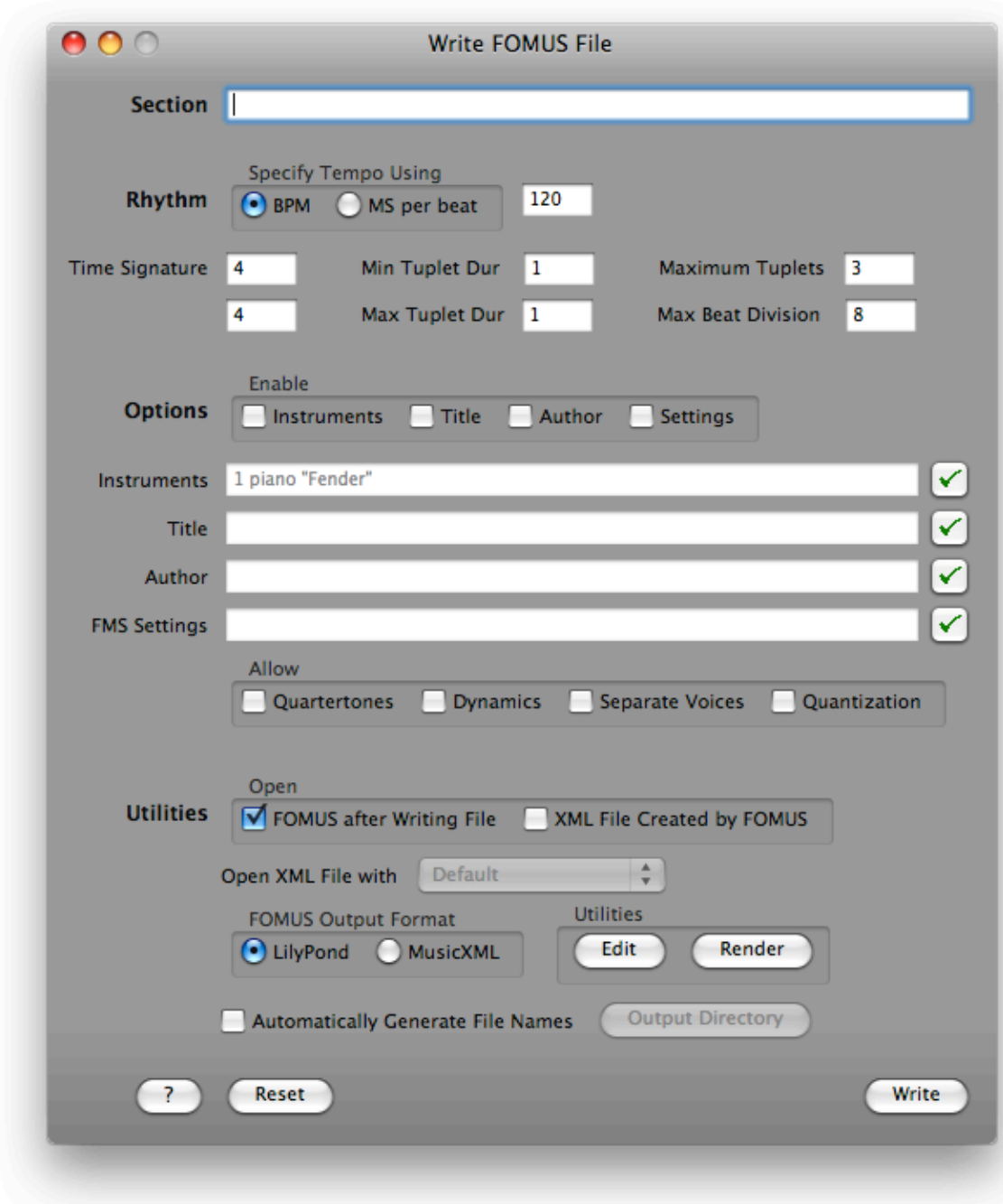
FOMUS
Nonetheless, it may be useful to have another way to examine data generated in the AC Toolbox. One way to link data to a variety of notation programs is FOMUS that is a music notation formatting tool. It tries to rationalize data to produce sensible rhythmic representations and pitch spellings. It also facilitates the use of quartertones. FOMUS can produce output in LilyPond format but also in MusicXML format. The latter can be read by programs such as Finale and Sibelius.

This may not be a representation that can be used verbatim as a score but it may provide insights and can be considered a sketch of certain features of the material.

FOMUS needs to be installed. It can be downloaded from http://fomus.sourceforge.net/
The AC Toolbox expects FOMUS to be installed in */usr/local/bin* which is where the FOMUS installer puts it. The AC Toolbox assumes that FOMUS version 0.1.10 or higher is installed.

A notation program should also be available. If LilyPond is chosen as the preferred output mode, it should be installed in the Applications folder (http://lilypond.org/). The free program Finale Reader can read MusicXML files as can the full versions of Finale and Sibelius. Programs may support different features of MusicXML.

The AC Toolbox can process section data to a file that can be input to FOMUS. This is done with the dialog opened by **File>Write FOMUS File**. In this dialog various options can be specified, including a choice of output type (LilyPond or MusicXML). The file produced by the Toolbox will have the extension *fms*. The Toolbox will call FOMUS and have it produce an output file with data in LilyPond format (extension *ly*) or in MusicXML (extension *xml*). If LilyPond is chosen as the output format, LilyPond will produce a pdf file with the notated results. If MusicXML is chosen, a program should be specified to read this file.



The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 22 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

These objects will be used to discuss some of the possibilities of the dialog for writing FOMUS files.

Beats, time signatures
Section *equal* has a number of ascending pitches with equal rhythms.

```
Name            equal
Clock unit      (bpm 240)
Number          20
Rhythm          1
Pitch           (walk c2 (rv 1 4))
Velocity        f
Channel         1
```

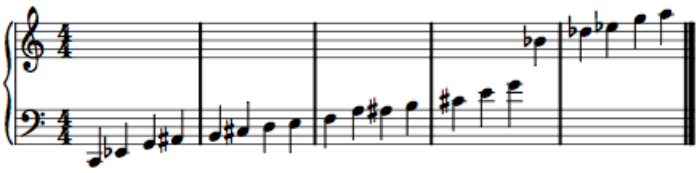
In the *Write FOMUS File* dialog (hereafter referred to as the FOMUS dialog), click on the *Reset* button to restore all values to their default settings. In the portion labeled *Rhythm*, tempo can be specified as BPM or as a value in milliseconds per beat. For the first example, set the BPM value to 240. This is the value used in the section specification. The time signature will be left at 4/4. In the portion of the dialog labeled *Utilities*, *Open FOMUS after Writing FILE* should be checked. The examples presented in this text will be made with LilyPond so the radio button for LilyPond is selected in the *FOMUS Output Format.*

If MusicXML output is to be made, select *MusicXML* as the output format and *Open XML File Created by FOMUS.* The popup menu for *Open XML File with* can be used to select an application to use to open the MusicXML file.

Enter the name for the section (*equal)* and click on the *Write* button. A FOMUS file will be written, FOMUS will process this data and produce a LilyPond or MusicXML file. Next, LilyPond will process the data and produce a pdf file. If MusicXML output was chosen, FOMUS writes an *xml* file that will be opened by the application indicated in the popup menu.

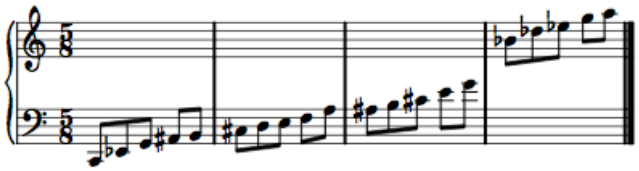
The status of the data processing can be seen in the Text Output window.

This process produces this notation for possible events in section *equal*:



• Try changing the value for BPM to 120. The result will be eighth notes.
• Change the *Time Signature* to 4/8 (Leave the top value at 4 and change the bottom value that is in a separate edit box to 8). Also return the BPM value to 240.

Changing the time signature to 5/8 (with a BPM of 240) produces the following result:



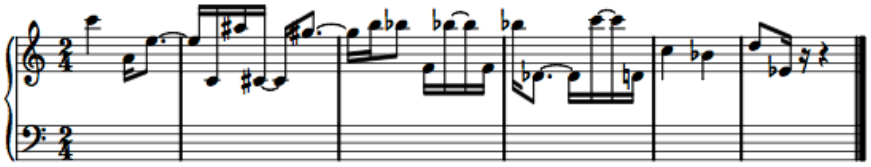
In the above examples, tempo was expressed in BPM. A beat is the denominator of the time signature.

Section *dotted* has a rhythmic pattern of *2 1 2 1…*
• Notate this section with 4/4.
• Notate this section with 3/4.

Instruments, dynamics
Section *melody1* uses the following rhythmic specification:

```
(rc '(1 1/2 1/4))
```

With a time signature of 2/4, quarter notes, half notes, and eighth notes will be produced. BPM was set to 120 for the following:



Each Midi channel can be mapped to a separate instrument. FOMUS has a number of predefined instruments such as piano, flute, cello, and tenor-trombone. More names and information can be found in the FOMUS documentation and code.

In the FOMUS source code (that can be downloaded separately), the file *src/data/fomus.conf* contains several instrument definitions. In addition, instrument definitions can be discovered in the text printed by typing the following command line in the Terminal application:

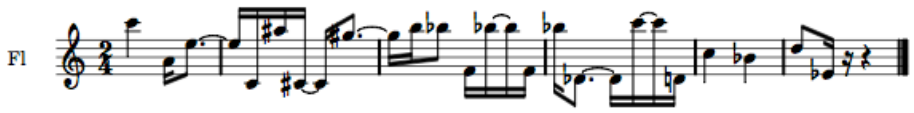
```
fomus –S –fuselevel=3
```

The text printed with the above command line documents various FOMUS options that can also be entered. Two of these will be discussed later.

To specify an instrument, check *Enable Instruments* in the *Options* portion of the dialog. This enables the edit box for Instruments. Enter the following description:

```
1 flute "Fl"
```

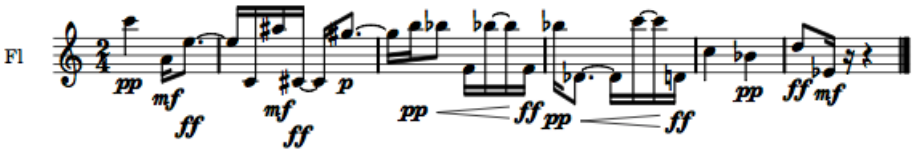
This maps Midi channel 1 to use the FOMUS definition for flute. The result will only have one staff that will be labeled *Fl*.



Melody1 contains dynamics produced by:

```
(rc '(pp p mf f))
```

To include the dynamics in the output, check *Allow Dynamics* in the *Options* portion of the FOMUS dialog.



Quartertones

Microtones are represented in the AC Toolbox as floating-point Midi numbers (see tutorial 18). Quartertones are values quantized to intervals of .5. 60 is middle C, 60.5 is a quartertone higher.

Floating-point values are produced in the Toolbox when the input to a generator or tool contains a floating point number. The following expression,

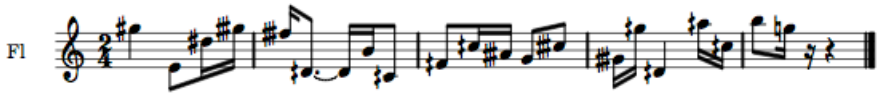
```
(random-value 60.0 72)
```

produces floating-point values in the range 60-72.

If the symbolic notation *c4* is used for note number 60, a floating-point value can be requested by enclosing c4 in function *float*. This is what happens in the pitch specification for *melody2*.

```
Name            melody2
Clock unit      (bpm 120)
Number          20
Rhythm          (rc '(1 1/2 1/4))
Pitch           (rv (float c4) c6)
Velocity        (rc '(pp p mf f))
Channel         1
```

• Make *melody2*. In the FOMUS dialog, check *Allow Quartertones* and uncheck *Dynamics*. The result could look like this:



FOMUS rounds the floating point input to quartertones.

LilyPond and Finale can deal with quartertone input from FOMUS. Another programs may not bother to read the MusicXML indications for quartertones.

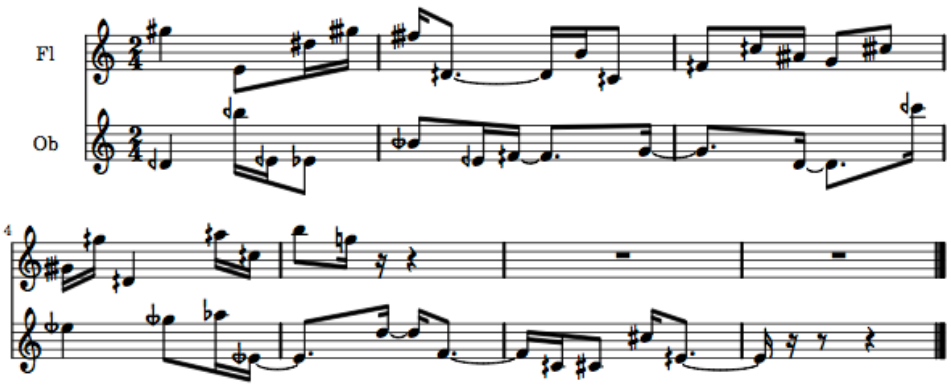
Using two instruments

• Make *melody3*. It uses Midi channel 2. Add another instrument to the instrument specification in the FOMUS dialog. The entire instrument specification is now:

```
1 flute "Fl" 2 oboe "Ob"
```

Midi channel 1 will still be assigned to flute and Midi channel 2 will be assigned to oboe.

• Write *melody3* to a FOMUS file. The resulting notation will only be of channel 2.
• Make *two-melodies* that is a parallel section containing *melody2* and *melody3*. The result will be similar to:



Additional rhythm options

Melody4 has a more problematic rhythm specified by:

```
(rv 0.5 2)
```

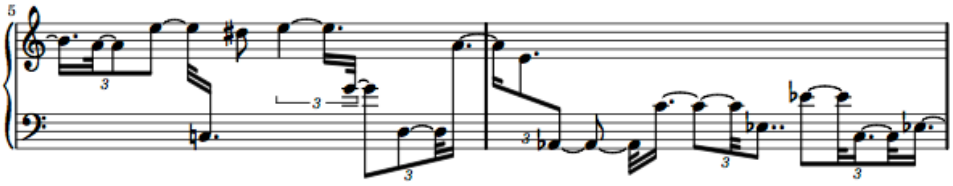
The result is floating-point values that do not neatly line up in conventional divisions of a rhythmic grid. Possible outputs for such a specification could be:

```
181/125 397/250 211/125 226/125 133/250 349/250 219/125 …
```

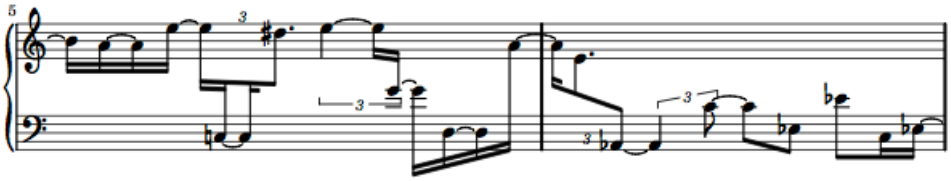
Option *Max Beat Division* is the maximum number of divisions of the beat allowed. If the time signature is 4/4, a *Max Beat Division* of 4 allows fewer possible rhythmic values than the default value of 8.

• Uncheck *Enable Instruments*.

A portion of a possible output with time signature 4/4, bpm of 120, and max beat division of 8:

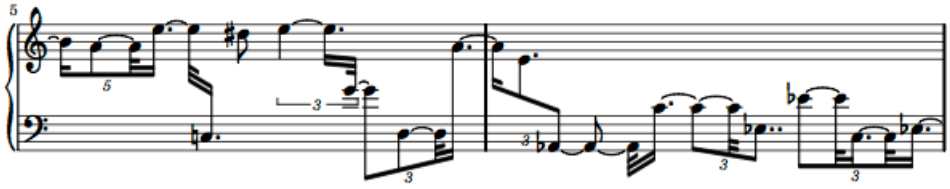


Using a max beat division of 4 for the same measures:



Maximum Tuplets is the maximum number of notes that can be grouped as a tuplet.

• Set *Maximum tuplets* to 5 and *Max Beat Division* to 8. Notate *melody4.* The previous two measures could look like:



FOMUS uses a method of quantization to rationalize rhythms. The method can be influenced by the various parameter settings as described above.

Quantize is an option to quantize in a different manner before FOMUS does its work. It uses the Pre-FOMUS Rhythmic Quantizer written by Luc Döbereiner. The original section is not modified. A quantized copy of the section is formatted by FOMUS.

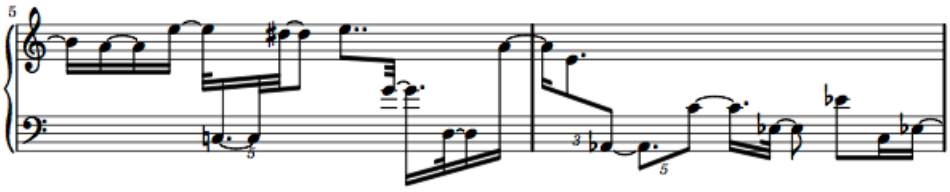
The Pre-FOMUS Rhythm Quantizer aligns the events of a given structure on fractions of beats before it is written out and processed by FOMUS. For complex rhythmical sections, this will likely result in a more easily readable score. The possible fractions are determined by the setting *maximum tuplets*. The quantizer searches for the best fitting division for each beat. A beat can be divided into 2, 3, 4, 5, 6, 7 and 8 onsets. Faster passages will be grouped in chords. In case of a structure containing several channels, the events of each channel will be quantized independently.

When the quantizer is used, the setting for *maximum tuplets* should be 2, 3, 5, or 7. Any other value will cause an error message.

The Quantization option is separate from the quantization that FOMUS might do. The Quantization option indicates if the Pre-FOMUS Rhythmic Quantizer should be used or not. It has no influence on the quantization that may be performed by FOMUS.

The Pre-FOMUS Rhythm Quantizer quantizes channels separately. It does not produce accurate results if events were created with the the generator *make-chord.*

With the same settings as the previous example (*Maximum tuplets* 5 and *Max Beat Division* 8) and with the *Allow Quantization* option checked, results similar to this could be produced:



Including other FOMUS settings

For any given section and any given situation, a lot of experimentation can be done to discover usable results. Many other options are available in FOMUS that have not been included in the interface of the FOMUS dialog. All FOMUS settings can be seen by typing

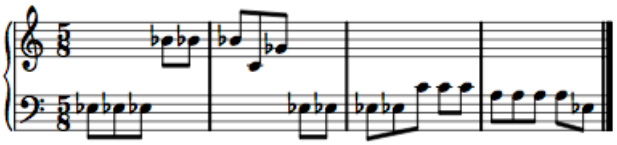
```
fomus –S –fuselevel=3
```

in the Terminal application.

To include one or more of these settings, *Enable Settings* should be checked and the appropriate settings entered in the edit box labeled *FMS Settings*. To see the settings that the AC Toolbox uses for FOMUS, click on the Edit button in the *Utilities* portion of the FOMUS dialog. If you have written a FOMUS file, this will open the most recent one.

```
Name            melody5
Clock unit      200
Number          20
Rhythm          1
Pitch           (group (rc '(c4 d#3 f#4 a3 bf4)) (rv 1 3))
Velocity        (rc '(pp p mf f))
Channel         1
```

• Make *melody5*. Set the time signature to 5/8. and then notate it. The results could look like:



FOMUS has a setting for accidentals. If you want an accidental to last for an entire measure instead of being specified for each note, you could enter:

```
acc-rule = measure
```

as an FMS setting.

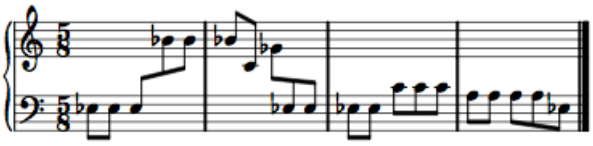
If you want the default measure division to be 2 and 3 instead of 3 and 2 as notated above, this setting can be used:

```
default-meas-divs = ((2 3))
```

To notate *melody5* with both settings, they both can be included in the edit box for settings:

```
acc-rule = measure default-meas-divs = ((2 3))
```

This could produce the following result:



The remaining portions of the FOMUS dialog that have not been discussed in this tutorial are mentioned in the help in the application. Either click on the *?* button in the FOMUS dialog or look at the item *Writing FOMUS Files* in the Index (**Help>Index**). For all other questions, the FOMUS documentation should be consulted.

Summary

Menu items
Write FOMUS File

Tutorial 23

More opportunities to filter

A few possibilities to filter objects were discussed in Tutorial 7. The filter method (**Help>Filter>Objects**) allows different types of filters to be applied to various parameters. Sections, shapes, stockpiles, etc. can be filtered. Some objects, such as streams, cannot be filtered.

In this tutorial, two generators for filtering data as it is being made will be discussed. In addition, some filters for the filter method will be mentioned. In particular, filters that combine conditions will be presented. An example of such a combination is that rhythm is only filtered if pitch is above a certain value,

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 23 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Generate without …
Without is a generator that can filter values coming from another source, such as a generator or a stockpile. Many different kinds of constraints can be used for this filtering process. *Without* rejects values that satisfy the specified constraints and asks the generator to produce a new value.

The specification for *without* is:

```
(WITHOUT VALUE GENERATOR-OR-STOCKPILE &KEY STOP)
```

Value is the constraint.

If value is a single number, that number is omitted if it is returned from the generator or stockpile.

If *random-value* produces 0 in the following example, that value is rejected and a new number is produced:

```
20 values from:
(without 0 (random-value -3 3))
        -2          2          1          3         -1
        -2          1          3         -3          1
        -2         -1          1          3         -3
        -1          1         -1          1          2
```

Filtering out zeroes can be useful when generating rhythmic values. Negative values are rests; positive values are event durations. Zeroes should be avoided.

Object *without1* prevents zeroes from being used for rhythm. The values 60, 64, and 67 are rejected for the pitch parameter.

```
Name           without1
Clock unit     200
Number         50
Rhythm         (without 0 (random-value -3 3))
Pitch          (without '(60 64 67) (random-value c3 c5))
Velocity       (rv p f)
Channel        1
```

Value can be a single value or a list of values. *Value* could also be a function for determining if a number should be allowed.

Anything
The tool *anything* can be used to produced a number of different constraints. `(anything > 60)`, `(anything = 50)`, `(anything >= 3)` are some examples.

In addition, *anything* can specify a region and exclude anything inside or outside that region.

```
20 values from:
(without (anything inside 60 72) (random-value 59 73))
        59        73        73        73        73
        73        73        59        73        73
        73        59        59        73        73
        59        59        73        59        59


20 values from:
(without (anything outside 60 72) (random-value 59 73))
        62        66        64        66        67
        68        63        69        61        61
        64        61        65        67        60
        71        69        63        63        66
```

An example involving the use of *anything* concerns the generator *1/f-value*. A feature of this generator is that the upper and lower limits are probably never reached and therefore the actual range of values is unpredictable.

Object *without2* generates 1/f values for rhythm and pitch without a filter.

```
Name           without2
Clock unit     100
Number         100
Rhythm         (1/f-value 128 0.1 5)
Pitch          (1/f-value 128 20 100)
Velocity       (rv p f)
Channel        1
```

The values produced by 1/f-value can be limited using *without*. In object *without2b*, any value for rhythm < 1 is rejected. For pitch, any value outside the range 40-60 is rejected.

```
Name           without2b
Clock unit     100
Number         100
Rhythm         (without (anything < 1) (1/f-value 128 0.1 5))
Pitch          (without (anything outside 40 60)
                        (1/f-value 128 20 100))
Velocity       (rv p f)
Channel        1
```

In object *without3*, random values for rhythm and pitch are produced. *Without* is used to reject values in the middle of the range.

```
Name           without3
Clock unit     100
Number         100
Rhythm         (without (anything inside 1.5 2.5)
                        (random-value 1.0 3))
Pitch          (without (anything inside g3 g4) (random-value c3 c5))
Velocity       (rv p f)
Channel        1
```

The gap in pitch values is demonstrated with this histogram:

Duplicates
Duplicate values can be filtered. A duplicate value is any value that has occurred in the output of the generator until that moment. It does not need to be the immediately preceding value. *Walk* is a generator that can make a random walk. This can involve many duplicate values.

Object *without4* uses *walk* for the pitch parameter.

```
Name            without4
Clock unit      100
Number          100
Rhythm          (without (anything inside 1.5 2.5)
                         (random-value 1.0 3))
Pitch           (walk c4 (random-value -2 2) c3 c5)
Velocity        (rv p f)
Channel         1
```

Looking at the first notes of a possible output, several duplicate values for pitch can be seen.

```
1    time: 0      duration:     145 channel:  1 velocity:  67 pitch:  60
2    time: 145    duration:     257 channel:  1 velocity:  69 pitch:  62
3    time: 402    duration:     111 channel:  1 velocity:  73 pitch:  61
4    time: 513    duration:     250 channel:  1 velocity:  76 pitch:  60 <<<
5    time: 763    duration:     277 channel:  1 velocity:  80 pitch:  60 <<<
6    time: 1040   duration:     128 channel:  1 velocity:  66 pitch:  62 <<<
7    time: 1168   duration:     296 channel:  1 velocity:  50 pitch:  64
8    time: 1464   duration:     134 channel:  1 velocity:  79 pitch:  66
9    time: 1598   duration:     133 channel:  1 velocity:  79 pitch:  64 <<<
10   time: 1731   duration:     264 channel:  1 velocity:  56 pitch:  62 <<<
```

Object *without4b* attempts to filter those kind of duplicate values.

```
Name            without4b
Clock unit      100
Number          100
Rhythm          (without (anything inside 1.5 2.5)
                        (random-value 1.0 3))
Pitch           (without (duplicates)
                        (walk c4 (random-value -2 2) c3 c5))
Velocity        (rv p f)
Channel         1
```

Using this specification for pitch, it is not possible to produce output that does not have duplicate values. An error message is the result.

One way to resolve the error condition is to limit the portion of the output that cannot have duplicate values. If the keyword *diversity* is bound to 3, there cannot be duplicates in

a group of 3 values. After that, the count starts again. Values 1-3 will be unique, values 4-6, etc. Values 3 and 4 could however be the same.

Object *without4c* produces groups of 6 pitch values that contain no duplicates.

```
Name            without4c
Clock unit      100
Number          100
Rhythm          (without (anything inside 1.5 2.5)
                    (random-value 1.0 3))
Pitch           (without (duplicates :diversity 6)
                    (walk c4 (random-value -2 2) c3 c5))
Velocity        (rv p f)
Channel         1
```

Without more than …
Sometimes in a range of values, you want no more than 1 or 2 of a certain value. The tool *more-than* can be used together with *without* to achieve this. `(more-than 1 2)` indicates that only one value of 2 is allowed. Any additional 2s will be rejected.

```
20 values from:
(without (more-than 1 2) (random-value 1 5))
          4          3          4          5          4
          3          4          4          5          2 <<<
          1          1          3          4          4
          3          4          4          4          1
```

More-than can have any number of these constraints. In object *without5*, several numbers are constrained in the pitch parameter. Of the 100 events calculated, no more than three may have the pitch 60, three may have the value 64 and three the value 67. If the generator only produced two values for 60, then the final output will only have two values for 60. The constraint is an upper limit on the number of times a value may occur.

```
Name            without5
Clock unit      100
Number          100
Rhythm          (without (more-than 1 10)
                    (random-choice '(1 1.5 2 3 10)))
Pitch           (without (more-than 3 60 3 64 3 67)
                    (random-value c4 c5))
Velocity        (rv p f)
Channel         1
```

In the rhythm parameter of *without5*, only one long rhythmic value is allowed. All of the rest will be significantly shorter.

Generating with …
The generator *with* is similar to *without* except that it includes values rather than rejecting them. The same filters, constraint expressions, etc. are available for use in *with* as were available for *without*.

`(with '(60 64 67) (random-value 48 72))` only allows the values 60, 64, and 67 to be used. This is demonstrated in object *with1*.

Instead allowing specific pitches, specific pitch classes can be allowed. A pitch class is a representation of values between 0-11, where c is 0, c# is 1, etc.

`(pitch-class 0 3 4 7 8 11)` allows any Midi note number to be used for pitch that can be reduced to one of the specified pitch classes: 0, 3, 4, 7, 8, 11. All other pitches are rejected.

```
20 values from:
(midi->notename (with (pitch-class 0 3 4 7 8 11) (random-value 48 72)))
       g3        b3        e3        c5        c3
       c4       g#4        e4        c5        g3
       g4        c3        g3        g4        e3
       c5       d#3        c3        e3       g#3
```

In the above example, the pitch classes correspond to C, D#, E, G, G#, and B. The results are printed as pitch names to show the relation to the specified pitch classes.

Object *with2* uses a pitch-class specification for the pitches.

```
Name            with2
Clock unit      100
Number          50
Rhythm          (walk 2.0 (random-value -0.5 0.5) 1 3)
Pitch           (with (pitch-class 0 3 4 7 8 11) (random-value 48 72))
Velocity        (rv p f)
Channel         1
```

Most other filters defined in the AC Toolbox can be used as constraints for *with* and *without*. These filters are included in the list of tools for the category *Filters* in the **Annotated Index** of the AC Toolbox.

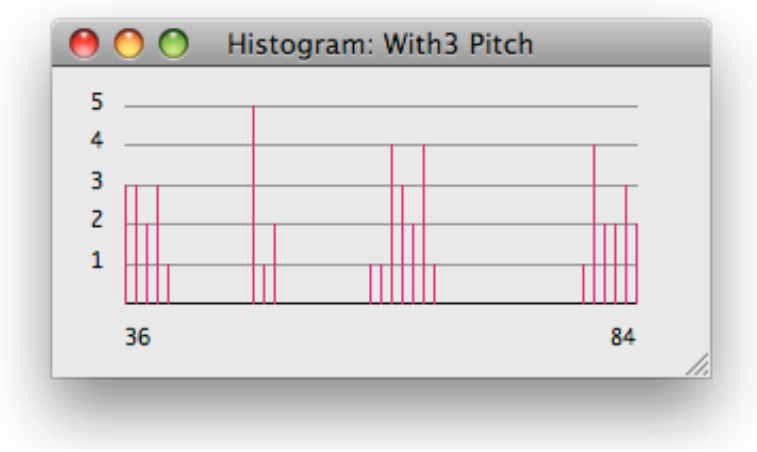
One of these filters is *bank*. A bank of band-pass filters can be specified. It allows values within any of several regions. Any number of regions can be defined.

```
20 values from:
(with (bank 1 5 10 12 18 19) (from 1 20))
         1         2         3         4         5
        10        11        12        18        19
         1         2         3         4         5
        10        11        12        18        19
```

Object *with3* uses *bank* to allow certain pitches to be accepted.

```
Name            with3
Clock unit      100
Number          50
Rhythm          (walk 2.0 (random-value -0.5 0.5) 1 3)
Pitch           (with (bank 36 40 48 50 59 65 79 84)
                    (random-value 36 84))
Velocity        (rv p f)
Channel         1
```

The four regions specified for *with3* can be clearly seen in a histogram of a possible result.



Using the filter method
With and *without* are generators and are used to filter values as they are being produced. If *with* or *without* does not approve of a value, a new value is generated by whatever generator is feeding one of those two filters.

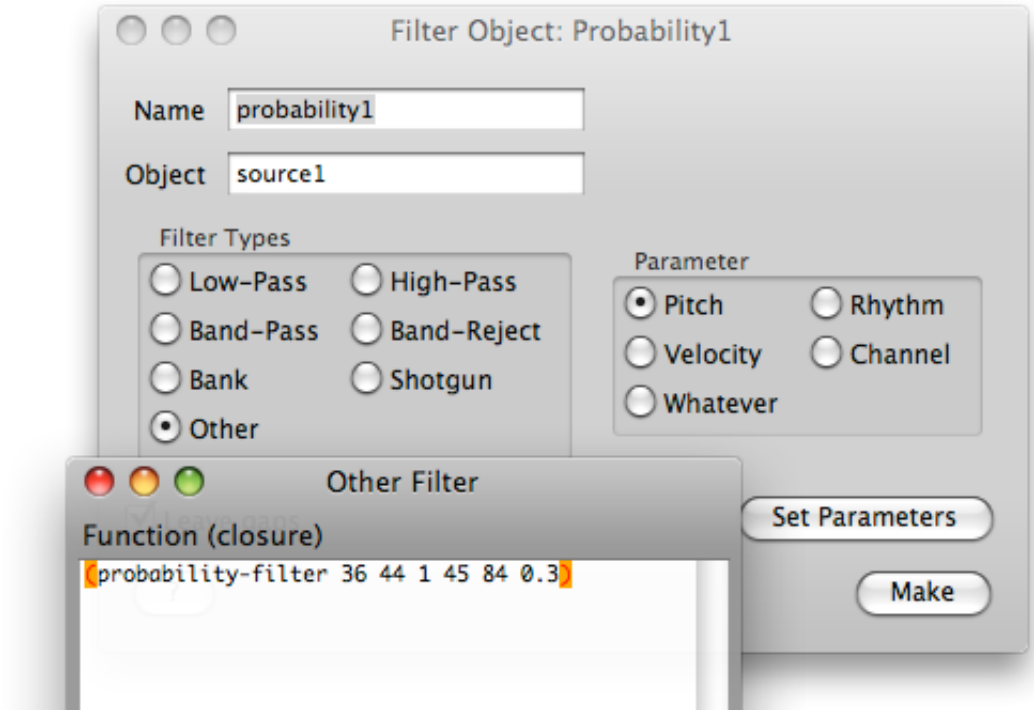
The filter method (**Methods>Filter>Object)** filters an object that has been made. The result in most cases will be fewer values than in the original object. While various types of objects can be filtered with this method, this tutorial will only discuss filtering sections.

Filtering with probabilities
The *probability-filter* is similar to *bank*: values within several regions can be included. Not all values within those regions need be accepted. Only a certain percentage of values in a region will be allowed. In this way, it is possible to filter a section to have fewer high pitch values or to emphasize the mid-range. Each region includes a probability factor (0-1) for the probability that a factor in a specific region should be included. 0 is never, 1 is always.

Object *source1* will be filtered by the probability-filter. It includes a wide, random range of pitches.

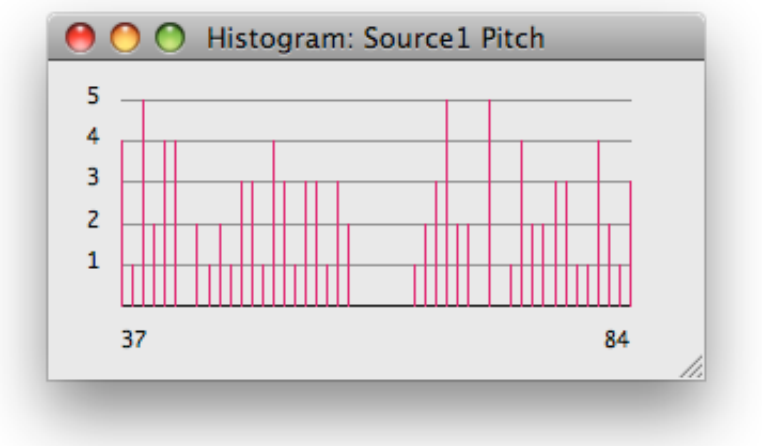
```
Name            source1
Clock unit      100
Number          100
Rhythm          (random-value 1.0 3)
Pitch           (random-value 36 84)
Velocity        (rv p f)
Channel         1
```

Object *probability1* will filter the pitches of *source1*. Filter type *Other* and parameter *Pitch* should be chosen. When the *Set Parameters* button is clicked, a window opens where the expression for the probability-filter can be entered.

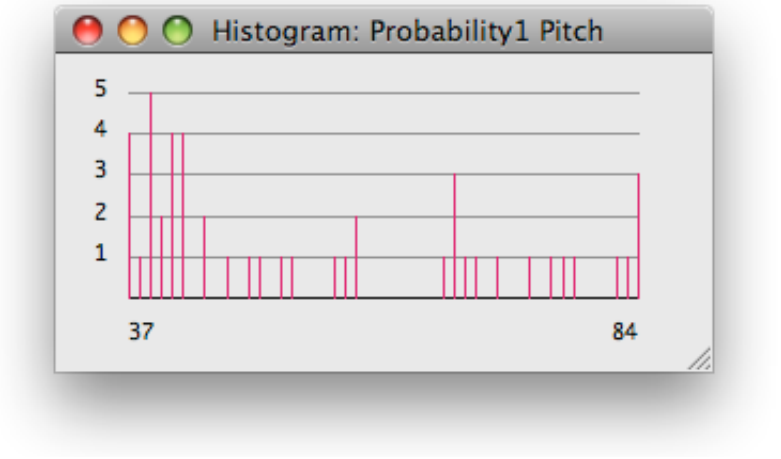


Pitches in the region between and including 36-44 will always be passed (1 is always). Pitches in the range 45-84 will pass 30% of the time (probability is .3). The result is that the original section is filtered to have fewer high values while maintaining all of the low ones.

The histogram for a possible pitch outcome of *source1*:



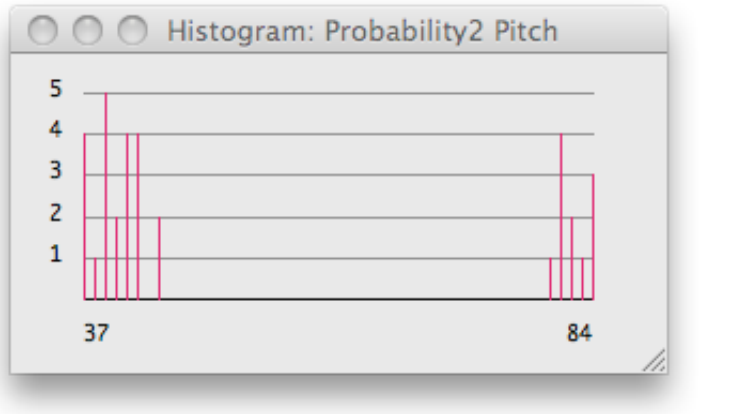
The histogram for the filtered version, *probability1*:



Object *probability2* filters the pitches of *source1* with the following expression:

```
(probability-filter 36 44 1 80 84 1)
```

This passes all values between 36-44 and 80-84. All the rest of the pitches are rejected.



The probability-filter can also be used with the generators *with* and *without*.

Filter if ...
Instead of applying a filter to all events in a section, it is possible to limit the use of the filter to events that meet some condition. This can be done with *filter-if.* E.g. if pitch is in

the range 36-60, remove events with short rhythmic values (in the range of 100-200 ms). Not all short events are removed, only the ones with pitches in the specified range.

The parameters for *filter-if* are:

```
(FILTER-IF IF-PARAMETER IF-FILTER THEN-PARAMETER THEN-FILTER)
```

If-parameter and *then-parameter* can be pitch, velocity, channel, or rhythm.
If-filter and *then-filter* can be any filter than works with one parameter.

Filter-if can only be used with sections. The filter type is O*ther* and the parameter is W*hatever*. By choosing *Whatever*, all event information is available to the filter method to do its work.

Object *filter1* filters object *source1*. The filter expression for the *Other* filter is:

```
(filter-if 'pitch (band-pass 36 60) 'rhythm (band-reject 100 200))
```

If a band-pass filter with limits 36-60 is applied to the pitch parameter of the input section and there is a positive result, then the band-reject filter is applied to the rhythm parameter. Note that the values for the rhythm parameter are expressed in milliseconds. All other values are passed without any filtering.

To state it in a different fashion: the durations for events with pitches in the range 36-60 are tested. If the duration is in the range 100-200 ms, the event is rejected. In all other cases, the event is passed.

The first five events of a possible outcome for *source1*:

```
1   time: 0      duration:  298 channel:  1 velocity: 79 pitch:  54
2   time: 298    duration:  117 channel:  1 velocity: 78 pitch:  51 <<<
3   time: 415    duration:  189 channel:  1 velocity: 78 pitch:  73
4   time: 604    duration:  181 channel:  1 velocity: 59 pitch:  81
5   time: 785    duration:  210 channel:  1 velocity: 76 pitch:  84
```

Event 2 contains pitch and duration values that meet the criteria: pitch between 36-60 and rhythm between 100-200.

In the first five events of a possible outcome for *filter1*, that event has been removed.

```
1   time: 0      duration:  298 channel:  1 velocity: 79 pitch:  54
2   time: 415    duration:  189 channel:  1 velocity: 78 pitch:  73
3   time: 604    duration:  181 channel:  1 velocity: 59 pitch:  81
4   time: 785    duration:  210 channel:  1 velocity: 76 pitch:  84
5   time: 995    duration:  235 channel:  1 velocity: 78 pitch:  77
```

The next example of *filter-if* involves looking at the Midi channel (that could be assigned to a different instrument). Using *filter-if*, the filter could be applied to one instrument and not the other.

Object *source2* will be used for the next example. It produces events in two Midi channels.

```
Name           source2
Clock unit     100
Number         20
Rhythm         (rv 1 2)
Pitch          (rv 60 80)
Velocity       mf
Channel        '(1 2)
```

Object *filter2* filters the pitches of channel 1. The filter expression is:

```
(filter-if 'channel (value-pass 1) 'pitch (band-pass 65 75))
```

Value-pass is true if the channel is 1. In that case, pitches in the range 65-75 pass. All other events in channel 1 are rejected. All events in channel 2 are passed.

The first five events of *source2*:

```
1  time: 0       duration:  200 channel:  1 velocity:  64 pitch:  65
2  time: 200     duration:  200 channel:  2 velocity:  64 pitch:  74
3  time: 400     duration:  100 channel:  1 velocity:  64 pitch:  60
<<<
4  time: 500     duration:  100 channel:  2 velocity:  64 pitch:  76
5  time: 600     duration:  200 channel:  1 velocity:  64 pitch:  61
<<<
```

Events 3 and 5 are in channel 1 but their pitches are not in the desired range.

In the first five events of a possible outcome for *filter2*, those two events have been removed:

```
1  time: 0       duration:  200 channel:  1 velocity:  64 pitch:  65
2  time: 200     duration:  200 channel:  2 velocity:  64 pitch:  74
3  time: 500     duration:  100 channel:  2 velocity:  64 pitch:  76
4  time: 800     duration:  200 channel:  2 velocity:  64 pitch:  69
5  time: 1000    duration:  200 channel:  1 velocity:  64 pitch:  75
```

Meeting all conditions
Filter-and allows several conditions to be specified. It only passes an event if all of the conditions have been met. Otherwise an event is rejected.

In *filter-if*, a filter is only applied if a condition is met (e.g. pitch is in a certain range). Since other events are not filtered, they are also not rejected.

In *filter-and*, all conditions must be met (e.g. pitch is in a certain range and rhythm is less than 100). Otherwise an event is rejected.

Filter-and allows a filter expression to involve several parameters. *Filter-and* is used in a filter of type *Other* with the parameter set to *Whatever*.

The parameters for *filter-and* are:

```
(FILTER-AND PARAMETER FILTER ... PARAMETER-N FILTER-N)
```

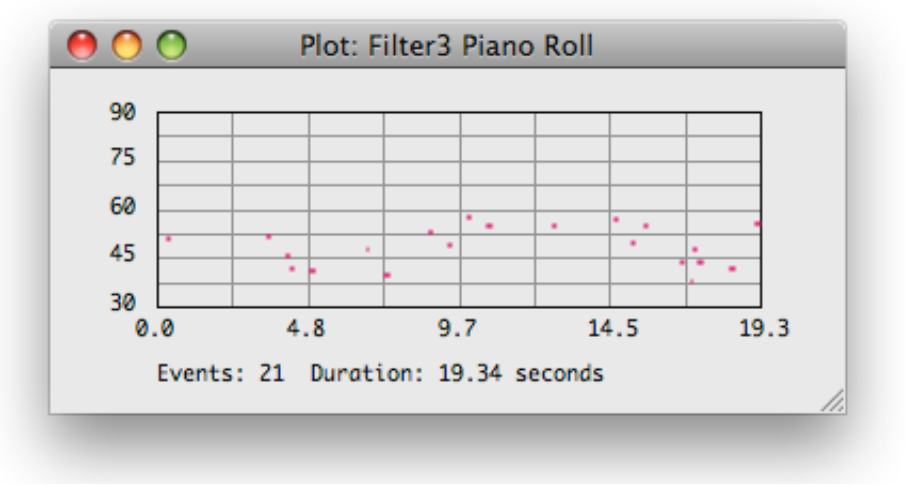
The parameter and filter combinations are expressed in a similar fashion to *filter-if*. Any number of parameter and filter combinations can be used in an expression. Filters can be almost any type of filter. Parameters can be pitch, velocity, channel, or rhythm.

Object *filter3* filters object *source1*. The filter expression for the *Other* filter is:

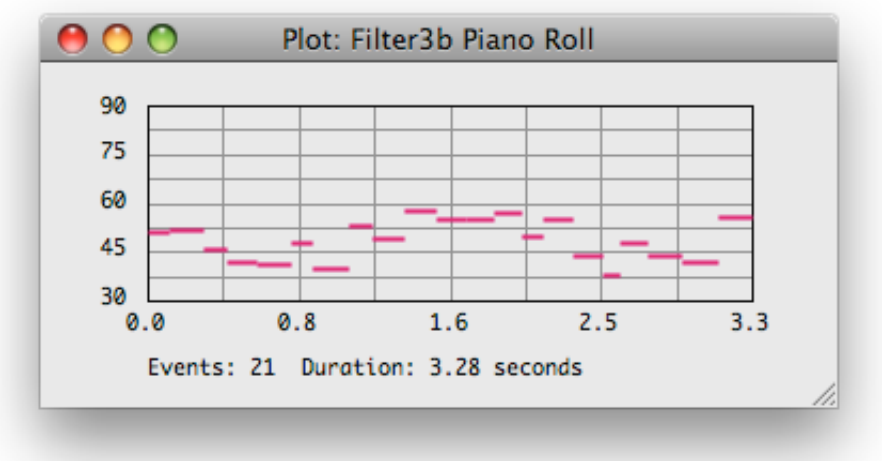
```
(filter-and 'pitch (band-pass 36 60) 'rhythm (low-pass 200))
```

In the above expression, if pitch is in the range 36-60 and rhythm is <= to 200 ms, an event is passed. Otherwise it is rejected.

The result of applying *filter3* to the possible outcome of *source1* is that only 21 events remain. The section has a lot of 'empty space'.



If this 'empty space' is not desired, it is possible to squeeze all of the remaining events together. To do this, uncheck the box *Leave gaps* in the filter dialog. This is what happens in object *filter3b*.



Object *filter4* filters object *source2* that has events in two Midi channels. The filter expression is:

```
(filter-and 'channel (value-pass 1) 'pitch (low-pass 70))
```

This passes events in channel 1 that have pitch values <= 70.

Object *source3* makes 4-note chords.

```
Name            source3
Clock unit      100
Number          50
Rhythm          (rv 1 2)
Pitch           (make-chord (rv 60 80) 4)
Velocity        mf
Channel         1
```

It is filtered by object *filter5*. The filter expression for *filter5* is:

```
(filter-and 'rhythm (low-pass 100) 'pitch (band-pass 65 75))
```

Rhythm must be <= to 100 and pitch must be in the range 65-75. The band-pass filter for pitch will filter out pitch values in the chords that are not in the proper range. This results in chords that may have less than four pitches.

Meeting one condition
Another way to combine conditions is with *filter-or*. It passes an event if at least one of its conditions is true. It is not necessary that they all be true. In this way, conditions such as "allow either short notes or high pitches" are possible.

The input format is similar to *filter-and*.

Object *filter6* filters object *source1*. The filter expression for the *Other* filter is:

```
(filter-or 'rhythm (low-pass 200) 'pitch (band-pass 70 84))
```

This expression allows events with either short notes (rhythms <= 200 ms) or pitches in the range 70-84.

Object *filter7* adds a condition for velocity as well:

```
(filter-or 'rhythm (low-pass 200)
           'pitch (band-pass 70 84)
           'velocity (high-pass 79))
```

Events are passed with short notes or high pitches or somewhat high velocity values (>= 79).

Filtering repetitions
The *repetition-filter* filters out immediate repetitions. If two successive values are the same, only the first one is allowed.

Object *source4* can contain repetitions since only 3 rhythmic values and 5 velocity values are allowed.

```
Name           source4
Clock unit     100
Number         100
Rhythm         (random-value 1 3)
Pitch          (random-value 36 84)
Velocity       (random-choice '(pp p mp f ff))
Channel        1
```

Object *filter8* filters the rhythm of *source4*. The filter expression entered in the *Other* filter is:

```
(repetition-filter)
```

Object *filter9* uses *filter-and* to apply the repetition filter to both rhythm and velocity.

```
(filter-and 'rhythm (repetition-filter) 'velocity (repetition-filter))
```

This expression only allows events where there is no repetition in both the rhythm and the velocity.

The repetition-filter can also be used with the generators *with* and *without*, and with the tool *filter-stockpile.*

Filtering pitch intervals
The filter *pitch-interval-filter* can be applied to sections. If there is only a monolinear series of events, the intervals of successive pitches are compared. The event containing the second pitch of the undesired interval is removed.

Object *source5a* is a monolinear series of events.

```
Name           source5a
Clock unit     200
Number         100
Rhythm         (random-choice (from 1 3 :step 0.5))
Pitch          (random-value 60 72)
Velocity       mf
Channel        1
```

In 10 values from a possible outcome of *source5a*, pitch repetitions can be seen.

```
11 time: 4800   duration:  400 channel:  1 velocity:  64 pitch:  69
12 time: 5200   duration:  200 channel:  1 velocity:  64 pitch:  69 <<<
13 time: 5400   duration:  600 channel:  1 velocity:  64 pitch:  66
14 time: 6000   duration:  600 channel:  1 velocity:  64 pitch:  60
15 time: 6600   duration:  300 channel:  1 velocity:  64 pitch:  66
16 time: 6900   duration:  500 channel:  1 velocity:  64 pitch:  68
17 time: 7400   duration:  600 channel:  1 velocity:  64 pitch:  70
18 time: 8000   duration:  600 channel:  1 velocity:  64 pitch:  70 <<<
19 time: 8600   duration:  300 channel:  1 velocity:  64 pitch:  66
20 time: 8900   duration:  600 channel:  1 velocity:  64 pitch:  66 <<<
```

Object *filter10* uses the *pitch-interval-filter* to remove those repetitions. The *pitch-interval-filter* must use the *Whatever* parameter. The filter expression for *filter10* is:

```
(pitch-interval-filter :intervals 0)
```

When only one interval is to be removed, it does not need to be in a list.

Object *filter11* will remove a repetition and an interval of a perfect fifth. The filter expression is:

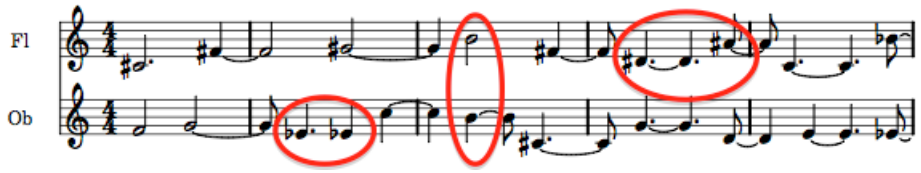
```
(pitch-interval-filter :intervals '(0 7))
```

Since more than one interval is specified, the intervals are included in a list. Any number of intervals can be specified.

Object *source5b* uses the same input specification as *source5a* except that it uses Midi channel 2. Object *source5-together* combines those two objects in parallel.

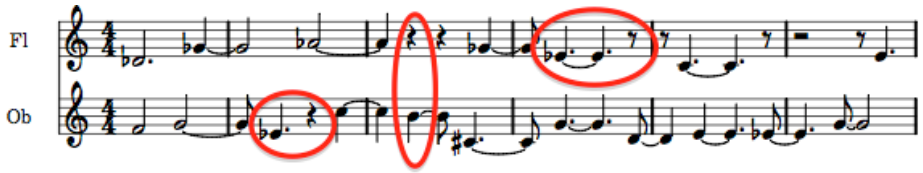
The section *source5-together* is no longer a monolinear string. Pitches that happen at the same time will also be examined for the undesired intervals. This only works if the pitches were not made with generator *make-chord*.

A representation of part of a possible outcome for *source5-together* where the notes in the two Midi channels are labeled *Fl* and *Ob*:



Occurrences of intervals 0 and 7 have been circled.

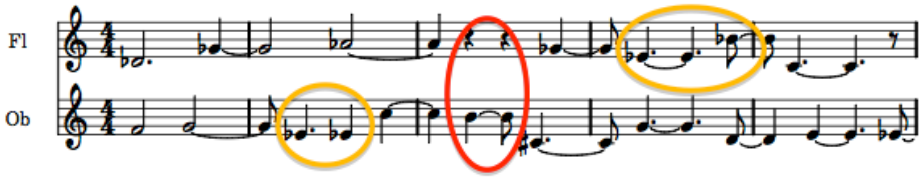
Object *filter12* will remove these undesired intervals. The filter expression is the same as in *filter11.*



The differences in the note spellings are a decision of FOMUS, the music formatting tool that was used to prepare the notation for these examples (see Tutorial 22).

It is also possible to only remove simultaneous intervals. In that case, keyword *:next* should be bound to *nil*. This is happens in object *filter13*. The filter expression is:

```
(pitch-interval-filter :intervals '(0 7) :next nil)
```



The repetitions were not removed. Only the simultaneous B in the flute was removed.

Summary

Generators
without, with

Tools
anything, duplicates, more-than, bank, probability-filter, filter-if, filter-and, filter-or, value-pass, repetitions-filter, pitch-interval-filter

Tutorial 24

Pitch and intervals

Some approaches to pitch organization are based on interval structures. This tutorial looks at a selection of generators and tools that are explicitly concerned with intervals. Of course, pitch structures can be made with other tools and generators than the ones discussed here.

A frequency-based (spectral) approach to pitch organization was discussed in Tutorial 20.

The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 24 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Random intervals
A simple approach to selecting intervals is to randomly choose from available intervals and create the next value by adding that interval to the previous one. If it is within specified limits, the new value is accepted. Otherwise a new choice of interval is made.

Generator *random-intervals* uses the above approach. Parameters for the generator are:

```
(RANDOM-INTERVALS LOW HIGH &REST INTERVALS)
```

All values returned by this generator should be in the range *low* – *high*. Any number of intervals can be entered. The first result is randomly chosen between *low* and *high*.

An example of a possible output for this generator:

```
20 values from:
(random-intervals c4 f#6 1 3 4)
        86        83        87        83        80
        81        77        73        70        67
        71        67        63        64        67
        70        71        67        66        69
```

The first value is between c4 (60) and f#6 (90). One of the intervals was added to this value. Note that the intervals may go up or down. The only possible intervals in this example are 1, 3, and 4.

Object *random1* uses the above specification for pitches.

To allow repetitions, the interval 0 can be specified. This happens in object *random2*. The specification is:

```
(random-intervals c4 f#6 0 1 3 4).
```

Object *random3* adds a larger interval to the possibilities.

```
(random-intervals c3 f#6 1 3 4 6 11)
```

Allowing certain intervals
Generator *allow-interval* produces a value with another generator. If the distance to the previous value is one of the allowed intervals, it is accepted. Otherwise, the generator should produce a new value until an acceptable one is found. If that does not happen soon enough, the Toolbox gives an error message.

Parameters for *allow-interval* are:

```
(ALLOW-INTERVAL GENERATOR INTERVAL-OR-LIST &KEY PITCH-CLASS STOP)
```

Generator can be any generator. *Interval-or-list* should be a single interval or a list of intervals that are allowed. The absolute value of the distance between values is used for the comparison with the allowed intervals.

```
20 values from:
(allow-interval (random-value c4 f#6) '(1 3 4))
        80        76        77        78        75
        71        67        64        68        67
        68        71        68        71        72
        75        74        77        78        82
```

An initial value is chosen by the generator. All subsequent values should be at a distance of one of the allowed intervals (positive or negative). In the above example all subsequent values have a distance of 1, 3, or 4.

Object *allow1* uses the above specification for pitch.

If the distances should not be sensitive to octaves, the keyword *pitch-class* should be bound to *t*. In that case, all distances are reduced to pitch classes before the comparison is made.

```
20 values from:
(midi->notename (allow-interval (random-value c4 f#6)
                                '(1 3 4)
                                :pitch-class t))
      d#4        d4       c#4        e5        f6
      f#4       d#4       f#5        f4       c#4
       d5       f#5       d#4        c6        e5
       c5       d#5        e4        g4       f#6
```

Object *allow2* binds *:pitch-class* to *t.* The specific result of this is a greater range of pitch values.

Almost any generator can be used to create the values. Object *allow3* uses *exponential-value*. Object *allow4* adds the *pitch-class* keyword to the specification of *allow3*. This will probably result in a greater range of pitch values.

Allow-interval can work with quartertones. Quartertones are expressed as Midi note numbers with fractional parts rounded to units of 0.5 (see Tutorial 18 for more information about microtones).

Object *allow5a* does not use *allow-interval*. It produces random pitch values, rounded to units of 0.5. The pitch specification is:

```
(random-value (float c4) f#6 :round 0.5)
```

`(float c4)` is needed to cause *random-value* to produce floating-point values. Keyword *round* quantizes values to the unit specified, in this case 0.5. This section contains random quartertone pitches.

Object *allow5b* uses the above specification within *allow-interval*. The interval-list includes fractional intervals.

```
(allow-interval (random-value (float c4) f#6 :round 0.5)
                '(1 2.5 3 3.5 4))
```

Object *allow5c* adds the *pitch-class* keyword. This can work with fractional values.

```
(allow-interval (random-value (float c4) f#6 :round 0.5)
                '(1 2.5 3 3.5 4)
                :pitch-class t)
```

Blocking intervals

Generator *block-interval* has essentially the same syntax as *allow-interval* but does the opposite. It rejects values produced by a generator if the distance to the previous value is one of the forbidden intervals.

If there is only one forbidden interval, a constant value can be used. Otherwise a list of intervals should be entered.

```
20 values from:
(block-interval (random-value 1 5) 0)
         1         4         5         4         5
         2         4         2         3         1
         3         5         3         2         5
         1         2         4         3         5
```

In the above example, all repetitions (interval 0) are blocked. Since there is only one forbidden interval, it is expressed as a constant value.

```
20 values from:
(block-interval (random-value 1 5) '(0 1))
         4         2         4         1         4
         2         5         2         4         1
         5         3         5         2         5
         3         5         3         1         4
```

Here, two intervals (0 and 1) are blocked. All other intervals are allowed.

Object *block1* blocks intervals of 0, 5,and 7 for pitch. The specification is:

```
(block-interval (random-value c4 f#6) '(0 5 7))
```

Object *block2* uses the *pitch-class* keyword. This has the same meaning as in *allow-interval*. Distances are reduced to pitch-classes before comparing them to the interval list.

Making chords

A chord is represented as a list of pitches. It can be generated with *make-chord*. The parameters for the generator are:

```
(MAKE-CHORD SOURCE-OBJECT SIZE-OBJECT &KEY BLOCK PITCH-CLASS STOP
ALLOW-REPEATS LIMIT)
```

Source-object can be generator to produce pitch values. *Size-object* is something to indicate the number of notes in a chord. Object *make-chord1* produces 4-note chords with random pitches. This could result in a chord containing duplicate pitches.

```
(make-chord (random-value c4 c5) 4)
```

Keyword *block* can be bound to a constant or a list to specify which interval or intervals in the chord should be blocked. Blocking 0 prevents repetitions of the same note in a chord. A new value is generated for the chord to replace the rejected one.

```
10 values from:
(make-chord (random-value c4 c5) 4 :block 0)
( 72 67 69 70 )
( 62 70 66 61 )
( 68 66 71 72 )
( 71 67 65 64 )
( 65 71 62 67 )
( 69 71 63 70 )
( 69 72 68 62 )
( 68 72 62 60 )
( 67 61 62 68 )
( 67 68 61 60 )
```

The above chords do not contain duplicate pitches in the same chord. Object *make-chord2* uses this specification.

Object *make-chord3* blocks the intervals 0, 3, and 7. The specification is:

```
(make-chord (random-value c4 c5) 4 :block '(0 5 7))
```

Producing chords with random numbers could mean that two successive chords contain one or more of the same values. If this not desired, keyword *allow-repeats* should be bound to *nil*. This happens in object *make-chord4*.

```
10 values from:
(make-chord (random-value c4 c5) 4 :block '(0 5 7) :allow-repeats nil)
( 65 64 67 68 )
( 69 72 66 63 )
( 71 62 68 65 )
( 64 72 61 63 )
( 70 68 69 67 )
( 62 64 66 65 )
( 69 61 71 70 )
( 67 65 64 66 )
( 62 71 60 61 )
( 64 67 66 65 )
```

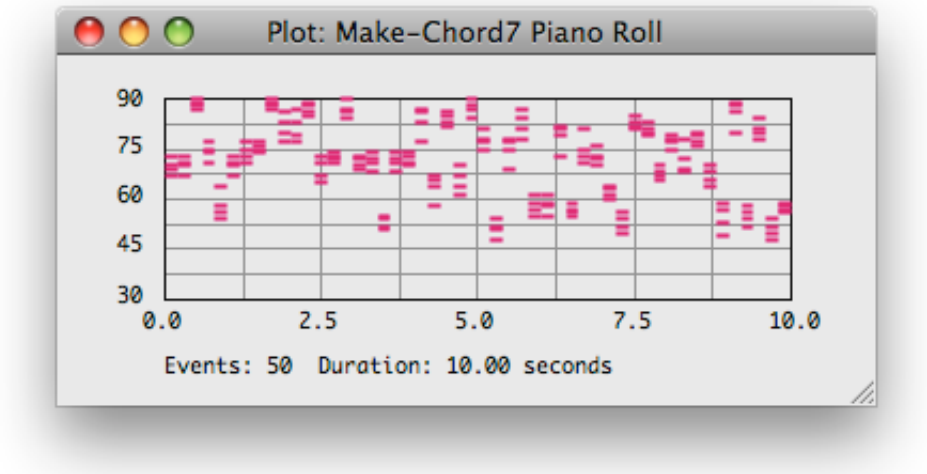
Object *make-chord5* binds the keyword *pitch-class* to *t*. This forces the generator to reduce the values to pitch classes before looking for blocked intervals.

Object *make-chord6* increases the range of the random generator creating pitches. *Pitch-class* is bound to *t*, and *allow-repeats* is bound to nil.

A final way of limiting intervals in a chord is to limit the allowed distance from the first pitch chosen. The keyword *limit* can be bound to a distance limiting the other pitches to be in a range of ± that distance.

```
10 values from:
(make-chord (random-value c3 f#6)
            4
            :block '(0 5 7)
            :allow-repeats nil
            :pitch-class t
            :limit 6)
( 51 49 50 48 )
( 80 84 82 81 )
( 86 90 82 88 )
( 60 68 58 64 )
( 66 64 62 65 )
( 63 64 66 65 )
( 76 73 72 70 )
( 76 77 74 80 )
( 65 71 73 69 )
( 62 56 52 58 )
```

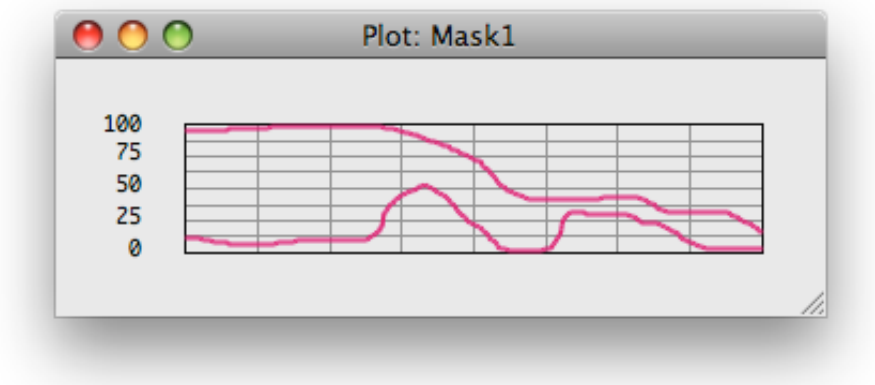
The first value chosen is the last one in the list. Object *make-chord7* uses this specification. It keeps each chord within a certain range.



Blocking intervals in a mask
Convert is the general tool for converting shapes, masks, functions, and lists to be a list of some arbitrary length in an arbitrary range.

When converting masks, it is also possible to block certain intervals. If a portion of a mask is too narrow, it may not be possible to produce one of the allowed intervals. In that case, an error message is returned.

Object *mask1* will be used for these examples though any mask that is not too narrow should work.



Object *convert1* converts this mask to a 100 values in the range c3-c5. This is a standard use of *convert*. The specification is:

```
(convert mask1 (from-number) c3 f#5)
```

The parameter specification for *convert* is:

```
(CONVERT OBJECT N LOW HIGH &OPTIONAL GENERATOR INTERVAL-OR-LIST PITCH-
CLASS STOP PREVIOUS ROUND)
```

The optional parameters can be used with masks. *Interval-or-list* is a constant value for an interval or a list of intervals to be blocked. This provides some control over the resolution of the conversion and over the intervallic content of the result.

Since *interval-or-list* is an optional parameter, all parameters to its left must be entered. If *t* is entered for *generator*, the default random generator is used for selecting values between the boundaries of the mask.

Object *convert2* blocks repetitions (interval 0) in the conversion process. Any repetitions are discarded and a new value is sought. If the mask is too narrow there may be a point where only repetitions are possible. This would produce an error message.

```
(convert mask1 (from-number) c3 f#5 t 0)
```

Object *convert3* blocks three intervals: 0, 5, and 7. The specification is:

```
(convert mask1 (from-number) c3 f#5 t '(0 5 7))
```

There is a tool called *convert2*. It is similar to *convert* except that it uses keywords instead of optional parameters. This means you do no need to fill in a value for a generator if you want to specify intervals to block. You do need to bind the intervals to :round.

Object *convert3b* uses tool *convert2.*

```
(convert2 mask1 (from-number) c3 f#5 :block  '(0 5 7))
```

Using a grid

After an object has been made, it is possible to force its values to conform to a grid. Values that are not in the grid are replaced with values that are. The transformer *funnel* forces values to be one of the values in a grid. It can be used with the transform method (**Methods>Transform>Object**).

The input specification for funnel is:

```
(FUNNEL LIST)
```

List is a list of values in the grid. That list can be specified or generated.

One way to make a grid is with the tool *generate-range*. It applies a generator (that should produce ascending values) until a certain limit has been reached. It returns a list of the values.

```
List returned from:
(generate-range (major c3) f#5)

(
        48        50        52        53        55
        57        59        60        62        64
        65        67        69        71        72
        74        76        77
)
```

Major generates ascending values that correspond to a major scale.

Object *c-major* is a stockpile made with this specification. It contains values corresponding to a c-major scale that can be used as a grid for *funnel.* Object *convert4* uses the transform method to force the pitches of *convert3* through the funnel containing c-major values.

A possible list of pitches for section *convert3*:

```
(57 60 70 56 53 68 51 61 71 60 68 77 55 53 70 60 64 77 68 53 70 56 73
56 59 68 52 61 75 55 58 64 52 53 73 72 63 71 65 67 68 72 70 66 63 61 65
69 58 61 65 64 61 57 60 57 61 49 55 58 49 59 55 52 54 57 59 58 59 58 60
58 60 57 60 59 61 59 56 57 55 57 56 54 57 54 56 54 52 49 50 52 53 52 56
54 50 52 50 49)
```

A list of the pitches for *convert4* (after they have been forced to be one of the values in the c-major grid):

```
(57 60 71 55 53 67 50 60 71 60 67 77 55 53 69 60 64 77 67 53 71 55 74
55 59 67 52 62 76 55 57 64 52 53 74 72 64 71 65 67 69 72 69 67 62 60 65
69 59 62 65 64 62 57 60 57 60 50 55 57 50 59 55 52 53 57 59 59 59 57 60
57 60 57 60 59 62 59 55 57 55 57 55 53 57 53 55 55 52 50 50 52 53 52 55
55 50 52 50 48)
```

Another-scale can be specified by *walk* with a list of intervals. These intervals are added to each previous value, thus creating an ascending series of values suitable for use with *generate-range*.

```
List returned from:
(generate-range (walk c2 '(1 2 1 3 1 4 1 3 1 2 1)) f#6)

(
        36        37        39        40        43
        44        48        49        52        53
        55        56        57        59        60
        63        64        68        69        72
        73        75        76        77        79
        80        83        84        88        89
)
```

Object *convert5* transforms the pitches of *convert3* with *funnel* using the *another-scale* object. The result follows the contours of the mask but the pitch alignment is determined by a different grid.

A list of pitches for *convert5* reflects the grid of *another-scale*.

```
(57 60 69 56 53 68 52 60 72 60 68 77 55 53 69 60 64 77 68 53 69 56 73
56 59 68 52 60 75 55 59 64 52 53 73 72 63 72 64 68 68 72 69 64 63 60 64
69 57 60 64 64 60 57 60 57 60 49 55 59 49 59 55 52 55 57 59 57 59 57 60
57 60 57 60 59 60 59 56 57 55 57 56 53 57 55 56 55 52 49 49 52 53 52 56
53 49 52 49 49)
```

Funnel is a transformer and changes values. It does not filter out undesirable values. A tool for filtering undesirable pitch intervals from a section is the *pitch-interval-filter* discussed in Tutorial 23.

Sieves
A sieve is a series of ascending integers, such as 1,3,5,7 or 4,7,10. Sieves can be joined to make more complicated sieves such as 1,3,4,5,7,10. Xenakis used sieves for rhythmic structures in *Psappha* and for pitch structures in many compositions.

Theoretically, sieves are endless. For practical purposes a slice needs to be taken from this endless series of numbers.

The sieve tool in the AC Toolbox produces a list of values that represent a slice of a sieve. The parameters for the tool are:

```
(SIEVE MOD START LOWER HIGHER)
```

The sieve will start at number *start*. New values occur at every *mod* number of steps. Values between *lower* and *higher* are returned in a list.

```
List returned from:
(sieve 2 1 1 10)

(
         1         3         5         7         9
)

List returned from:
(sieve 2 2 1 10)

(
         2         4         6         8        10
)
```

Sieves can be joined with the tool *sieve-union*. It takes 2 or more sieves as arguments. It returns a list with all values from all of the sieves. If there are duplicate values in the sieves, a value is only returned once. Joining sieves together can created interesting intervallic relationships.

```
List returned from:
(sieve-union (sieve 3 1 1 10) (sieve 4 1 1 10))

(
         1         4         5         7         9
        10
)
```

Sieve can be abbreviated to *sv*. *Sieve-union* can be shortened to *su*.

```
List returned from:
(su (sv 2 1 10 20) (sv 5 1 10 20))

(
        11        13        15        16        17
        19
)
```

Object *sieve1* is a stockpile created from

```
(sieve-union (sieve 3 1 c2 f#6) (sieve 4 1 c3 f#5))
```

The resulting sieve is:

```
        37        40        43        46        49
        52        53        55        57        58
        61        64        65        67        69
        70        73        76        77        79
        82        85        88
```

Object *sieve-section1* uses *sieve1* as pitch values. When this section is played, the elements of the sieve are played in order.

Object *sieve-section2* uses generator *series-choice* to make random choices from the sieve. *Series-choice* does not repeat until all values have been used once.

Sieves can be used as a grid, for example with *funnel*.

Using intervals and a grid

Scale-intervals allows intervals to be chosen with a generator or read from a list. The chosen interval is added to the previous value. If the new value is one of the allowed values in the grid (scale), it is used. Otherwise another interval is chosen. The first value is specified by the user.

This generator works with intervals, but limits the results to be members of a scale or pitch structure.

The parameters for *scale-intervals* are:

```
(SCALE-INTERVALS FIRST INTERVALS ALLOWED-VALUES &KEY STOP)
```

First is the first value. It can be a generator or a constant value. *Intervals* is a generator, stockpile, or list of intervals. Unlike some of the other generators discussed in this tutorial, positive and negative intervals are entered separately. *Allowed-values* should be a list or stockpile of allowed output values. When an interval is added to a previous value, the result is compared to this list or stockpile. If *first* is not one of the *allowed-values*, then there should be an interval available that could be added to *first* to make one of the allowed values.

```
10 values from:
(scale-intervals 60 '(1 2 3 -3 -2 -1) (from 60 72))
        60        61        63        66        63
        61        60        61        63        66
```

When *intervals* is a list, it is read in order. In the above example, the first interval is 1, the next is 2, etc.

If a sieve is being used for *allowed-values*, it may be useful to see what intervals are present in the sieve:

```
List returned from:
(get-intervals sieve1 :between t)

(
         3         3         3         3         3
         1         2         2         1         3
         3         1         2         2         1
         3         3         1         2         3
         3         3
)
```

Object *scale-interval1* uses the following pitch specification:

```
(scale-intervals (random-choice sieve1)
                 (series-choice '(-4 -3 -1 1 3 4))
                 sieve1)
```

The allowed values are taken from *sieve1*. The first value is randomly chosen from *sieve1*, ensuring that it is an allowed value. Intervals are chosen with a generator. The choices include 4 and –4. Four is not one of the intervals of the sieve but it is a multiple of 2 which is one of the intervals.

A possible outcome for the pitches of *scaled-interval1* is:

```
(67 64 65 64 61 64 65 64 67 70 67 64 67 64 67 70 69 70 67 70 67 70 67
64 67 64 65 64 67 64 65 69 70 67 70 67 70 69 70 73 77 73 70 67 70 67 70
67 70 69 65 69 73 69 70 69 70 67 70 67 70 73 70 69 70 69 65 69 65 61 65
64 61 64 65 61 65 69 65 61 64 67 64 65 61 57 61 64 61 64 65 61 65 61 58
61 65 61 65 64)
```

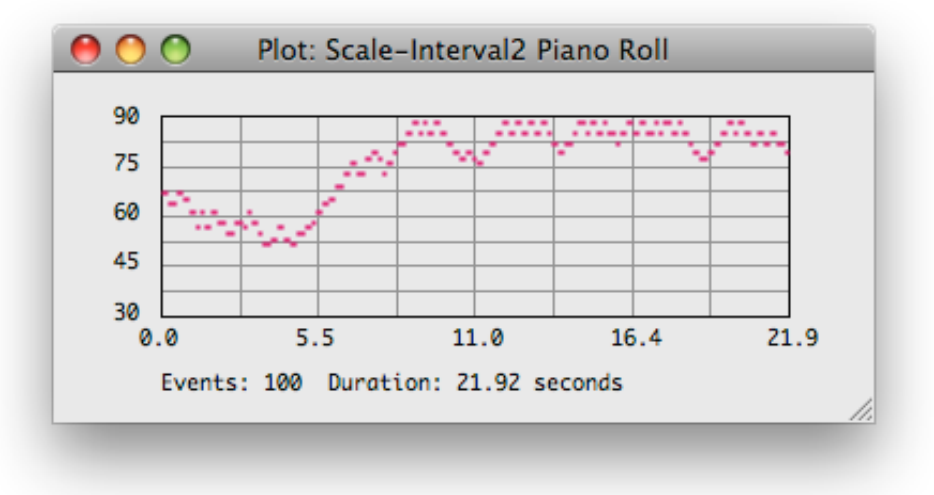
Plus-min is a generator that returns a positive or negative value according to a specified probability. A probability of 1 means that the result will always be negative. 0 means that it will always be positive.

```
10 values from:
(plus-min (rv 1 5) 0.5)
         3          2          3         -2         -3
        -5         -1          5          3          5
```

In the above example, the value is produced with (rv 1 5) and 0.5 is the probability.

Object *scale-interval2* uses *plus-min* to tilt the balance toward positive intervals:

```
(scale-intervals (random-choice sieve1) (plus-min (rv 1 4) 0.3) sieve1)
```



It may be desirable to use only part of an existing stockpile when generating values. The tool *limit* allows this. It allows values in a certain range to be accepted or rejected.

```
List returned from:
(limit sieve1 c3 c4)

(
        49         52         53         55         57         58
)
```

In the above example, the values in *sieve1* between c3 (48) and c4 (60) are returned.

Object *scale-interval3* limits the first value to a specific range of the sieve.

```
(scale-intervals (random-choice (limit sieve1 c3 c4))
                 (plus-min (rv 1 4) 0.4)
                 sieve1)
```

Object *scale-interval4* has a higher probability for negative values:

```
(scale-intervals (random-choice sieve1) (plus-min (rv 1 4) 0.7) sieve1)
```

Object *sieve2* is the union of three sieves:

```
(sieve-union (sieve 9 1 c2 f#6) (sieve 13 1 c2 f#6) (sieve 4 1 c2 f#6))
```

Its values are:

```
37      40      41      45      46
49      53      55      57      61
64      65      66      69      73
77      79      81      82      85
89
```

This sieve is used in *scale-interval5*.

```
(scale-intervals (random-choice sieve2) (plus-min (rv 2 6) 0.5) sieve2)
```

Scale-interval5 uses a larger range of possible intervals than the previous examples.

Selecting a pattern

Although the topic of using a pattern does not directly deal with interval structures, tool *select-patterns* is useful for choosing between different groups of pitches (or rhythms). It will be used in some of the examples dealing with a pitch-matrix later in this tutorial.

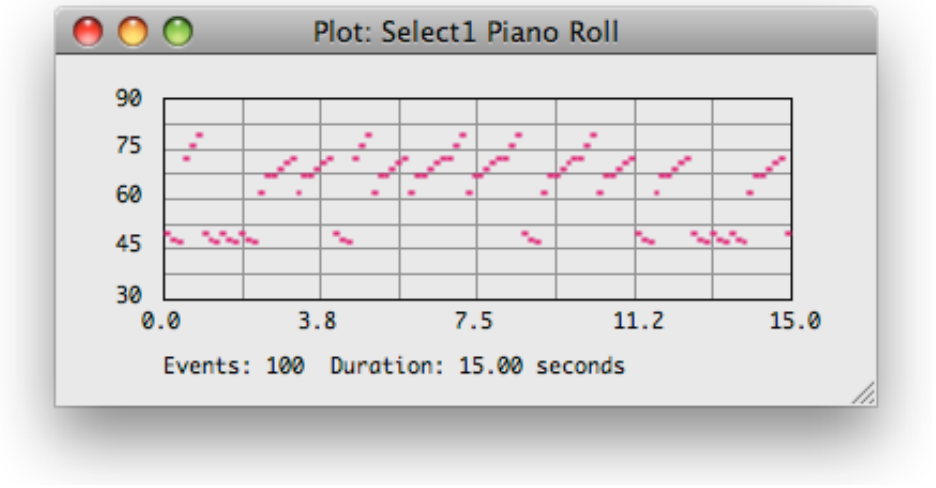
The basic idea is that a generator should pick a list or stockpile. The values in that list are returned one at a time, until the entire list has been returned. At that time, the generator is applied again to choose a new list.

```
20 values from:
(select-patterns (random-choice '((1 2 3) (10 20))))
        10        20         1         2         3
         1         2         3        10        20
         1         2         3         1         2
         3        10        20        10        20
```

In the above example, there are two patterns: (1 2 3) and (10 20). Generator *random-choice* returns one of those patterns. Each element is returned, one at a time. Then a new choice is made.

Object *select1* selects pitches from three available patterns:

```
(select-patterns (random-choice '((72 76 79)
                                  (62 67 67 69 71 72)
                                  (50 48 47))))
```



Select-patterns has an optional second parameter that can be a generator to choose the order of the elements taken from the selected pattern.

```
20 values from:
(select-patterns (random-choice '((1 2 3) (10 20))) #'series-choice)
        20        10        20        10         1
         2         3         1         2         3
        20        10        20        10         3
         2         1        10        20        20
```

Once a pattern has been chosen in the above example, the order of elements is determined by *series-choice*. This means they will be returned in random-order without repetitions. Sometimes the order is 1,2,3. Other times the order may be 3,2,1 or 2,1,3, etc. The generator specified to arrange the elements should expect a list as input (e.g. one of the *choice* generators).

Object *select2* uses *series-choice* to distribute the elements in the chosen pattern.

```
(select-patterns (random-choice '((72 76 79)
                                  (62 67 67 69 71 72)
                                  (50 48 47)))
                 #'series-choice)
```

Transposition matrix

Some approaches to composition are informed by a matrix of transpositions of a series or row. The row could be considered pitch classes or intervals.

Stockpile *row1* is a series of 12 numbers in the range 0-11. The first number is 0 since that is convenient when making a matrix of transpositions.

```
0 11 1 4 2 7 10 5 3 6 8 9
```

Object *row1-section* shifts the pitch classes to be in octave 4. This allows the row to be heard.

This row can be 'inverted', that is subtracted from 12, modulo 12.

```
0 1 11 8 10 5 2 7 9 6 4 3
```

A row could also be generated with tool *make-row*. This returns a list of 12 values in the range 0-11 with the first value being 0.

```
(make-row)
(0 1 7 8 10 9 4 6 2 11 3 5)
```

A traditional approach to making a matrix of transpositions is to use the inversion of a series as the transposition factor. If the first value in the row is 0, the first value in the matrix is the original series.

Stockpile *matrix1* is a transposition matrix based on row1. The transposition factors are from the inversion of row1. *Pitch-matrix* is the tool that will make this matrix.

To see the matrix with one row on a line, enter the name of the matrix in the *For Example* dialog (**Tools>For Example**) and set the option for *per line* to be 1.

```
12 values from:
matrix1
( 0 11 1 4 2 7 10 5 3 6 8 9 )
( 1 0 2 5 3 8 11 6 4 7 9 10 )
( 11 10 0 3 1 6 9 4 2 5 7 8 )
( 8 7 9 0 10 3 6 1 11 2 4 5 )
( 10 9 11 2 0 5 8 3 1 4 6 7 )
( 5 4 6 9 7 0 3 10 8 11 1 2 )
( 2 1 3 6 4 9 0 7 5 8 10 11 )
( 7 6 8 11 9 2 5 0 10 1 3 4 )
( 9 8 10 1 11 4 7 2 0 3 5 6 )
( 6 5 7 10 8 1 4 11 9 0 2 3 )
( 4 3 5 8 6 11 2 9 7 10 0 1 )
( 3 2 4 7 5 10 1 8 6 9 11 0 )
```

Since the transposition factors are taken from the inversion of the stockpile, there are 12 transpositions in this matrix. In the above example, the first value in each transposition comes from the inversion.

The inversion of a row can be calculated with the tool *invert-stockpile*.

```
(invert-stockpile row1)
```

Transposition could also be derived from the row itself.

```
(pitch-matrix row1 'row)

(
( 0 11 1 4 2 7 10 5 3 6 8 9 )
( 11 10 0 3 1 6 9 4 2 5 7 8 )
( 1 0 2 5 3 8 11 6 4 7 9 10 )
( 4 3 5 8 6 11 2 9 7 10 0 1 )
( 2 1 3 6 4 9 0 7 5 8 10 11 )
( 7 6 8 11 9 2 5 0 10 1 3 4 )
( 10 9 11 2 0 5 8 3 1 4 6 7 )
( 5 4 6 9 7 0 3 10 8 11 1 2 )
( 3 2 4 7 5 10 1 8 6 9 11 0 )
( 6 5 7 10 8 1 4 11 9 0 2 3 )
( 8 7 9 0 10 3 6 1 11 2 4 5 )
( 9 8 10 1 11 4 7 2 0 3 5 6 )
)
```

A generator could be used to produce the transposition factor. In that case, the number of transpositions in the matrix will equal the length of the row.

```
(pitch-matrix row1 (series-value 0 11))

(
( 6 5 7 10 8 1 4 11 9 0 2 3 )
( 4 3 5 8 6 11 2 9 7 10 0 1 )
( 11 10 0 3 1 6 9 4 2 5 7 8 )
( 10 9 11 2 0 5 8 3 1 4 6 7 )
( 9 8 10 1 11 4 7 2 0 3 5 6 )
( 0 11 1 4 2 7 10 5 3 6 8 9 )
( 7 6 8 11 9 2 5 0 10 1 3 4 )
( 1 0 2 5 3 8 11 6 4 7 9 10 )
( 5 4 6 9 7 0 3 10 8 11 1 2 )
( 3 2 4 7 5 10 1 8 6 9 11 0 )
( 8 7 9 0 10 3 6 1 11 2 4 5 )
( 2 1 3 6 4 9 0 7 5 8 10 11 )
)
```

The transposition factors could also be a list or stockpile. In that case, the number of transpositions will equal the number of values in the transposition list or stockpile.

```
(pitch-matrix row1 '(0 1 2 3 4))

(
( 0 11 1 4 2 7 10 5 3 6 8 9 )
( 1 0 2 5 3 8 11 6 4 7 9 10 )
( 2 1 3 6 4 9 0 7 5 8 10 11 )
( 3 2 4 7 5 10 1 8 6 9 11 0 )
( 4 3 5 8 6 11 2 9 7 10 0 1 )
)
```

Reading from a matrix

A matrix can be read with *read-permutation.* It will read each row in the matrix in order, and will return the values one at a time.

Object *matrix-section1* shifts the values in the matrix to be in octave 4. If you play this section, you can hear the matrix in order. The pitch specification is:

```
(pitch-and-octave (read-permutation matrix1) 4)
```

Object *matrix-section2* spreads the matrix over three octaves:

```
(pitch-and-octave (read-permutation matrix1) (rv 3 5))
```

If the keyword *random* is bound to *t* in *read-permutation,* the transpositions are read in random order. This means that a transposition is chosen at random, and then each element in that transposition is returned, one at a time. When all elements in a transposition have been returned, a new transposition is chosen.

Object *matrix-section3* reads the matrix in random order:

```
(pitch-and-octave (read-permutation matrix1 :random t) (rv 3 5))
```

Chords can be made with the values read from the matrix in combination with the generator *make-chord*. This happens in object *matrix-section4*. The specification is:

```
(make-chord (pitch-and-octave (read-permutation matrix1 :random t)
                              (rv 3 5))
            (rv 3 5))
```

Another way to read the matrix is using *read-from*. With this generator you can read from a stockpile in a number of ways. You could, for example, give a list of index values (the index for the first value is 1):

```
12 values from:
(read-from '(a b c d e f) '(5 1 2 3 2 1))
         e         a         b         c         b         a
         e         a         b         c         b         a
```

A generator could be used to produce the index values (in the range 1 to the length of the stockpile):

```
12 values from:
(read-from '(a b c d e f) (series-value 1 6))
         e         a         f         b         c         d
         c         a         e         b         d         f
```

When *read-from* is used to read from *matrix1*, each value returned is a transposed row (as a list):

```
5 values from:
(read-from matrix1 '(5 1 2 4 3))
( 10 9 11 2 0 5 8 3 1 4 6 7 )
( 0 11 1 4 2 7 10 5 3 6 8 9 )
( 1 0 2 5 3 8 11 6 4 7 9 10 )
( 8 7 9 0 10 3 6 1 11 2 4 5 )
( 11 10 0 3 1 6 9 4 2 5 7 8 )
```

To read through each element in one of these transpositions when it is returned, *select-patterns* should be used. It returns each element in a selected pattern before choosing a new pattern.

```
24 values from:
(select-patterns (read-from matrix1 '(5 1)))
        10         9        11         2         0         5
         8         3         1         4         6         7
         0        11         1         4         2         7
        10         5         3         6         8         9
```

Object *matrix-section5* uses the combination of *select-patterns* and *read-from* to produce pitch values derived from *matrix1*. The pitch specification is:

```
(pitch-and-octave (select-patterns
                     (read-from matrix1
                                (beta-value 1 12 0.3 0.3)))
                  (rv 3 5))
```

The generator *beta-value* is used to pick the number of the transposition. *Beta-value* tends to pick values close to one of its boundaries, in this case 1 or 12.

Manipulating a matrix

Invert-stockpile produces the inversion of the row. This was seen above. If the input is a matrix, each list in the matrix is inverted:

```
(invert-stockpile matrix1)

(
( 0 1 11 8 10 5 2 7 9 6 4 3 )
( 11 0 10 7 9 4 1 6 8 5 3 2 )
( 1 2 0 9 11 6 3 8 10 7 5 4 )
( 4 5 3 0 2 9 6 11 1 10 8 7 )
( 2 3 1 10 0 7 4 9 11 8 6 5 )
( 7 8 6 3 5 0 9 2 4 1 11 10 )
( 10 11 9 6 8 3 0 5 7 4 2 1 )
( 5 6 4 1 3 10 7 0 2 11 9 8 )
( 3 4 2 11 1 8 5 10 0 9 7 6 )
( 6 7 5 2 4 11 8 1 3 0 10 9 )
( 8 9 7 4 6 1 10 3 5 2 0 11 )
( 9 10 8 5 7 2 11 4 6 3 1 0 )
)
```

Retrograde-stockpile produces the retrograde version of the row.:

```
(retrograde-stockpile row1)
(9 8 6 3 5 10 7 2 4 1 11 0)
```

If the input is a matrix, each list in the matrix is reversed:

```
(retrograde-stockpile matrix1)

(
( 9 8 6 3 5 10 7 2 4 1 11 0 )
( 10 9 7 4 6 11 8 3 5 2 0 1 )
( 8 7 5 2 4 9 6 1 3 0 10 11 )
( 5 4 2 11 1 6 3 10 0 9 7 8 )
( 7 6 4 1 3 8 5 0 2 11 9 10 )
( 2 1 11 8 10 3 0 7 9 6 4 5 )
( 11 10 8 5 7 0 9 4 6 3 1 2 )
( 4 3 1 10 0 5 2 9 11 8 6 7 )
( 6 5 3 0 2 7 4 11 1 10 8 9 )
( 3 2 0 9 11 4 1 8 10 7 5 6 )
( 1 0 10 7 9 2 11 6 8 5 3 4 )
( 0 11 9 6 8 1 10 5 7 4 2 3 )
)
```

A retrograde inversion of the matrix can be made by combining the tools *invert-stockpile* and *retrograde-stockpile*:

```
(retrograde-stockpile (invert-stockpile matrix1))

(
( 3 4 6 9 7 2 5 10 8 11 1 0 )
( 2 3 5 8 6 1 4 9 7 10 0 11 )
( 4 5 7 10 8 3 6 11 9 0 2 1 )
( 7 8 10 1 11 6 9 2 0 3 5 4 )
( 5 6 8 11 9 4 7 0 10 1 3 2 )
( 10 11 1 4 2 9 0 5 3 6 8 7 )
( 1 2 4 7 5 0 3 8 6 9 11 10 )
( 8 9 11 2 0 7 10 3 1 4 6 5 )
( 6 7 9 0 10 5 8 1 11 2 4 3 )
( 9 10 0 3 1 8 11 4 2 5 7 6 )
( 11 0 2 5 3 10 1 6 4 7 9 8 )
( 0 1 3 6 4 11 2 7 5 8 10 9 )
```

Interpreting a matrix as chords

A pitch matrix contains a list of lists. Chords in the AC Toolbox are represented as lists. A matrix could be interpreted as a list of chords. It is a question of how you want to consider the data.

Considering each transposition of the matrix as a chord means the entire matrix would be a set of chord transpositions.

Stockpile *matrix2* is a matrix based on the list '(60 61 71 68) that could be considered a chord. *Pitch-matrix* reduces the values to pitch classes before constructing the transposition matrix.

```
4 values from:
matrix2
( 0 1 11 8 )
( 11 0 10 7 )
( 1 2 0 9 )
( 4 5 3 0 )
```

Since there are only four numbers in the input, there are only four transpositions.

Object *matrix-section6* treats the lists in the matrix as chords. *Read-from* will return a chord each time it is called. *Select-patterns* is not used because the list is being used as a chord rather than as a succession of separate pitches. The pitch specification is:

```
(pitch-and-octave (read-from matrix2 (rv 1 (get-length matrix2)))
                  (rv 3 5))
```

Transposing lists of chords

Pitch-matrix can also take a list of lists as input. Each element of this list is treated as a chord that will be reduced to pitch classes before being transposed.

Stockpile *chords1* is a stockpile of 3-pitch chords, generated with series-value:

```
(make-chord (series-value 0 11) 3)
```

A possible list of chords generated for this stockpile is:

```
((3 11 10) (0 2 4) (7 1 8) (9 5 6))
```

Object *chord-section1* can be used to play through the four generated chords.

Stockpile *matrix3* is a transposition matrix based on this list of chords. The entire list of chords is transposed. When transposing a list of chords, *row* and *inversion* are not available as transposition options. A list or a generator should be provided for the transposition factors. The input for *matrix3* is:

```
(pitch-matrix chords1 (from 0 11))
```

There will be 12 transpositions since (*from 0 11)* produces a list with 12 values.

```
12 values from:
matrix3
( (3 11 10) (0 2 4) (7 1 8) (9 5 6) )
( (4 0 11) (1 3 5) (8 2 9) (10 6 7) )
( (5 1 0) (2 4 6) (9 3 10) (11 7 8) )
( (6 2 1) (3 5 7) (10 4 11) (0 8 9) )
( (7 3 2) (4 6 8) (11 5 0) (1 9 10) )
( (8 4 3) (5 7 9) (0 6 1) (2 10 11) )
( (9 5 4) (6 8 10) (1 7 2) (3 11 0) )
( (10 6 5) (7 9 11) (2 8 3) (4 0 1) )
( (11 7 6) (8 10 0) (3 9 4) (5 1 2) )
( (0 8 7) (9 11 1) (4 10 5) (6 2 3) )
( (1 9 8) (10 0 2) (5 11 6) (7 3 4) )
( (2 10 9) (11 1 3) (6 0 7) (8 4 5) )
```

Object *matrix-section7* uses *read-from* to read a transposition list from the matrix. *Select-patterns* will read through that list of chords using *series-choice* to determine the order of the chords. *Pitch-and-octave* transposes the entire chord. The pitch specification is:

```
(pitch-and-octave (select-patterns
                     (read-from matrix3
                                (rv 1
                                    (get-length matrix3)))
                    #'series-choice)
                  (rv 2 6))
```

Summary

Generators
random-intervals, allow-interval, block-interval, make-chord, scale-intervals, plus-min, select-patterns, read-permutation, read-from

Tools
convert, generate-range, sieve, sieve-union, sv, su, get-intervals, pitch-matrix, invert-stockpile, retrograde-stockpile, make-row

Transformers
funnel

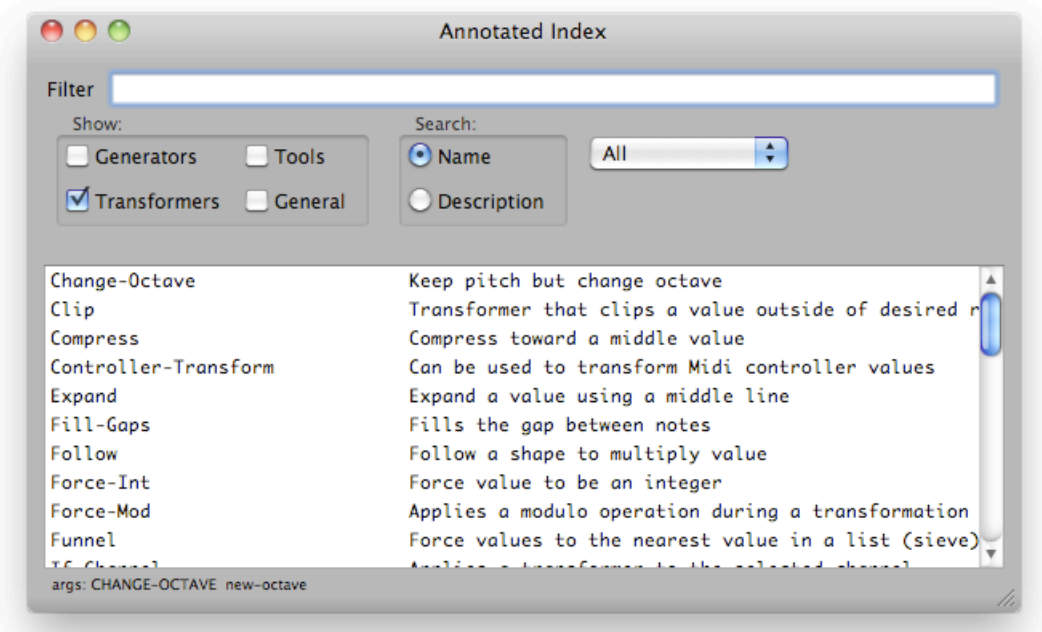
Tutorial 25

Transforming objects

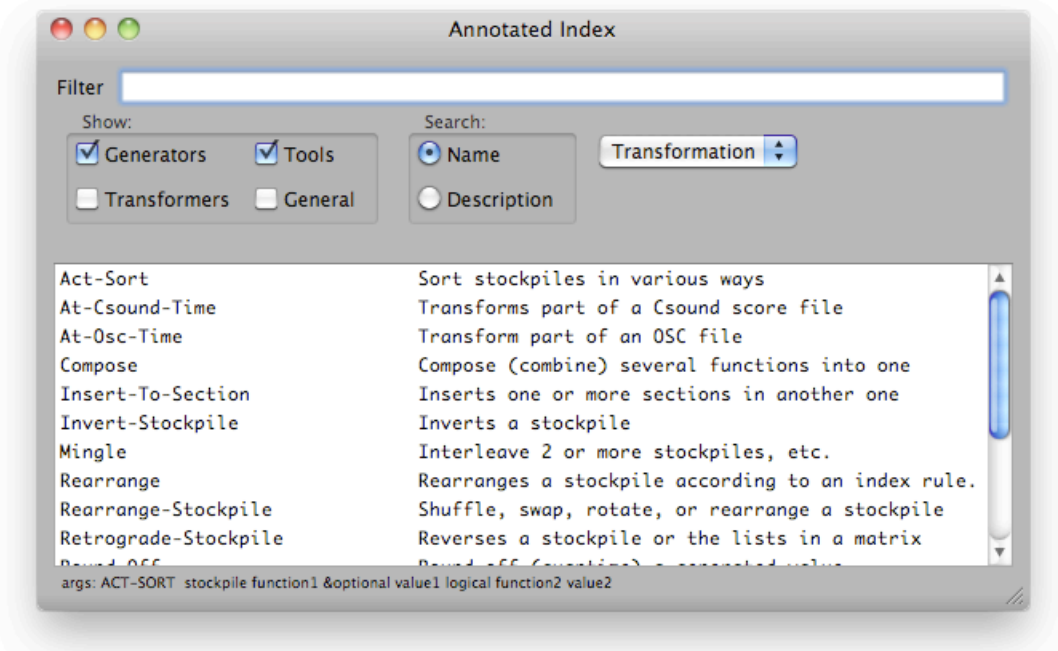
A few possibilities to transform objects were discussed in Tutorial 7. The transform method (**Help>Transform>Object**) allows different types of transformers to be applied to various objects. Sections, shapes, stockpiles, etc. can be transformed. Some objects, such as streams, cannot be transformed.

Transformers offer a means to interact with data that has already been made.

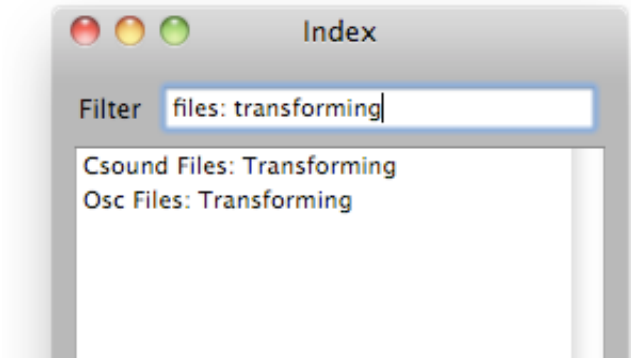
A number of transformers are available in the AC Toolbox. They are functions that change a value in some way. Simple transformers include *transpose, stretch,* and *mirror.* A list of all available transformers can be found in the *Annotated Index*. Transformers can be applied, for example, to all pitches in a section. It is possible with conditional transformers such as *transform-if*, to only transform pitches with short durations or which occur in a certain time frame of a section.



Tools and generators can also transform data. Several of these are listed in the Annotated Index in the category *Transformation*. Values in stockpiles can be sorted, rearranged, or shuffled. On a larger scale, sections can be inserted into other sections (as an interruption).



In addition, Csound score files and binary OSC files generated by the AC Toolbox can be transformed with the menu items **Methods>Transform>Csound File** and **Methods>Transform>OSC File**. Transformations on these files can be limited to a particular time frame with *at-csound-time* and *at-osc-time*. Transformation of Csound and OSC files are not discussed in this tutorial. See the general help items in the application for more information.



The AC Toolbox objects discussed in this tutorial can be found in the file *Tutorial 25 Examples*. Load these objects by selecting **Load Examples** in the **File** menu. A table listing the object definitions appears. To see the object definition, click on the name of the object in the table. To make the object, click on *Make* in the dialog box.

Simple transformers

Transformers can be applied to various types of objects. In the following examples, they will primarily be applied to sections or stockpiles.

Translate
Translate is a transformer that will map a value in one range, say pitches 40 to 80, to be in the range 60 to 64.

The specification for *translate* is:

```
(TRANSLATE A B LOW HIGH &KEY CLIP CHORD ROUND)
```

A and *B* specify the existing range. *Low* and *high* are the target range. If *low* and *high* are integers, the result is an integer. Otherwise the result is a real number. *Round* can be used to quantize the result to the unit bound to that key word.

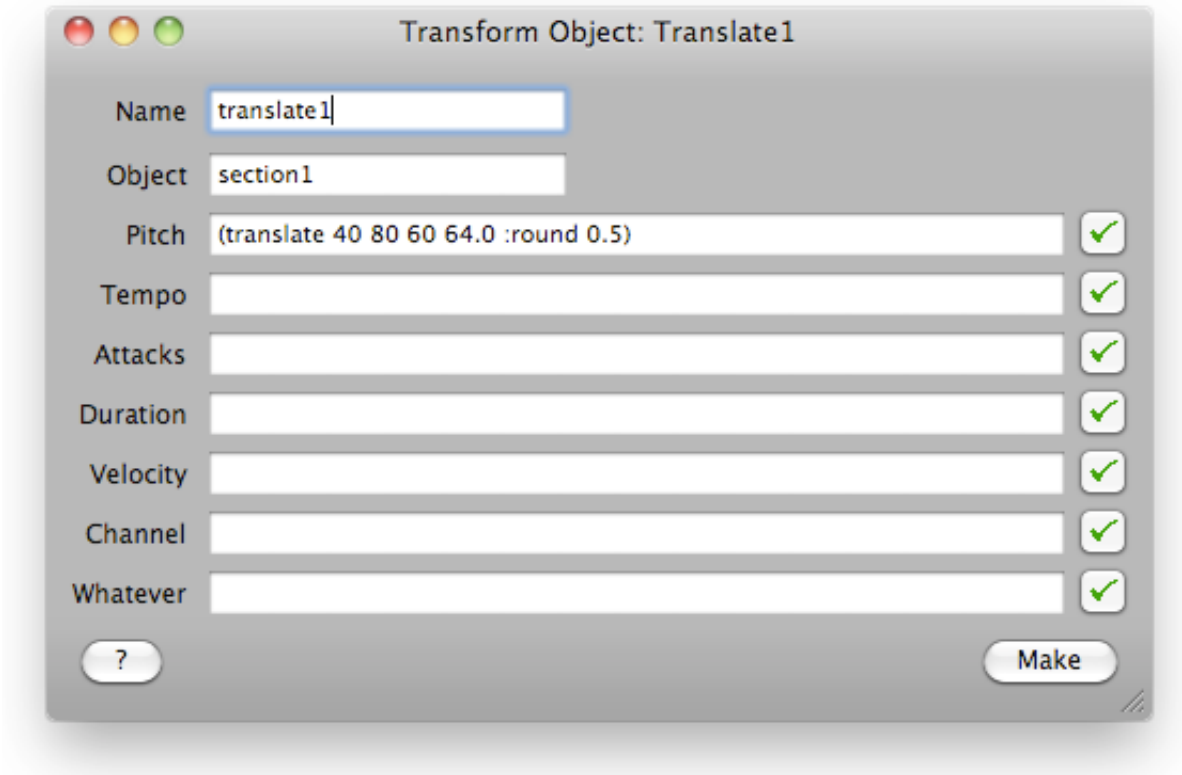
```
Applying transformer
(translate 40 80 60 64.0 :round 0.5)
gives the following results:

Value:    64    Transformation: 62.500
Value:    78    Transformation: 64.000
Value:    78    Transformation: 64.000
Value:    64    Transformation: 62.500
Value:    63    Transformation: 62.500
Value:    79    Transformation: 64.000
Value:    59    Transformation: 62.000
Value:    75    Transformation: 63.500
Value:    48    Transformation: 61.000
Value:    65    Transformation: 62.500
```

The values on the left of the above list are in the range 40 to 80. They are mapped into values in the range 60 to 64.0 and then quantized in units of 0.5. If these values are pitches, the unit is a quarter-tone. The above listing is made with the tool *show-transformation* which is used in help windows for transformers.

Object *section1* is defined so it can be transformed.

```
Name            section1
Clock unit      100
Number          30
Rhythm          (random-value 1.0 3)
Pitch           (series-value c3 g#5)
Velocity        (random-choice '(pp p mf f))
Channel         1
```

Object *translate1* will map the pitches of *section1*. This maintains the pitch contour but compresses its range. The pitches are rounded to quarter-tones.

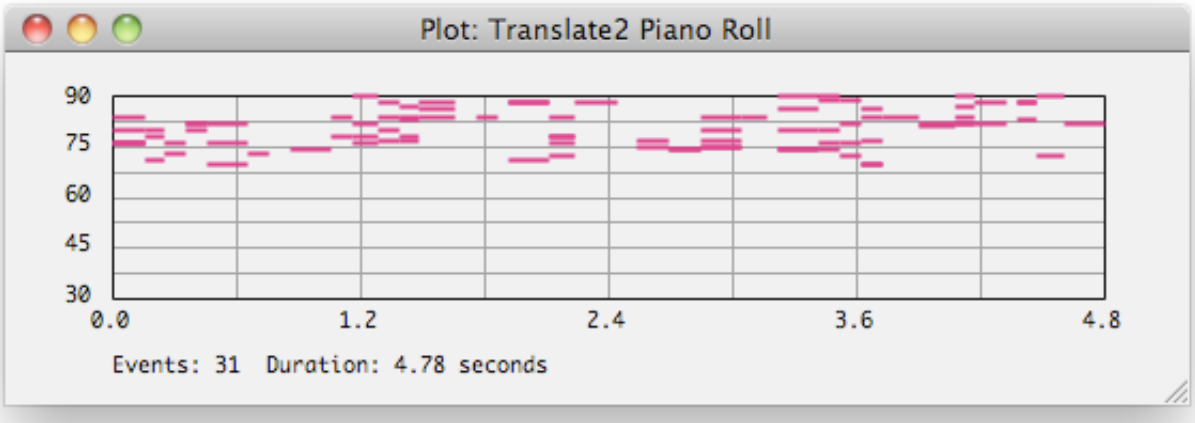


Section2 is an object with chords. A negative value for rhythm denotes a rest.

```
Name            section2
Clock unit      100
Number          40
Rhythm          (rc '(-0.5 1 1.25 1.5 1.75 2))
Pitch           (make-chord (rv 40 80) (rv 1 5))
Velocity        mf
Channel         1
```

Translate2 maps the chords of *section2* to be in the range of 70 to 90. The results are integer values.

```
Name            translate2
Object          section2
Pitch           (translate 40 80 70 90)
```



It is necessary to know the original range of a parameter or object before it can be translated. Usually the values can be derived from the input specification. Otherwise, *find-range* can be used in the For Example dialog or in the Listener. If it is applied to a section, the default parameter is pitch. It can also be used to find the range of duration or velocity values in a section.

```
(find-range section1)
("Minimum: 48.00 " "Maximum: 78.00 ")

(find-range section1 :parameter 'duration)
("Minimum: 104.00 " "Maximum: 297.00 ")
```

Limit-range

If *translate* is applied to pitch, it modifies the pitch relationships. *Limit-range* can be used to shift pitches to a different range but maintain the pitch class.

The specification for *limit-range* is:

```
(LIMIT-RANGE LOW HIGH &OPTIONAL MODULO)
```

Modulo is added or subtracted from the current value until it is in the range *low to high*. If *modulo* is 12, which is the default value, values will be moved up or down an octave until they fit in the desired range. If the desired range is too narrow to allow the change, an error message occurs.

```
Applying transformer
(limit-range 60 72)
gives the following results:

Value:   40    Transformation:   64
Value:   45    Transformation:   69
Value:   50    Transformation:   62
Value:   55    Transformation:   67
```

```
Value:   60    Transformation:   60
Value:   65    Transformation:   65
Value:   70    Transformation:   70
Value:   75    Transformation:   63
Value:   80    Transformation:   68
Value:   85    Transformation:   61
Value:   90    Transformation:   66
```

Limit1 transforms *section1* by adjusting the octaves so that the pitch classes are placed in the octave from c4 to c5.

```
Name           limit1
Object         section1
Pitch          (limit-range c4 c5)
```

This does not necessarily maintain the pitch contour but it does maintain the pitch classes.

Quantize

It may be desirable to quantize rhythmic or other values to some unit. This can be done with transformer *quantize.* Its specification is:

```
(QUANTIZE UNIT &OPTIONAL TRANSFORMER)

Applying transformer
(quantize 50)
gives the following results:

Value:  100    Transformation:  100
Value:  120    Transformation:  100
Value:  160    Transformation:  150
Value:  180    Transformation:  200
Value:  210    Transformation:  200
```

When quantize is applied to tempo, attack, or duration values, the units should be expressed in milliseconds. The values represent the clock unit multiplied by the rhythm value. The duration values in milliseconds for a section can be seen in the Edit window (select Edit in the Objects dialog or in the popup menu made when the right mouse button is selected above the Make button of the object).

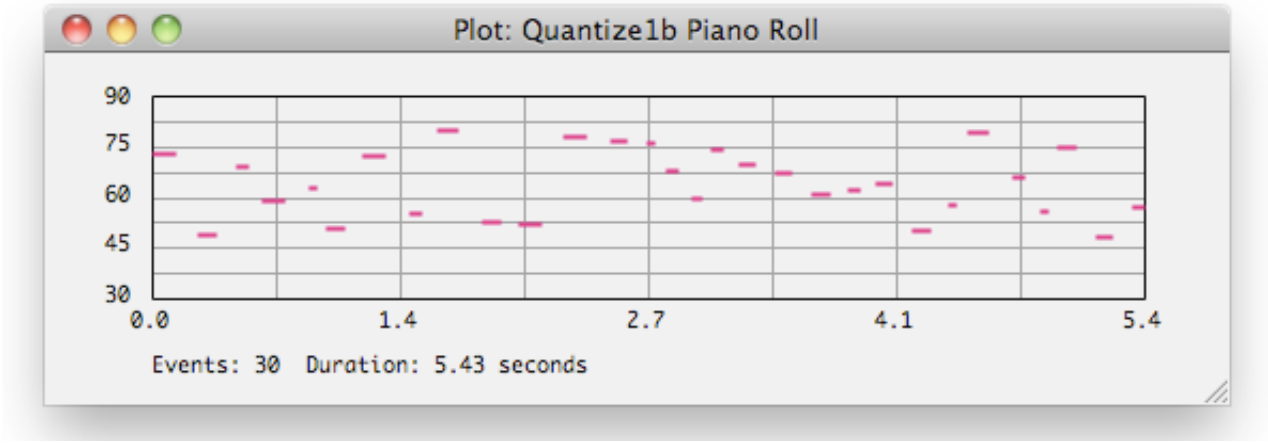
Object *quantize1* quantizes the tempo of *section1* to units of 50 milliseconds.

```
Name           quantize1
Object         section1
Tempo          (quantize 50)
```

Tempo adjusts both the duration and attack times of the object. These two parameters can also be transformed separately.

Object *quantize1b* multiplies the duration values of section *quantize1* by 0.5. The attack times are not changed.

```
Name           quantize1b
Object         section1
Duration       (stretch 0.5)
```



Quantize has an optional parameter for another transformer which is applied to the value before it is quantized.

```
Applying transformer
(quantize 0.5 (transpose (rv -1.0 1)))
gives the following results:

Value:   60    Transformation: 61.000
Value:   64    Transformation: 63.500
Value:   67    Transformation: 67.000
```

Object *quantize2* transposes pitch by a random factor between –3.0 and 3 (resulting in a floating-point pitch value) and then rounds it off to units of a quarter-tone (0.5).

```
Name            quantize2
Object          section1
Pitch           (quantize 0.5 (transpose (rv -3.0 3)))
```

Insert-rest

Insert-rest can be used to inserts rests into a section. It must be used as the w*hatever* parameter in the Transform dialog. It will insert a rest at the specified start times. The start times of notes occurring after the rest are adjusted accordingly. The specification is:

```
(INSERT-REST START-TIMES LENGTH-OF-REST &KEY (MARGIN 0))
```

Start-times can be a list, stockpile or number. Its values need to be start times that actually occur in the section. One way to see the start times that occur in the section is to look at the Edit window. If a start time for inserting a rest does not occur in the section, no rest is inserted.

Section3 will be used for the transformations.

```
Name            section3
Clock unit      100
Number          (until-time 30)
Rhythm          (rc '(-0.5 1 1.25 1.5 1.75 2))
Pitch           (make-chord (series-value 70 90) (rv 1 5))
Velocity        (rc '(p mf f ff))
Channel         1
```

A list of the start times in section3 can be retrieved via the For Example dialog by evaluating this expression:

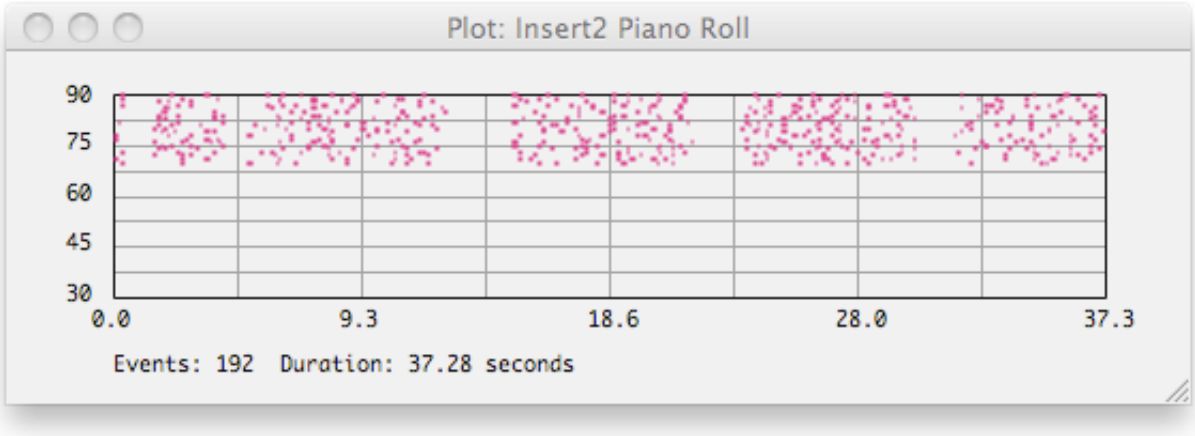
```
(extract 'raw-attack section3)
```

Object *insert1* will have rests inserted at moments chosen by hand from that list. The length of the rest is expressed in milliseconds. After 3175 ms, a rest of length 900 ms is inserted, etc. These start-times might not exist in the *section3* you made. Replace the start times in the list with start times you find in *section3*.

```
Name            insert1
Object          section3
Whatever        (insert-rest '(3175 17875 26075) '(900 1500 2500))
```

The insertion points can be generated from the list of available start times by choosing from the extracted list of attack points. In *insert2*, *produce* makes a list containing five attacks chosen with *series-choice* from the list of all available points. The length of the rest is chosen from a list of five user-specified lengths.

```
Name            insert2
Object          section3
Whatever        (insert-rest
                  (produce 5 (series-choice
                               (extract 'raw-attack section3)))
                  (series-choice '(800 1000 1400 1700 2400)))
```



A generator is not a suitable input for determining start times. The transformer wants to arrange the start times in ascending order before beginning. That is the reason that a list was produced in the previous example for the value for start-times.

A more adventuresome approach to determining start times can come from just generating a list of values and seeing if any of them actually occur in the section. If a note occurs at time X, a rest is added before that note. A margin in milliseconds can be specified to increase the chance of success. If an event starts in the range time X to time X + MARGIN, a rest is inserted.

```
Name            insert3
Object          section3
Whatever        (insert-rest
                  (produce 5 (series-value 1000 25000))
                  (series-value 1000 2000)
                  :margin 1000)
```

Transform-by-index

Transformations can be limited to certain events. One way to do this is by indicating the index number for each event to be transformed. Other possibilities are discussed in the section *Conditional transformers* later in this tutorial.

```
(TRANSFORM-BY-INDEX TO-PRODUCE-INDEX TRANSFORMER)
```

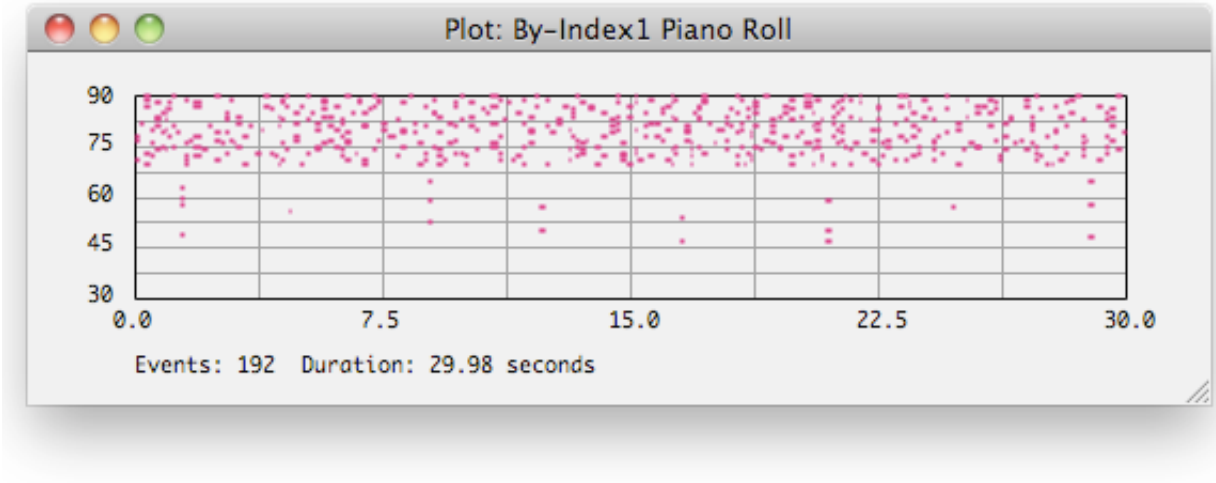
Index numbers start with 1. In an Edit window for a section, index values are printed in the leftmost column.

If a generator is used to produce the index values, the generator should produce ascending values. *Walk* can do this. Once the last index of the object has been reached, no more values are produced.

```
10 values from:
(walk (rv 5 20) (rv 20 30))
         5         29         50         70         99
       122        148        169        195        216
```

The first value was chosen in the range 5-20. A new value in the range 20-30 was added to it. A similar generator is used to choose index values in section3. At the chosen events, the pitch is transposed down two octaves.

```
Name            by-index1
Object          section3
Pitch           (transform-by-index
                     (walk (rv 5 20) (rv 20 30))
                     (transpose -24))
```



If the index values are a list or stockpile, the transformer will sort them into ascending order. The length of the object being transformed can be accessed with tool *get-length*. In *by-index2*, the number of transformations is chosen with generator *(rv 4 8).*

```
Name            by-index2
Object          section3
Pitch           (transform-by-index
                     (produce (rv 4 8)
                              (series-value 5 (get-length section3)))
                     (transpose -24))
```

In *by-index2*, 4 to 8 events will be transformed. The events will be chosen from event 5 to the last event in the section.

If two parameters in a section should be transformed at the same time, the index values can be stored in a stockpile. That stockpile can be used for both transformers.

Stockpile *index-stockpile* contains between 4 and 8 indices. It is used indicate which events should be transformed in section 3. For the chosen events, pitch is transposed down by two octaves and tempo is stretched by 10.

```
Name            by-index3
Object          section3
Pitch           (transform-by-index index-stockpile (transpose -24))
Tempo           (transform-by-index index-stockpile (stretch 10))
```

Transform-by-index can also be used for transforming other objects since each event in an object has an index. OSC files produced by the AC Toolbox can be transformed with it. Csound score files do not work with this transformer.

Transform-stockpile
This tool is for transforming stockpiles or lists. Any available transformer can be applied to a stockpile or a list. The result is a list with transformed values. The specification is:

```
(TRANSFORM-STOCKPILE STOCKPILE TRANSFORMER)
```

Transform-stockpile can be used when defining another object. It can also be evaluated in the Listener or the For Example dialog to examine results.

Row1 is a stockpile that will be transformed. It contains a twelve-tone row.

```
(make-row)
```

A possible result could be:

(0 11 10 2 1 8 7 9 3 5 4 6)

This stockpile can be transposed:

```
(transform-stockpile row1 (transpose 60))
(60 71 70 62 61 68 67 69 63 65 64 66)
```

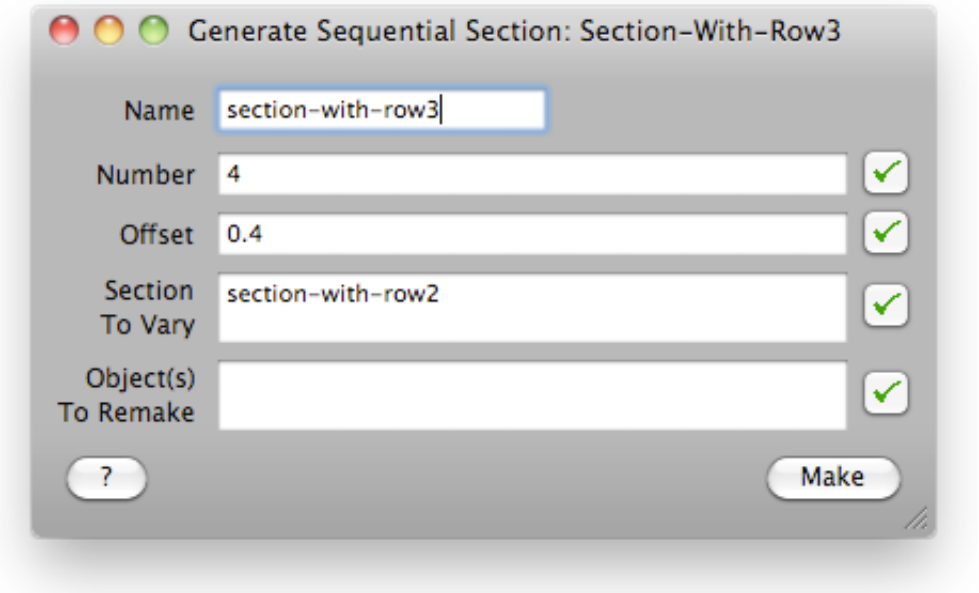
Section-with-row1 uses transform-stockpile to transpose the row to make pitch values. *Series-choice* will choose from the list of transposed values.

```
Name            section-with-row1
Clock unit      200
Number          12
Rhythm          1
Pitch           (series-choice
                 (transform-stockpile row1 (transpose 60)))
Velocity        mf
Channel         1
```

Section-with-row2 picks one transposition factor when the section is made. *Make* applies a generator once to return a value.

```
Name            section-with-row2
Clock unit      200
Number          12
Rhythm          1
Pitch           (series-choice
                  (transform-stockpile row1
                     (transpose (make (rv 40 80)))))
Velocity        mf
Channel         1
```

Section-with-row3 is a sequential section with 4 variants of *section-with-row2*. *Section-with-row2* picks a new transposition factor each time it is made. The four variants in *section-with-row3* therefore each have a newly selected transposition factor.



Tran

Some transformers just perform a simple arithmetic calculation. *Transpose* adds, *stretch* multiplies. These operations can be directly specified with transformer *tran*.

```
(transform-stockpile '(60 64 67) (tran + 12))
(72 76 79)
```

This transformer allows the transformation to be the result of one or more operations that are carried out left to right.

```
(transform-stockpile '(1 2 3) (tran * 2 + 1))
(3 5 7)
```

The operations can have numbers, lists, stockpiles, generators, etc. as arguments. With arguments that are not constant values, the next value (from the list, stockpile, or generator) is used.

```
(transform-stockpile '(1 2 3 4 5) (tran + '(2 3 5 7)))
(3 5 8 11 7)

(transform-stockpile '(1 2 3 4 5) (tran + (rv -1.0 1)))
(1.8694096 2.5629106 3.0419844 4.104805 5.6486764)
```

To quantize values, operator *roundq* can be used. The argument is the quantization unit.

```
(transform-stockpile '(60 64 67) (tran + (rv -1.0 1) roundq 0.5))
(60.5 63.5 68.0)
```

In the above example, a random value in the range -1 to 1 is added to each value in the list. The result is quantized to units of 0.5.

Tran is similar to generator *on-the-fly* except the initial value is the value to be transformed. *Tran* can be used anywhere that a transformer is allowed. Section *tran1* is *section1* with the pitch transposed down an octave and perhaps adjusted up or down a quarter-tone.

```
Name          tran1
Object        section1
Pitch         (tran - 12 + (rc '(-0.5 0 0.5)))
```

Conditional transformers

There are three transformers available that allow specifying specific conditions when a transformer can be applied. These transformers can express things such as:

- stretch the tempo by 2 if the pitch is > 60;
- if the channel is 2 and the start-time is between 5000 and 10000 milliseconds, replace the velocity;
- if the pitch is high or the duration is short, increase the attack time.

The three transformers are *transform-if, transform-and,* and *transform-or*. They only work with sections and should be used with the *whatever* parameter. The whatever parameter allows the transformer access to all of the necessary values.

For each event in a section, the transformer will decide if the transformation should be made or not.

transform-if

The specification for transform-if is:

```
(TRANSFORM-IF PARAMETER CONDITION TPARAMETER TRANSFORMER)
```

Transform-if can test one condition. If that condition is true, the transformer is applied.

Parameter is the parameter being tested. It can be:
pitch, velocity, channel, duration, or start-time.

Conditions can be made with the tool *anything*. A few examples:

`(anything < 80)` is true if the parameter value being tested in < 80;
`(anything outside 60 72)` is true if the value is outside of the range 60-72;
`(anything inside 60 72)` is true if the value is inside the range 60-72;

Tparameter is the parameter being transformed. It can be:
pitch, velocity, channel, tempo, duration, or attack.

Section4 will be used to test these transformers. It consists of random walks for *rhythm* and *pitch*.

```
Name            section4
Clock unit      50
Number          200
Rhythm          (walk 2.5 (rv -0.5 0.5) 0.5 5)
Pitch           (walk 60 (rv -5 5) 20 100)
Velocity        mf
Channel         1
```

Section *if1* sets the velocity to be *ff* if the pitch is lower than 50. Despite its name, *replace-all* just replaces one value with *ff*.

```
Name            if1
Object          section4
Whatever        (transform-if 'pitch (anything < 50)
                              'velocity (replace-all ff))
```

Section *if2* transposes pitch up two octaves if the duration if > 200 ms.

```
Name            if2
Object          section4
Whatever        (transform-if 'duration (anything > 200)
                              'pitch (tran + 24))
```

Section *if3* stretches the tempo 5 times for the events that occur between seconds 10-20 in the original section. Start times are expressed in milliseconds.

```
Name            if3
Object          section4
Whatever        (transform-if
                   'start-time (anything inside 10000 20000)
                   'tempo (stretch 5))
```

Note that using *start-time* as the parameter for the condition allows time-dependent transformations to be made. The transformer is only applied in the specified time frame.

transform-and

The specification for *transform-and* is:

```
(TRANSFORM-AND PARAMETERS-AND-CONDITIONS TPARAMETER TRANSFORMER)
```

It only makes a transformation if all of the conditions in the list *parameters-and-conditions* are met. The parameters that can be tested are the same as for *transform-if*.

The possible values for *tparameter* are the same as in *transform-if.*

Parameters-and-conditions should be specified as a list, using the Lisp function *list*. For example, to test if pitch is < 60 and the channel is 1, the following expression could be used:

```
(list 'pitch (anything < 60) 'channel (anything = 1))
```

There can be any number of parameters and conditions. For each parameter, there must be a condition.

In section *and1*, if the pitch is > 60 during the first 15 seconds, the tempo will be stretched 5 times.

```
Name            and1
Object          section4
Whatever        (transform-and
                  (list 'start-time (anything < 15000)
                        'pitch  (anything > 60))
                  'tempo (stretch 5))
```

In section *and2*, if the pitch is < 60 between time 10 and 20 seconds, the note duration (but not the attack time) is multiplied by 0.2.

```
Name            and2
Object          section4
Whatever        (transform-and
                  (list 'start-time (anything inside 10000 20000)
                        'pitch  (anything < 60))
                  'duration (tran * 0.2))
```

Important in regard to *transform-and* is that all of the conditions must be met before a transformer is applied. There is one list of parameters and conditions.

transform-or
The specification for *transform-or* is:

```
(TRANSFORM-OR PARAMETERS-AND-CONDITIONS TPARAMETER TRANSFORMER)
```

It makes a transformation if one or more of the conditions is true. *Parameters-and-conditions* are in a list made with *list*. The possible parameters are the same as with *transform-if* and *transform-and*. The possible *tparameters* are the same as with those two transformers as well.

Any number of conditions can be specified in the list *parameters-and-conditions*. Only one of them needs to be true for the transformer to be applied.

In section *or1*, if pitch is > 67 or duration is < 120 milliseconds, the velocity is set to *ff*.

```
Name            or1
Object          section4
Whatever        (transform-or
                  (list 'pitch (anything > 67)
                        'duration (anything < 120))
                  'velocity (replace-all ff))
```

In section *or2*, if pitch is in the range 60-72 or duration is outside the range 120-175 ms, the pitch is transposed two octaves.

```
Name            or2
Object          section4
Whatever        (transform-or
                  (list 'pitch (anything inside 60 72)
                        'duration (anything outside 120 175))
                  'pitch (transpose 24))
```

Tools to change order

A transformation can be about the order in which elements occur. This part of the tutorial will look at a few tools and generators that do that. In particular, the focus will be on stockpiles and lists.

Rearrange-stockpile
The tool *rearrange-stockpile* provides a number of ways to return a list of values from a stockpile. The original stockpile is not changed.

```
(REARRANGE-STOCKPILE STOCKPILE TYPE &OPTIONAL A B)
```

Type can be shuffle, swap, rotate-left, rotate-right, adjacent, or order.

A possible list of values for *row1* is `(0 11 10 2 1 8 7 9 3 5 4 6)`.

Shuffle rearranges the stockpile in a random order.

```
(rearrange-stockpile row1 'shuffle)
(7 9 10 1 5 0 8 3 2 6 4 11)
```

Swap exchanges values at two locations in the stockpile. *A* and *B* would be the locations to swap. The numbering for locations starts with 1.

```
(rearrange-stockpile row1 'swap 1 5)
(1 11 10 2 0 8 7 9 3 5 4 6)
```

Adjacent swaps the element at location *A* with the element one place to the right.

```
(rearrange-stockpile row1 'adjacent 1)
(11 0 10 2 1 8 7 9 3 5 4 6)
```

Order rearranges the stockpile in the specified order. The first element is in position 1. The order is bound to A and can be a list, stockpile, generator, etc.

```
(rearrange-stockpile row1 'order (from 12 1))
(6 4 5 3 9 7 8 1 2 10 11 0)

(rearrange-stockpile row1 'order (series-value 1 12))
(10 4 5 1 11 7 8 9 0 6 3 2)
```

Section *rearrange-stockpile1* shuffles stockpile *row1* and adds 60 to the values. This is used for pitch.

```
Name            rearrange-stockpile1
Clock unit      200
Number          36
Rhythm          1
Pitch           (cal 60 + (rearrange-stockpile row1 'shuffle))
Velocity        mf
Channel         1
```

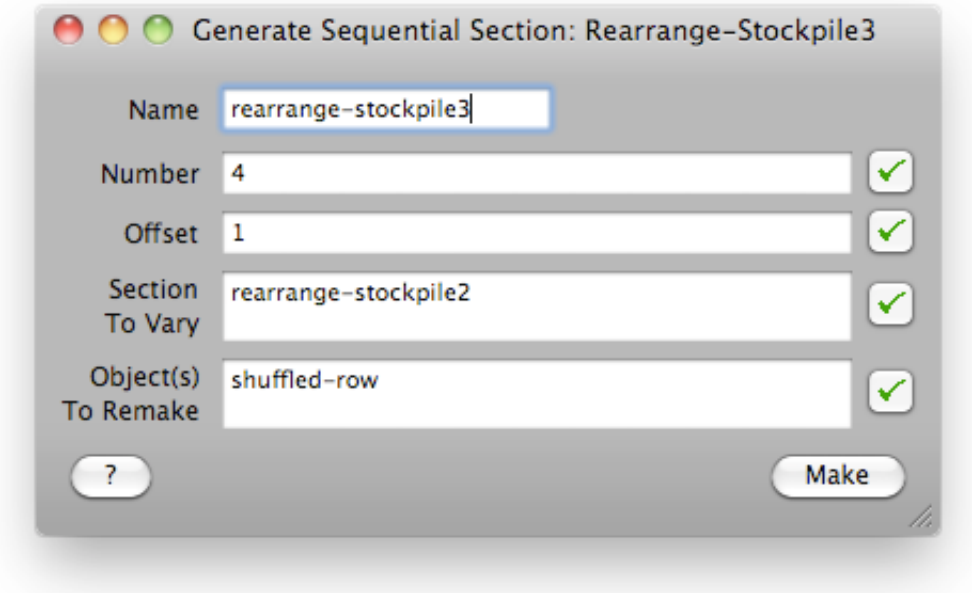
Stockpile *shuffled-row* shuffles stockpile *row1*.

```
Name            shuffled-row
Tool            (rearrange-stockpile row1 'shuffle)
```

Section *rearrange-stockpile2* uses stockpile *shuffled-row* and adds a random value to the result. `(make (rv 60 72))` produces one random value so that all values in the row are added to the same value. Each time the section is made, it could have a different value in the range 60-72 to add to *shuffled-row*.

```
Name            rearrange-stockpile2
Clock unit      200
Number          12
Rhythm          1
Pitch           (cal (make (rv 60 72)) + shuffled-row)
Velocity        mf
Channel         1
```

Section *rearrange-stockpile3* generates a section made of 4 variants of *rearrange-stockpile2*. Stockpile *shuffled-row* is remade before each variant. The result is that each variant will have a different order for the row.



Since the original stockpile is not changed, *rearrange-stockpile* always transforms the original stockpile, not the result of the transformation. For a continuous rearrangement, see the discussion of the generator *rearrange* later in this tutorial.

Act-sort
A list or stockpile can be sorted using functions and logical combinations of functions that are provided by the user. All values < c4 could be moved to the front of the list. Or all high or low values could move to the front. Values that meet the condition(s) are moved to the head of the list.

```
(ACT-SORT STOCKPILE FUNCTION1 &OPTIONAL VALUE1 LOGICAL FUNCTION2
VALUE2)
```

Act-sort does not change the original stockpile.

The only input required is a stockpile and a function. If the function is <, the stockpile is sorted in ascending order. If it is >, the stockpile will be sorted in descending order.

```
(act-sort (from 1 20) >)
(20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
```

Optionally, a value can be given. This will be used as an argument to the function. If *function* is > and *value* is 10, the values that are > 10 will move to the front of the list. Note that they are moved to the front of the list in the order they had in the original stockpile.

```
(act-sort (from 1 20) > 10)
(11 12 13 14 15 16 17 18 19 20 1 2 3 4 5 6 7 8 9 10)
```

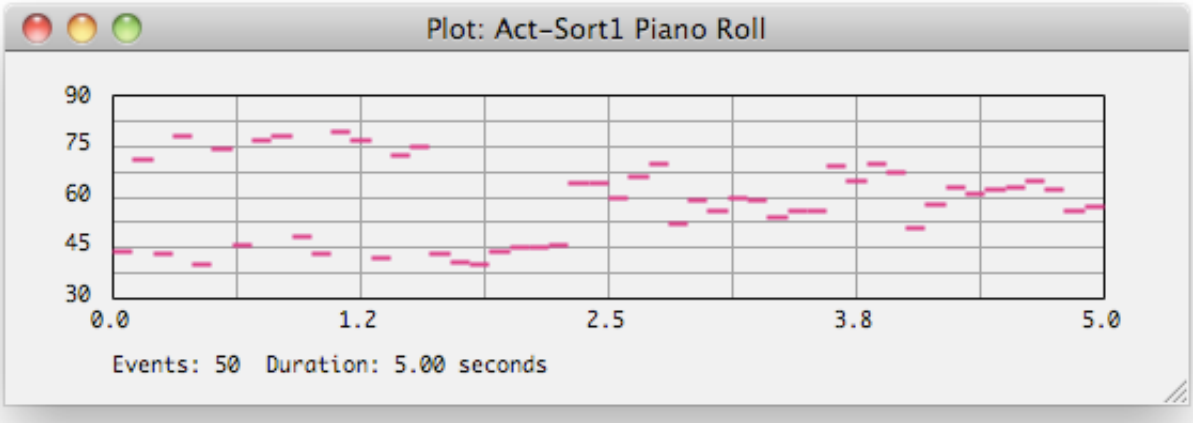
A second condition can be specified. It can be related to the first condition with a logical operation. Parameter *logical* can have the value *and* or the value *or*. *And* is true if both of the conditions are met. *Or* is true if at least one of the conditions is met. The second condition is specified by *function2* and *value2*. To move values that are > 10 and < 15 to the front of the list:

```
(act-sort (from 1 20) > 10 and < 15)
(11 12 13 14 1 2 3 4 5 6 7 8 9 10 15 16 17 18 19 20)
```

Stockpile *to-sort1* has 50 random values in the range 40-80. These values will be used for pitch.

Section *act-sort1* sorts those pitches so that low (< 50) or high (> 70) values come before the other ones.

```
Name            act-sort1
Clock unit      100
Number          50
Rhythm          1
Pitch           (act-sort to-sort1 < 50 or > 70)
Velocity        mf
Channel         1
```



Section *act-sort2* sorts the pitches so that values in the range 50-70 come first. It also produces a list of 50 rhythm values and sorts them in ascending order.

```
Name            act-sort2
Clock unit      100
Number          50
Rhythm          (act-sort (produce 50 (rv 1.0 3)) <)
Pitch           (act-sort to-sort1 > 50 and < 70)
Velocity        mf
Channel         1
```

Generators to change order

Generators continually produce the next value in their series. They can provide a continuous reordering of stockpile data.

Rearrange

The generator *rearrange* returns values from a list or stockpile. The order of the values constantly changes according to a list of indices.

```
(REARRANGE STOCKPILE INDEX-RULE)
```

Index-rule is a list or generator that produces index values. The index of the first value in the stockpile is 1. The stockpile is read in the original order. Then it is read using the indices in the *index-rule* to determine the order. Using the new order, the values are read again using the *index-rule*. The sequence of values is continually changing based on the order specified in the *index-rule*.

```
20 values from:
(rearrange '(a b c d e) '(3 1 4 5 2))
        a       b       c       d       e
        c       a       d       e       b
        d       c       e       b       a
        e       d       b       a       c
```

In the above example, the values a, b, c, d, and e occur in that order. Then the third value, c, becomes the first value. The first value, a, becomes the second, etc.

Section *rearrange1* produces a list of 6 pitches and then continually rearranges them with the rule '(4 1 2 6 5 3).

```
  Name            rearrange1
  Clock unit      200
  Number          48
  Rhythm          1
  Pitch           (rearrange (produce 6 (series-value 40 80)) '(4 1 2 6
5 3))
  Velocity        mf
  Channel         1
```

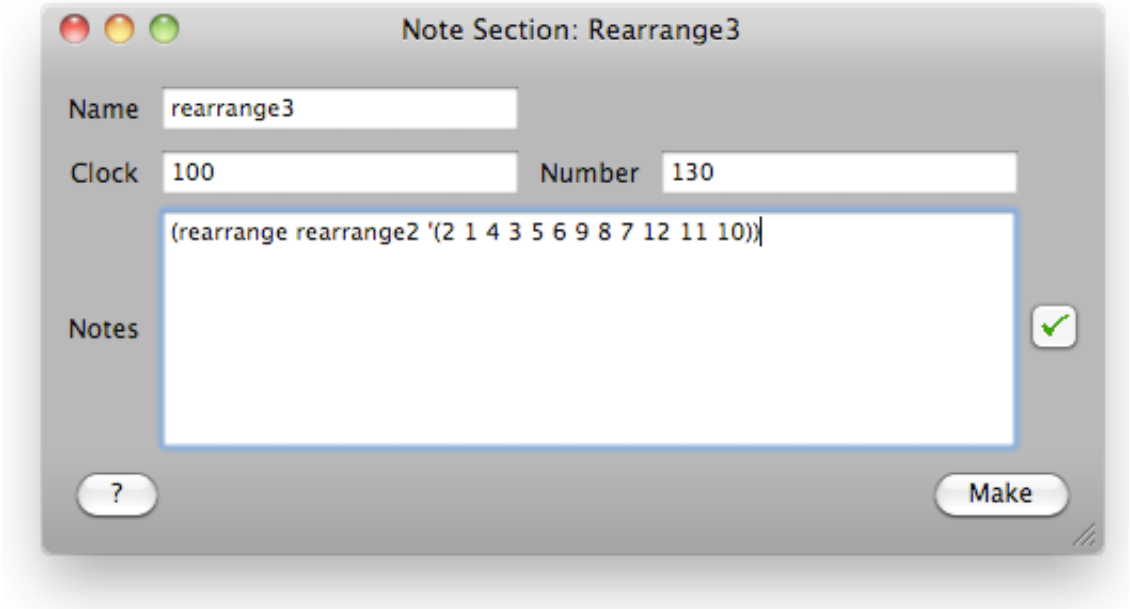
A possible result for pitch in this section is:

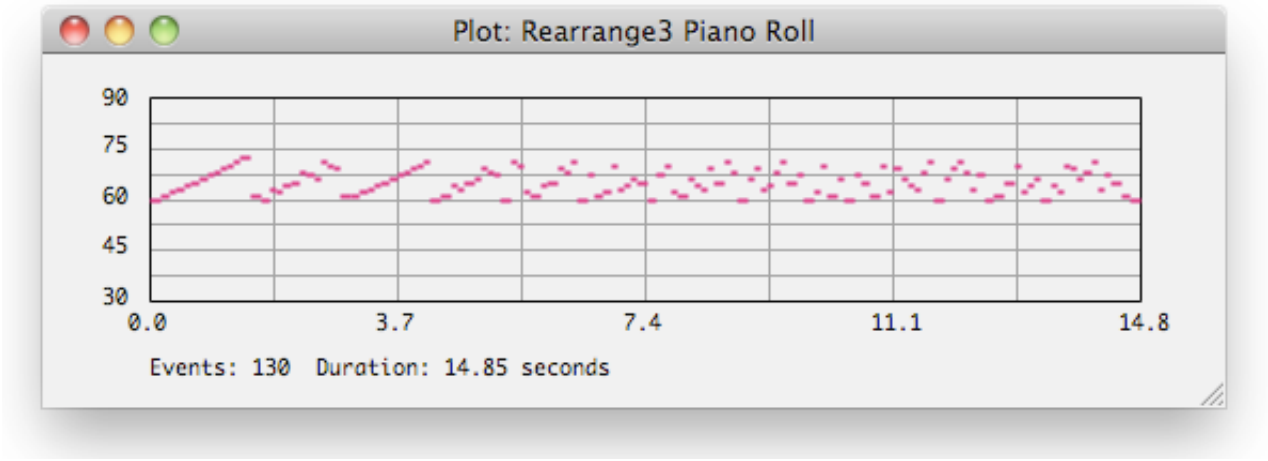
```
List returned from:
(extract 'pitch rearrange1)

(
        79      64      63      61      74      68
        61      79      64      68      74      63
        68      61      79      63      74      64
        63      68      61      64      74      79
        64      63      68      79      74      61
        79      64      63      61      74      68
        61      79      64      68      74      63
        68      61      79      63      74      64
)
```

The first row is the original list. The second row is that list rearranged using the *index-rule*. The third row is the rearrangement of the second row, etc.

The stockpile can be derived from a section. When a section is the input to *rearrange*, a list of notes in sequential order from that section is used as the stockpile.

Section *rearrange2* has 13 pitches of a chromatic scale. Note section *rearrange3* continually rearranges those 13 notes.





Shuffle

The generator *shuffle* randomly orders a list or stockpile or just about any other AC Toolbox object. It returns the shuffled values in sequence and then shuffles again.

```
(SHUFFLE STOCKPILE &OPTIONAL SIZE)
```

The unusual thing about this generator is that the stockpile can be subdivided. Reordering will then be limited to each subdivision.

Size can be a list indicating the size of the various subdivisions. If *size* is a list with (3 3 4), the first two subdivisions will have three values and the last one will have four values.

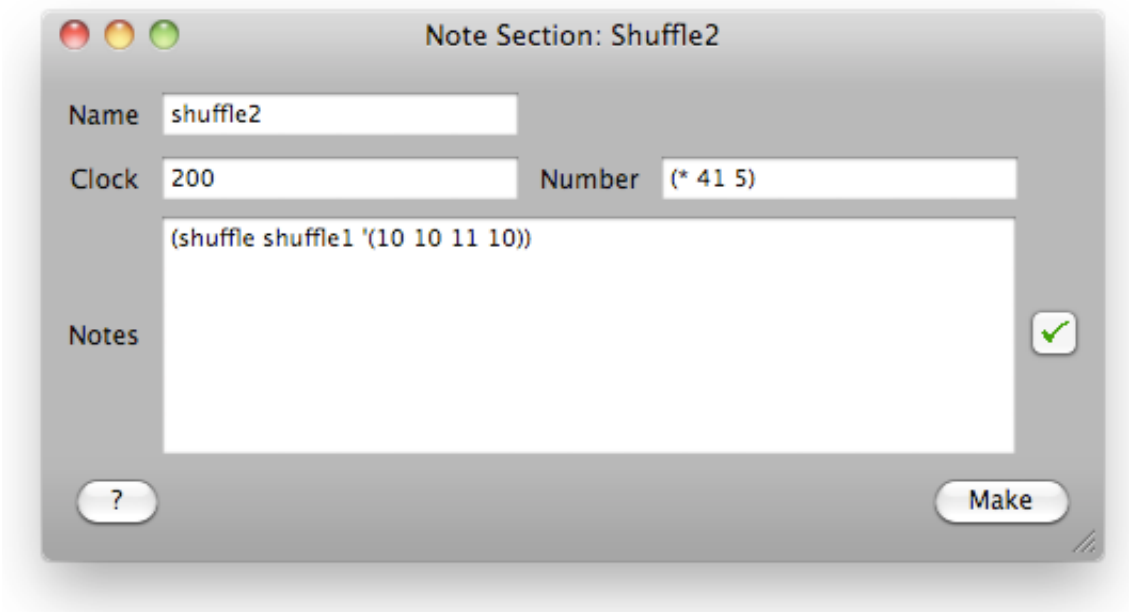
```
40 values from:
(shuffle (from 1 10) '(3 3 4))
          3    1    2    4    5    6   10    7    9    8
          2    3    1    6    5    4    8    9   10    7
          2    3    1    4    5    6    8   10    7    9
          1    2    3    5    4    6    7    9   10    8
```

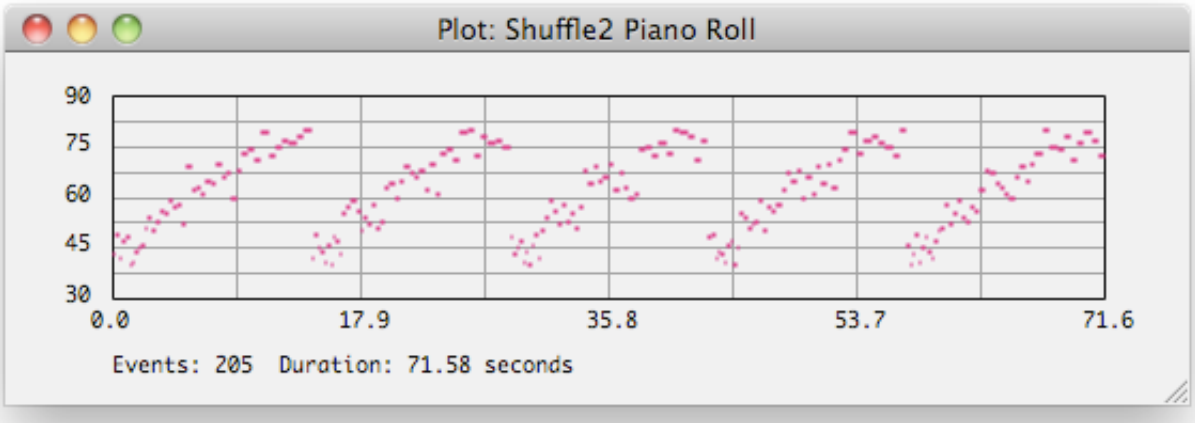
Section *shuffle1* will be used as source material for a shuffle. It contains ascending pitches and increasing rhythmic values.

```
Name            shuffle1
Clock unit      200
Number          41
Rhythm          (act-sort (produce (from-number) (rv 1.0 3)) <)
Pitch           (from 40 80)
Velocity        mf
Channel         1
```

Shuffle2 is a note section that shuffles the notes of *shuffle1.* The subdivision sizes are 10, 10, 11, and 10. This means that the first 10 notes are reordered, then the next 10, etc.

Shuffle2 has enough notes to produce 5 iterations of the shuffling process.





Interrupting sections

A section can be transformed by inserting other sections into it. The original section is interrupted at specified times.

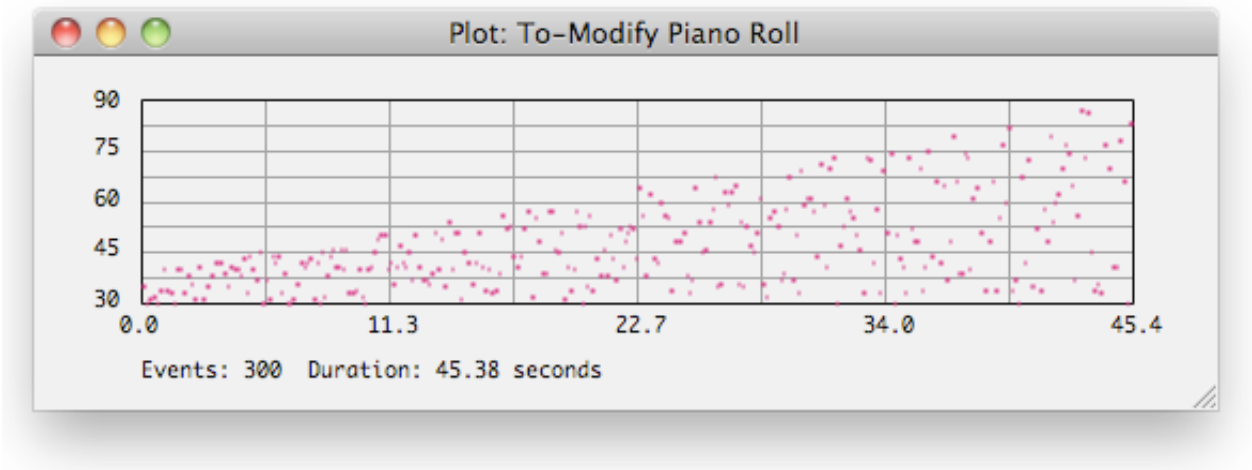
Insert-to-section
The tool *insert-to-section* will insert one or more sections into another section.

```
(INSERT-TO-SECTION SECTION-TO-MODIFY TIMES SECTIONS &KEY (MARGIN 0)
```

The *times* for these insertions can be a list or stockpile of start times in milliseconds. The *sections* to be inserted can come from a list, stockpile, generator, etc.

Section *to-modify* uses a simple tendency-mask for pitch. The upper boundary increases linearly.

```
Name            to-modify
Clock unit      100
Number          300
Rhythm          (random-value 1.0 2 :round .25)
Pitch           (tendency-value '(300 30 40 30 90))
Velocity        (random-value p f :round 5)
Channel         1
```



Two sections will be inserted.

To-insert1 is a random-walk with 100 events. The clock unit is half of that in *to-modify*.

```
Name            to-insert1
Clock unit      50
Number          100
Rhythm          (random-value 1.0 2 :round .25)
Pitch           (walk 60 (rv -5 5) 20 90)
Velocity        (random-value p f :round 5)
Channel         1
```

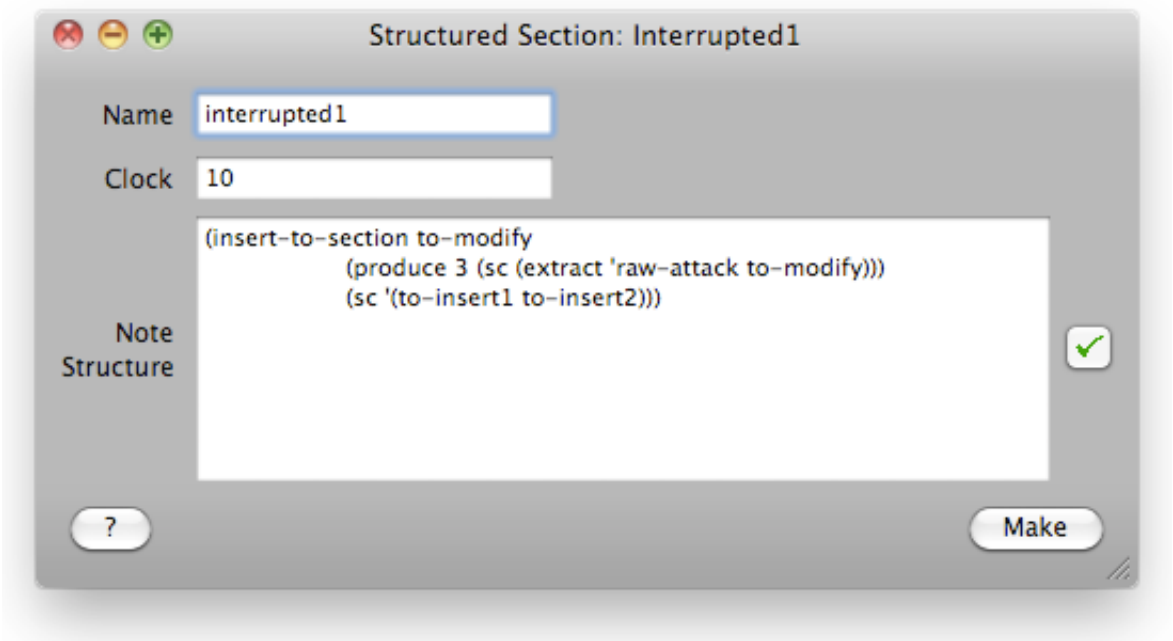
To-insert2 consists of 4-note chords. No repeated values occur in the chords.

```
Name            to-insert2
Clock unit      100
Number          25
Rhythm          (random-value 1.0 2 :round .25)
Pitch           (make-chord (random-value 40 80) 4 :block 0)
Velocity        (random-value p f :round 5)
Channel         1
```

Times should contain start-times that actually exist in *section-to-modify.* A choice can be made from the start times seen in the Edit window for *to-modify*. Another possibility is to extract start times from *to-modify* and to choose from those values.

```
(extract 'raw-attack to-modify)
```

Insert-to-section can be used to make a new structured section. The clock unit is ignored if this tool is used to make a structured section.



In *interrupted1*, three start times are chosen with series-choice from a list of actual start times in section *to-modify*. *Series-choice* is also used to pick the order of the two sections being inserted. When a section is inserted, the remaining values in the original are shoved back in time.

To limit the choices of start times to values >= 10000, *interrupt2* uses the following expression to pick 3 start times:

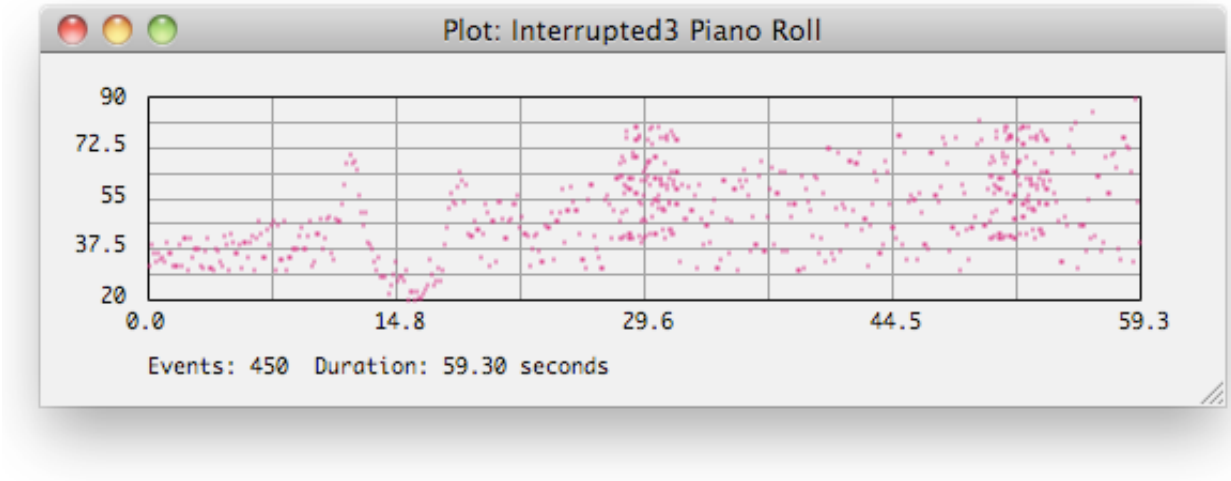
```
(produce 3 (with (anything >= 10000)
                 (sc (extract 'raw-attack to-modify))))
```

Only values >= 10000 will be kept. Other values will be rejected and a new choice will be made.

To get more control of the moments when section *to-modify* will be interrupted, the actual attack times of *to-modify* will be stored in stockpile *insert-times*. Different portions of that stockpile can be used when choosing a time for an interruption. The following expression returns a list of values from *insert-times*: one value in the range 10000-15000, one in the range 20000-25000, and one in the range 35000-40000.

```
(produce 3
  (take 1 (with (anything inside 10000 15000) (rc insert-times))
        1 (with (anything inside 20000 25000) (rc insert-times))
        1 (with (anything inside 35000 40000) (rc insert-times))))
```

Section *interrupt3* uses this expression to determine where the insertions should be made.



Summary

Generators
rearrange, shuffle

Tools
act-sort, insert-to-section, rearrange-stockpile, transform-stockpile

Transformers
insert-rest, limit-range, quantize, tran, translate, transform-and, transform-by-index, transform-if, transform-or