Mikael Laurson and Mika Kuuskankare

PWGL Book

(version 1.0 beta RC 17)

July 31, 2011

Contents

Introduction

PWGL is an OpenGL based visual programming language specialized in computer aided composition and sound synthesis. PWGL is a successor of PatchWork (PW) and aims to develop and improve many of the useful concepts behind PW. PWGL provides a direct access to its base languages, Common Lisp and CLOS. Besides a library of basic boxes (arithmetic, lisp functions, list handling, loops, abstractions, conversion, etc.), PWGL contains several large scale applications, such as:

(1) Expressive Notation Package, ENP (represented in PWGL by Score-editor and Chord-editor)

(2) 2D-editor

(3) PWGLSynth

(4) Constraints

This Help Viewer contains some basic information and several example patches demonstrating PWGL. For more information see the entry 'Overview/Tutorials'. PWGL is distributed as freeware. Currently, it runs under Mac OS X (Universal) and Windows XP operating systems. It is downloadable from our web page (www.siba.fi/PWGL). PWGL is distributed in two different configurations: as a standalone application, called 'PWGL-application', that is targeted mainly to end users, and as a developers version, 'PWGL-binaries', that requires the LispWorks 5.0 programming environment (www.lispworks.com). This version will be made available as a pre-compiled module that is loaded on top of LispWorks.

1 Overview

1.1 Quick-Start

A PWGL patch is the main workspace where the user can add boxes and create relations between them using connections.
All main operations are performed using a 3-button mouse, where the left mouse button is used for selecting, dragging, adjusting the size of a box and editing input boxes. The scroll wheel (middle button) is used for pan and zoom operations. The right button is used for context sensitive popup menus (there are separate popup menus for: main window, each box type, each input box type, and connections).
When the mouse is moved above a patch window the cursor indicates what operations are possible in the current mouse position.
Boxes can be added using the window popup menu (right-click on the window area). Boxes can be positioned by dragging them (left-click on the box area). Connections can be created by clicking the mouse on an output of a box and dragging the resulting red connection line above an input-box of another box (the connection line becomes green). Selected boxes and connections can be removed from the patch by typing the 'delete' key or by using the cut operation.
Some operations are typically performed directly from the keyboard such as: patch evaluation ('v'), playing ('space'), and box or window documentation ('d'). For more information see the 'Keyboard Shortcuts' menu entry under the 'Help' menu.

1.2 User-Interface

1.2.1 Utilities Menu

When PWGL is launched the main menu bar contains a menu called 'Utilities'. This menu is used to switch between Listener (Command/Control+L, not available in PWGL-Application), PWGL output (Command/Control+B) and PWGL (Command/Control+1). Command/Control+2 and Command/Control+3 can be used to access the current ENP and 2D window. If the current window is a PWGL, a ENP or a 2D window, the menu bar contains a 'Help' menu to access tutorials, keyboard shortcuts, documentation, etc.

1.2.2 Mouse Operations

PWGL requires currently a 3-button mouse (left button, scroll wheel, right button). Mouse operations are as follows (these operations typically require only one hand):

(1) move (i.e. no buttons are pressed) changes the cursor ('arrow', 'hand', 'pointing-finger', 'resize', 'left/right-resize', 'up/down-resize', 'cross') depending on what kind of object is under the current mouse position. The cursor shape gives a hint of what operation is going to occur if the mouse is being pressed.

(2) left button can be used to select boxes/input-boxes/connections, move boxes,resize boxes, scroll menu input-boxes, edit numerical values, etc. If left button is double-clicked the system typically opens an editor depending on where the click occurred (window-click opens a Lisp function dialog, box-click a box-editor, abstraction-click an abstraction, editor input-box-click an editor application window, etc.). The shift key can be used to extend the current selection.

(3) scroll wheel (middle button) allows to pan (middle button drag), or zoom (scroll wheel up/down) either globally or locally. Control middle button drag allows to zoom continuously.

(4) right button is used for context sensitive popup menus. A window-click opens a window popup menu, a box-click a box popup menu, an input-box-click a input popup menu, a connection-click a connection popup menu, and so on.

1.2.3 Keyboard Shortcuts

Keyboard shortcuts can be inspected by selecting the 'Keyboard Shortcuts' menu from the 'Help' menu.

1.2.4 Documentation

Window documentation can be accessed by typing 'd' (no boxes should be selected) or from the window popup menu. Box documentation can be accessed by selecting the box and typing 'd' or from the box popup menu.

1.3 PWGL-Keyboard-Shortcuts

Shortcut	**Documentation**
Command + n	New patch
Command + o	Open patch
Command + s	Save patch
Command + S	Save patch as
Command + e	Export EPS
Command + u	Load library

Command + U	Autolaod libraries
Command + z	Undo
Command + x	Cut
Command + c	Copy
Command + v	Paste
Command + a	Select All
Command + d	Duplicate
Command + f	Fit in Window
Command + .	Stop PWGL Process(es)
Command + ?	PWGL Help
Up	move an input box up/down
Down	move an input box up/down
Left	move to previous measure
Right	move to next measure
Backspace	delete boxes or connections
Tab	wiki-link-show-target
Shift + Tab	shift-Tab: select next window snapshot
Escape	Reset frame-box mode
Space	start/stop ENP score/2D-editor
+	add input-box/input-boxes
-	remove input-box/input-boxes
0	set box-string to empty
1	send 'user-key-received' message to boxes with trigger-string = 1
2	send 'user-key-received' message to boxes with trigger-string = 2
3	send 'user-key-received' message to boxes with trigger-string = 3
4	send 'user-key-received' message to boxes with trigger-string = 4
5	send 'user-key-received' message to boxes with trigger-string = 5
6	send 'user-key-received' message to boxes with trigger-string = 6
7	send 'user-key-received' message to boxes with trigger-string = 7
8	send 'user-key-received' message to boxes with trigger-string = 8
9	send 'user-key-received' message to boxes with trigger-string = 9
A	add special synth output for recursive connections
C	set box complement color
D	show slider numeric display
L	toggle abstraction lambda mode on/off
M	minimize/maximize box
N	Select previous frame-box
P	open play-mixer window
S	select current ENP-score
T	edit tags
X	align selected boxes by x co-ordinates - center alignment
a	apply add operation to selected boxes
c	set window color or selected box/input-box/connection color
d	show documentation

e	edit definition
f	print code compilation expression of an abstraction in 'lambda' mode
h	Show man
i	inspect
k	apply kill operation to selected boxes
l	toggle lock on/off
m	switch box input connection mode or connection draw mode
n	Select next frame-box
o	open application windows or open box
p	set shell-box post-process
q	toggle quicktime player on or off
r	reset PWGL box
s	stop synth or stop PMC processes
t	Show box tutorial
v	patch-value from the selected output(s) or the left-most output
x	align selected boxes by x co-ordinates - the reference point is given by the upmost box
y	align selected boxes by y co-ordinates - the reference point is given by the left-most box
z	zoom inside next sweep selection
F7	replace an old version of a box
F8	print synth-debug info
F12	print decompilation expression

1.4 2D-Keyboard-Shortcuts

Shortcut	**Documentation**
Command + o	Open 2D-objects
Command + w	Close window
Command + e	Export midifile
Command + i	Import midifile
Command + z	Undo
Command + x	Cut
Command + c	Copy
Command + v	Paste Object
Command + V	Paste selection
Command + a	Select All
Command + d	Duplicate Object
Command + f	Fit in Window
Command + ?	PWGL Help
Enter	select main PWGL window
Up	bpf: increase selected point distances in x direction

Down	bpf: decrease selected point distances in x direction
Prior	scroll to previous page
Next	scroll to next page
Home	scroll to begin
End	scroll to end
Left	bpf: decrease selected point distances in y direction
Right	bpf: increase selected point distances in y direction
Backspace	delete active object
Tab	select next object as active
Shift + Tab	select previous object as active
Space	play/stop sample/chord-seq
0	x scroll to 0
=	x/y scroll to 0/0
F	fit inside selection
L	fit inside co-ordinate limits
R	reload default sample
c	edit active object color
f	fit active-object in window
i	inspect 2D view
n	rename active object
z	zoom inside next sweep selection

1.5 Main-Menus

1.5.1 File

Contains standard file menu-items such as: New, Open..., Close Window, Save, and Save As.... The menu-item 'Export EPS...' exports the current patch as an '.eps' file. Finally there are options to load libraries. The term 'PWGL-library' refers to official libraries that are distributed inside the 'PWGL' folder to all users, whereas the term 'User-library' refers to private user libraries that are typically found in the 'PWGL-User' folder (inside your home folder). You can install your own libraries in both places - installing them outside the PWGL-Application folder has the simple benefit of not accidentally deleting them, if you install a new version of PWGL. For more information on writing your own libraries see the Section "Libraries".

1.5.2 Edit

Contains standard edit menu-items: 'Undo', 'Cut', 'Copy', 'Paste', 'Select All', and 'Duplicate'. All these have a more or less standard behaviour, except for 'Undo', that works only when making destructive changes where boxes are removed (either using cut or delete operations) from a patch. Note that undo has to be called immediately after deletion operation.

1.5.3 PWGL

Contains menus for 'About PWGL' and three preference options: 'PWGL Preferences...', 'Audio/MIDI Setup...', and 'Sample Paths...'. Two menu-items, 'Fit In Window' and 'Stop PWGL process', allow to scale the contents of a patch to fit the current window size and to stop the current process. Each of the preference options open a dialog, where the user can specify default values and/or behaviour of the PWGL system. Each option can be saved and loaded (the preferences are saved inside 'PWGL-User'). Preferences are discussed in more detail in the section 'Preferences'

1.5.4 Patches

Contains menu-items for all main patch windows that are currently open. The current patch has a 'carrot' sign before the patch name. All patches that have been modified are marked with a 'dot'.

1.5.5 Help

is used to access tutorials and documentation ('PWGL Help...' and 'ENP Help...'), and keyboard shortcut listings ('Keyboard Shortcuts...'). This menu also contains two reference dialogs. The first one, 'PWGL box reference...', lists all PWGL boxes that are found in the main window popup menu of a PWGL window. The dialog can be sorted either according to box name or to menu name. It shows the documentation string of the selected box and also lists all tutorials where this box is used. The tutorials can be opened by clicking at the respective tutorial pathname. The second one, 'Constraints', serves as a reference for the most important utility functions used by the constraints system.

1.5.6 Utilities

is used to switch between Listener (Command/Control+L, not available in PWGL-Application), PWGL output (Command/Control+B) and PWGL (Command/Control+1). Command/Control+2 and Command/Control+3 is used to access the current ENP and 2D window.

1.6 Preferences

1.6.1 PWGL Preferences

contains defaults for patch windows, window popup menus, connections, boxes, synth, 2D, ENP and MIDI.

(1) General The 'Active menu items' option is used to specify which menus are visible in the main window popup menu when the 'Menu Filter' option (see the next preference option) is on. If the 'Menu Filter' option is off, then all kernel and loaded library menus available in the system are shown. The 'Menu Shortcut' option specifies which menu item can be accessed directly using a shift right-button click. The

'Compact tutorial' option allows to specify whether the tutorial window is drawn in compact mode or not.

(2) Appearance Here the user can control various appearance options dealing with default connection mode, box coloring scheme, skin name, and window scaler.

(3) Synth Contains general sound synthesis defaults that will be copied by each new 'synth-box' (these defaults can be locally modified and saved using the box editor). These parameters deal with saving of the output of a synthesis patch to a file (i.e. when a synthesis patch is run in 'file' mode) and they are used to define the current file format, sample rate, bit depth, file mode, and oversampling. The parameter 'extra time' is used to extend the file length when working, for instance, with reverberated signals that require extra decay time. 'Pathname' gives the current sound sample pathname.

(4) Constraints When 'Multi-search' is set to 'better response', then during search the graphics part of the system is responsive (i.e. the user can stop processes, play scores, check partial solutions). This option is usually recommended as the user has full control of the system. If 'Multi-search' is set to 'better performance', the system becomes less responsive and it may take several seconds before the system responds to mouse clicks. The search is here typically much faster than in the first option. This mode is recommended when a search has been already tested in the slower mode, and the user needs full speed.

(5) MIDI Contains the current MIDI device. Specialized MIDI devices can be defined by the user. The pitchbend range parameter is used for micro-tonal tuning (note that this parameter should match the pitchbend settings of your MIDI synthesizer). Micro-tonal tuning is realized automatically by analyzing the score and reserving channels for playback. If a score can be played with equal temperament then only one channel is reserved, in case of 1/4 tone temperament 2 channels are reserved, in case of 1/8 tone temperament 4 channels are reserved, and so on. The maximum resolution is 1/64 tone temperament that requires 16 channels. Play speed scaler is used to scale the playback speed of MIDI information (1.0 no change, 0.5 half speed, 2.0 double speed). Chord-editor arpeggio speed is given in seconds. When 'Use default startup volume' is on, then before the actual playback starts, the system sends to all channels MIDI volume messages (the volume value is given in the next parameter input). When 'Play continuous control' is on, then the system automatically sends control information when it encounters in the score continuous dynamics expressions (either crescendo or diminuendo markings), or Score-BPFs (the type of the Score-BPF must be ':midi-cc'). In the latter case the the Score-BPF can contain up to three separate bpfs each controlling its own continuous control information. The controller numbers (1-127) for each case (dynamics expression, bpf 1-3) are given in the next four parameter inputs.

1.6.2 Audio/Midi Setup

is used to define current audio out/in and MIDI devices. In Mac the user has also the option to use the internal QuickTime synthesizer. PWGL supports eight MIDI-out

ports (’A’ channels 1-16, ’B’ channels 17-32, ’C’ channels 33-48, ’D’ channels 49-64, ’E’ channels 65-80, ’F’ channels 81-96, ’G’ channels 97-112, ’H’ channels 113-128), for MIDI-in there is only one port.

1.6.3 Sample Paths

A PWGL patch typically saves all required information (e.g. windows, boxes, connections, text-files, etc.) so that the patch is functional even when it is later used in another machine or even in another operating system. Sound samples are an exception in this scheme as only the pathname of the sound sample is saved. This can potentially create problems if a patch is loaded in another environment. ’Sample Paths’ can be used to define directory pathnames that are used in case the absolute sample pathname that was stored along with the patch is not found. Each directory pathname is tested one by one if a sample with the same file name is found within the current directory.

1.7 Tutorials

The tutorial part of this Help Viewer aims to demonstrate how PWGL works in practice. The examples are organized as a hierarchical folder structure - shown in the left part column of the help window - that contains at the leaves patch examples (file names with the extension ’.pwgl’). When a patch file name is selected, the respective patch window is opened in the right part of the tutorial window (if a patch is large there can be noticeable delay, because the patch is loaded from the hard disk). The patches are fully functional, i.e. they can be evaluated, played, boxes and connections can be added or deleted, and so on. Saving is disabled. Some tutorial patches have a window documentation text which can be opened by typing ’d’ (no boxes should be selected) or by choosing the ’Window documentation...’ option in the window popup menu.
There is also a dedicated tutorial for ENP (found in the ’Help’ menu).

1.8 Programming-Interface

PWGL is a cross-platform application. Therefore, when naming files and folders you should avoid the following directory separator and wild-card characters: ’:’, ”, ’/’, ’*’, ’?’, ””, ’<’, ’>’, and ’|’. Generally you should also avoid spaces in file and directory names as they pose legibility problems at least in Unix based systems and Web browsers. The recommended characters in safe cross-platform file names include ’a-z’, ’A-Z’, ’0-9’, ’-’, and ’_’. If you plan to use mixed mode alphabetic letters you should also note that most Unix file systems are case sensitive. Also, if you plan to transfer files over a network, certain server software may truncate long file names.

1.8.1 Errors

As PWGL-Application is a programming environment, the user will encounter occasionally error situations. PWGL has error handling routines that will open a dialog with a

text indicating the cause of the error. By clicking the 'OK' button the user can typically continue to use PWGL without having to restart the system. Sometimes, however, an error situation may open the 'Terminal' application. In this case the 'Terminal' window displays a list of numbered options that indicate how to continue. Normally it is best to choose the 'abort' option by writing: ':a' followed by return or enter.

1.8.2 Textual Programming

PWGL-Application contains three basic textual tools that allow Lisp programming: text-box, Lisp-code box, and code-box. Code evaluation and compilation options are found in the 'Eval' menu.
For your personal code never use the system package ':ccl' or 'system'.
For more details see the tutorial patches:

(1) Basic/programming.pwgl

(2) Basic/box-creation.pwgl

1.8.3 Libraries

The 'PWGL-library' folder in the distribution folder contains an example library template called 'mylib'. This example contains several demo boxes and a hierarchical user menu. See also the next entry of this tutorial.

1.9 Libraries

1.9.1 General Information about PWGL Libraries

1.9.1.1 Where Libraries Are Installed

PWGL searches for Libraries in two locations. One inside the PWGL-Application folder, the other inside your home folder (click on the links below to see the exact location).

1.9.1.2 Creating a PWGL Library

The easiest way of creating a new library is to go to File ▶ Create PWGL Library... and use the Library Tool to fill in the appropriate information. When finished the tool creates the required components, all the needed files and folders along with some additional parts, such as dynamically updated front and info pages, logo, etc. It is a good practise to try to provide all the pertinent information about the library. It helps maintaining and distributing your work. It also makes it easier to use some of the advanced documentation features of PWGL, details of which are given in the following sections.

1.9.2 Techincal Details

This section enumerates the key points you need to know in order to be able to develop your own libraries. The PWGL user library scheme relies on the ASDF system definition facility. You should take a moment to get acquainted with ASDF. A good place to start the official documentation: asdf Manual

1.9.2.1 Constituent Bits of a Typical Library

Writing your own libraries requires a couple of files. You can find an example library in pwgl-library/mylib.
The following sections explain briefly the purpose of each file:

FILE	SECTION
mylib.asd	Defining an ASDF system
package.lisp	Defining your own package
standard-lisp-code.lisp	Defining standard lisp code
boxes.lisp	Defining boxes
menus.lisp	Defining menus
tutorial	Tutorial patches
lib-properties.txt	Library properties

1.9.2.2 The ASDF System Definition

An ASDF system basically lists all the source files of your library. Typically, they need to be loaded in a certain order. An easy way of specifying this is to use the option `:serial t` and then list them in the desired order after the :components keyword as shown in the following example:

```
(in-package :asdf)

(defsystem :mylib
  ;; :serial t means that each component is only compiled, when the
  ;; predecessors are already loaded
  :serial t
  :components
  ((:file "package")                    ; use your own package
   ;; macros first, if you need some
   (:file "macros")
   ;; define your boxes and other lisp functions
   (:file "code")
   ;; specify the entries in the popup-menu, which is used to add
   ;; boxes to a patch (right-click)
   (:file "menus")))
```

If you ever need more options for defining your system, you can look at: Defining systems with defsystem

1.9.2.3 Defining Your Own Package

It is recommended that you define your own package for your library, as this keeps its symbols together and also allows for autoloading your library by simply opening a patch that uses it (see also 'Library properties').

1.9.2.4 Defining Standard Lisp Code

You can write of course any lisp code you like! (see 'Compiling your library') Among all the possibilites you have there, you will mainly want to define functions using 'defun' and possibly macros using 'defmacro'.

1.9.2.5 Defining Boxes

The boxes make up the interface of your library - those functions that are actually used from a patch. A simple box definition looks as follows:

```
(PWGLdef fn1 ((a 1))
    "fn1"
    ()
  (list a))
```

PWGL provides a special definer for boxes: 'PWGLdef' A box definition works similarly to a function definition using 'defun' (in fact a box is a function!). But 'PWGLdef' allows you to make a much richer definition. You can specify default values and control the appearance of the box in much detail. For more information see Tutorial/Basic/box-creation.pwgl

1.9.2.6 Defining Menus

This defines the entries in the popup-menu that you get with ctrl-click. If you need more explanation about the syntax of 'add-PWGL-user-menu', please ask us on the mailing list.

```
(ccl::add-PWGL-user-menu
 '(:menu-component
   ("MyLib"
    ((fn1)
     ("fns1"  (fn2))
     ("fns2"  (fn3  fn4)
              ("deepfns2"  (fn5)))
     (:menu-component
      (fn6 fn7))))))
```

1.9.2.7 Compiling Your Library

If you load a library using File ▶ Load library, the library will be automatically compiled, before it is loaded. The rationale of this is that you can write your own lisp code and use it with PWGL (or download any thirdparty lisp code from the internet) even if

you are using PWGL standalone (this is a new feature in PWGL-rc10). Depending on which PWGL system you are using, the compilation happens slightly differently. On all systems, a so called 'fasl-file' will be placed next to the lisp source code file. If you are using PWGL standalone (without LispWorks), for a lisp file called boxes.lisp the compiled version will be called boxes.clufasl. This is a 'universal fasl format', that can be loaded on any platform (mac, windows ...). If you are using the Binary version of PWGL together with LispWorks, a fasl file in the native format of the actualy platform being used will be generated. If you need more information on this, please contact us on the mailing list.

1.9.2.8 Tutorial Patches

If a library contains a folder called 'tutorial' then this folder will appear (with the current library name) in the PWGL Help navigation pane after the library has been loaded. Typically this folder contains example patches and text files. The patches should use a standard size (the correct size can be obtained from Window ▶ Set Tutorial size). It is also recommended that a tutorial patch should contain written documentation (the documentation string can be added using Window ▶ Edit Window...). For more lengthy textual information you can use also a plain text file (with an extension '.txt.') where the text part should be enclosed inside quotation marks.

1.9.2.9 Library Properties

The file 'lib-properties.txt' is used to specify additional properties of a library in a lispy format. For mylib this file currently contains:

```
(:package-names ("my-package"))
```

The property ':package-names' is used here, because the name of the package that mylib uses is different from the library name (this property is only needed in such a case). It allows a library to be automatically loaded, by simply opening a patch that uses it. More additional properties will be used in the future to allow checking for updates on the web etc...

1.10 Documentation

Here you find a collection of articles that are useful for understanding some of the advanced tools in PWGL dealing with constraint programming, scripting, ENP-score-notation, box design, and sound synthesis.
There are 2 articles related to our constraint syntax. The first one is an old text from 1996 that describes the basic PMC syntax. The second one, in turn, describes some more recent work that deals with a new Score-PMC syntax (see the Score-PMC-syntax' page in the 'Constraints' section that explains the most recent syntax).
ENP-scipt is introduced in the following article:
'Recent Developments in Enp-Score-Notation' discusses the main concepts behind ENP-score-notation.

The following paper provides information for users who want to develop software for PWGL.
'Multichannel Signal Representation in PWGLSynth' gives an overview of our synthesis system. We describe how to represent visually multichannel signals in a synthesis patch.

1.11 Publications

Here is a list of more recent publications dealing with PWGL, ENP and PWGLSynth:

M. Laurson and M. Kuuskankare. PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL. In Proc. of ICMC02, pp. 142-145, Gothenburg, Sweden, Sept. 2002.

M. Kuuskankare and M. Laurson. ENP2.0 A Music Notation Program Implemented in Common Lisp and OpenGL. In Proc. of ICMC02, pp. 463-466, Gothenburg, Sweden, Sept. 2002.

M. Laurson and M. Kuuskankare. From RTM-notation to ENP-score-notation. In Journees d'Informatique Musicale, Montbeliard, France, 2003.

M. Laurson and V. Norilo. RECENT DEVELOPMENTS IN PWSYNTH. In Proc. of DAFx 2003, pp. 69-72, London, England, Sept. 2003.

M. Kuuskankare and M. Laurson. ENP-Expressions, Score-BPF as a Case Study. In Proc. ICMC03, pp. 103-106, Singapore, Sept. 2003.

M. Laurson and M. Kuuskankare. Some Box Design Issues in PWGL. In Proc. ICMC03, pp. 271-274, Singapore, Sept. 2003.

M. Laurson and V. Norilo. Copy-synth-patch: A Tool for Visual Instrument Design. In Proc. ICMC04, Miami.

M. Kuuskankare and M. Laurson. Intelligent Scripting in ENP using PWConstraints. In Proc. ICMC04, Miami.

M. Kuuskankare and M. Laurson. Recent Developments in ENP-score-notation. In Proc. SMC04, 2004.

M. Laurson and M. Kuuskankare. PWGL Editors: 2D-Editor as a Case Study. In Proc. SMC04, 2004.

M. Laurson, V. Norilo and M. Kuuskankare. PWGLSynth, A Visual Synthesis Language for Virtual Instrument Design and Control. Computer Music Journal, vol. 29, no. 3, pp. 29-41, 2005.

M. Kuuskankare and M. Laurson. Expressive Notation Package. Computer Music Journal, vol. 30, no. 4, 2006.

1.12 Credits

PWGL is based on many concepts and ideas that were originally developed for PatchWork (Laurson, Rueda, Duthen, Assayag, Agon) and thus credit should be given to numerous programmers, composers and researchers that were involved in this project. However, PWGL has been completely rewritten and redesigned in order to create a modern cross-platform environment for computer assisted composition, analysis and sound synthesis.

1.12.1 Development Team

The current research team at Sibelius Academy behind PWGL consists of Mikael Laurson, Mika Kuuskankare and Vesa Norilo. Recently the team has been augmented by Kilian Sprotte.
The approximate contribution list within the project is as follows:

PWGL: Mikael Laurson, Mika Kuuskankare and Kilian Sprotte

ENP: Mika Kuuskankare

PWGLSynth: Vesa Norilo (C++ code), Mikael Laurson (PWGL interface)

Programming tools and CAPI interface: Mika Kuuskankare

A special thanks goes to Kimmo Kuitunen who has provided several tutorials for PWGL-Help.

1.12.2 Third-Party Libraries

PWGL uses the following software packages that are distributed with their own respective licenses:

PortAudio Portable Real-Time Audio Library Copyright (c) 1999-2000 Ross Bencina and Phil Burk

libsndfile by Erik de Castro Lopo (Released under the terms of the GNU Lesser General Public License.) Copyright (c) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

FOMUS for Lilypond and MusicXML export Copyright (c) 2005, 2006 David Psenicka, All Rights Reserved

MIDI (file I/O) Copyright (c) 2007 by David Lewis, Marcus Pearce, Christophe Rhodes and contributors

2 Tutorial

2.1 Basic

2.1.1 Start-Here

This is a basic demonstration patch generating a sine wave function.
The final result can be seen in the ’2D-Editor’ box that is found in the bottom of the patch.
To inspect any intermediate result select the respective box and press ’v’. The result will be printed in the ’PWGL output’ window.
Documentation of the current patch window can be accessed by pressing ’d’ (note that no boxes should be selected). In the tutorial this information is also shown either on the right side of the window or below the navigation panel
Documentation of individual boxes are accessed by selecting the respective boxes and by pressing ’d’.
For general user-interface issues (mouse operations, keyboard short-cuts, popup-menus, etc.) see the ’User-Interface’ text in the ’Overview’ section.

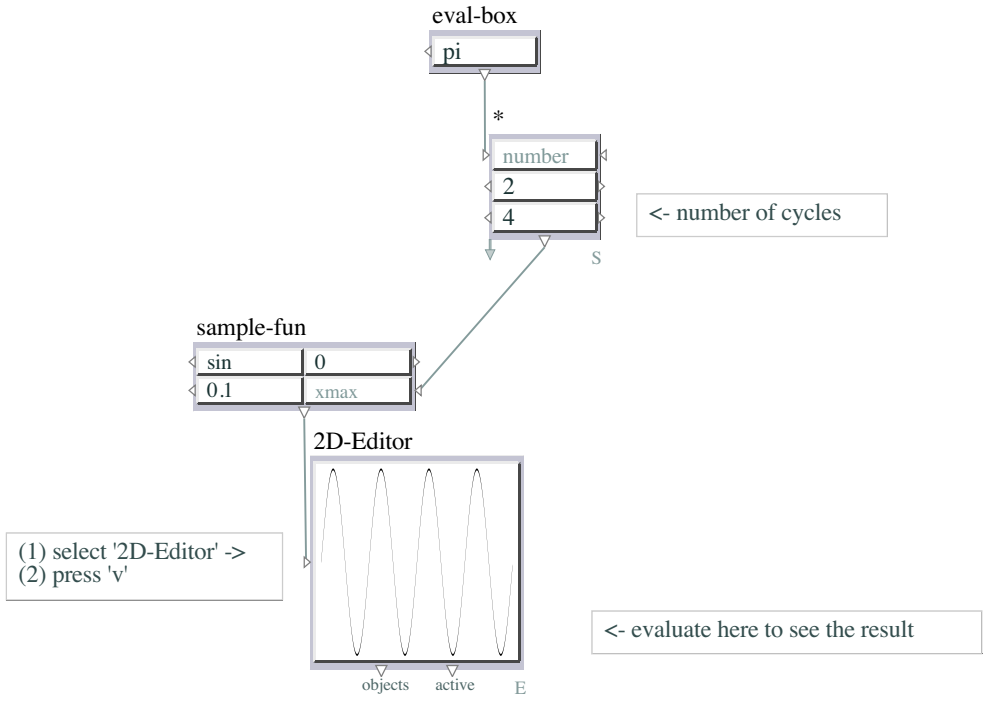

Figure 2.1: 01-start-here

2.1.2 Abstraction

An abstraction-box can be opened by a double-click (1).
An abstraction can have arbitrary many inputs and outputs (2 and 3). Abstraction inputs and outputs are added in the abstraction-window using the window popup-menu.
The box either has the label 'A' or 'La' (for 'Lambda').
In the former case, when evaluated, the abstraction-box returns a value as any other PWGL-box (2 and 3). In example 3 the individual outputs can be evaluated by selecting only the respective output and pressing the 'v' key.
If the label is 'La', the box is in a lambda-state and returns an 'anonymous function', that is, the result of compiling the contents of the abstraction window to a Lisp function (4, 5 and 6). This output can be fed to a box that requires a function as input (typical examples are for instance 'mapcar' and 'PWGL-apply').
Normally the number of inputs of the lambda-state abstraction box correspond directly to the number of arguments of the resulting anonymous function (thus in (4) and (5) the number of arguments is 1). In (6), however, the abstraction box with 2 inputs results in a 1-argument function, as the second input is connected to a value-box. Thus all connected inputs are excluded from the argument list of the resulting function. This

scheme is useful in cases where the user wants to give directly from the top-level patch extra data that is used internally by the function.

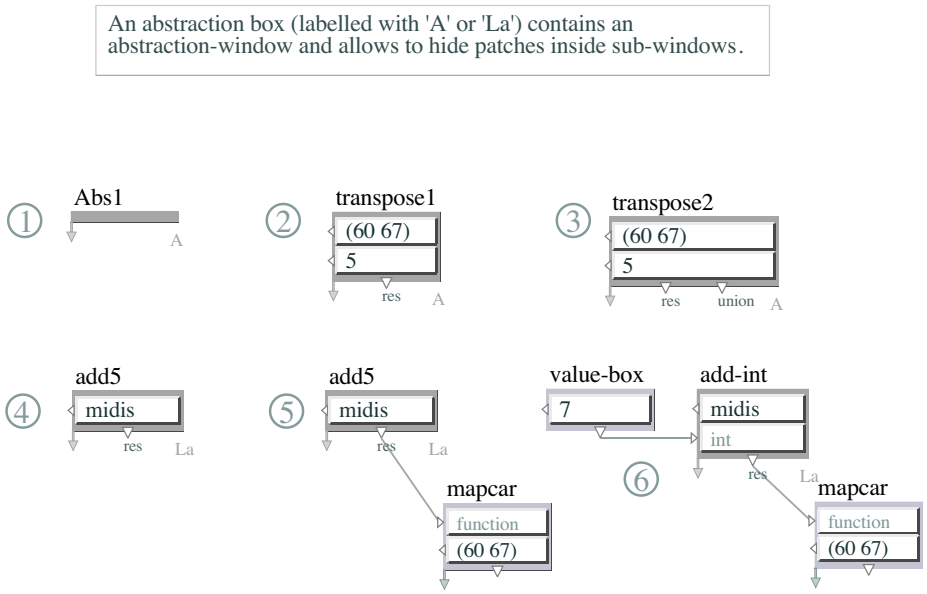

Figure 2.2: 02-abstraction

2.1.3 Extendable

This patch demonstrates some boxes with different argument lists. A box is extendable if it has an arrow in the low-left corner. The arrow either points downwards or upwards. In the first case input-boxes can be added. In the second case the box does not allow anymore new input-boxes and input-boxes can only be removed.
To add input-boxes type '+', to remove them type '-'.
The patch contains boxes with different argument lists.
Boxes that represent normal lisp function argument lists extend boxes either with one input-box (&optional and &rest) at a time or with two input-boxes (&key).
The last box, Multi-PMC, has a more complex extension pattern of 3 and 4 input-boxes.

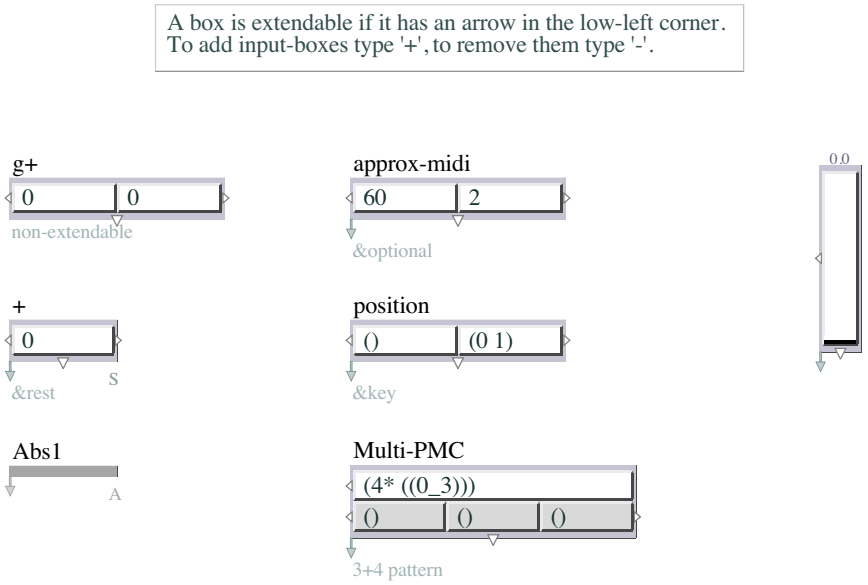

Figure 2.3: 03-extendable

2.1.4 Input-Boxes

The main conceptual unit in PWGL is a box that contains typically one or several input-boxes.
This patch gives an overview of some of the most commonly used input-boxes. The properties of an input-box can be inspected and edited by selecting the 'edit input-box' option from the input-box popup-menu item.
The first row shows the input-boxes for textual Lisp values like numbers, lists, symbols, strings, etc. Note that the value input-boxes support the 'expand-list' short-hand notation that allows to generate automatically lists.
The second row shows several menu input-boxes. Both simple menus and hierarchical menus are supported. A simple menu contains at the top-left side a small scroll triangle. The hierarchical menu is represented by two scroll areas (one at the top-left side, one at the bottom-left side). The top-level menus of a hierarchical menu can be changed by dragging the bottom-left scroll area. Traditional popup-menus can be invoked with a ctrl-click. The last box in the second row, called 'menu-box', is special as it allows to edit the contents of its menu input-box directly in a text-editor. The text-editor can be opened by double-clicking the input-box.

The third row shows a slider input-box and a button input-box.

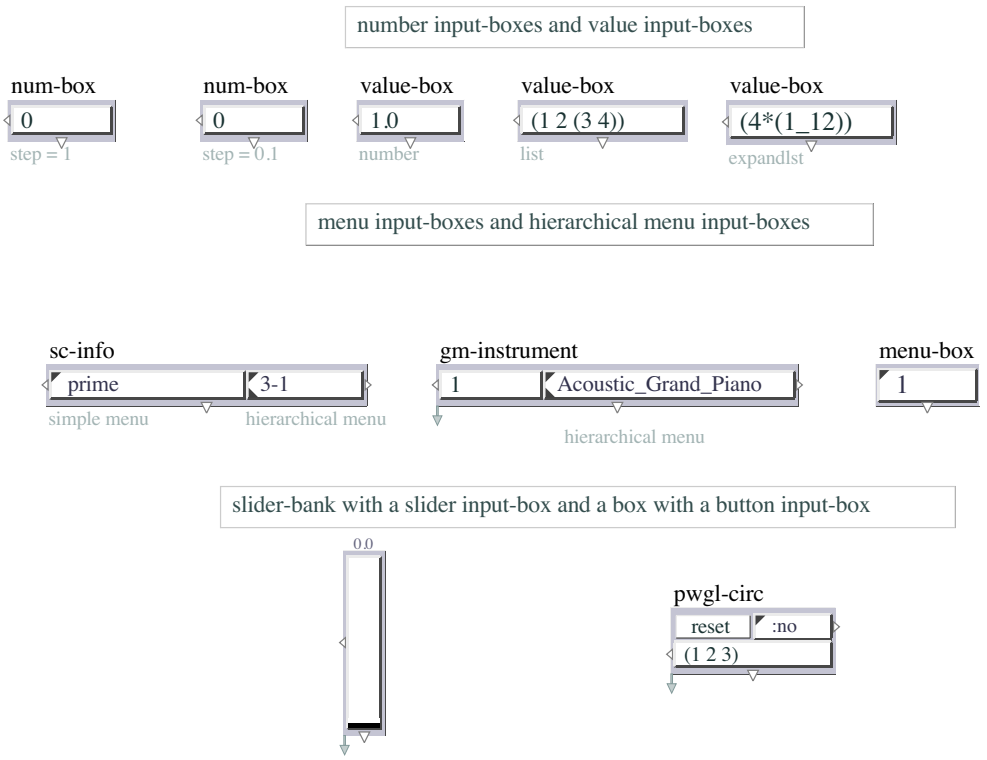

Figure 2.4: 04-input-boxes

2.1.5 Trigger-Boxes

PWGL contains boxes that can be triggered from the keyboard or using external MIDI controllers. This feature is mainly useful when using PWGL synth.
All trigger boxes in this patch, except the first one (code-box), are synth boxes (see the 'S' label at the low-right corner of the boxes).
The trigger string (normally a number ranging from 1-9) can be edited by opening the 'edit box' dialog (to open the dialog use the box popup menu). A green number (or in more rare cases text) will appear at the left side of the box. If the user presses the corresponding number from the keyboard, the box will triggered.
It is also possible to trigger a box using MIDI CC (continuous controller) numbers (1-127). The small green MIDI CC number will appear at the low-right corner of the box (see the upper row of boxes where the boxes have been assigned MIDI CC numbers 64, 65, and 66).
See the 'Synth' tutorial for practical examples how to use the trigger feature.

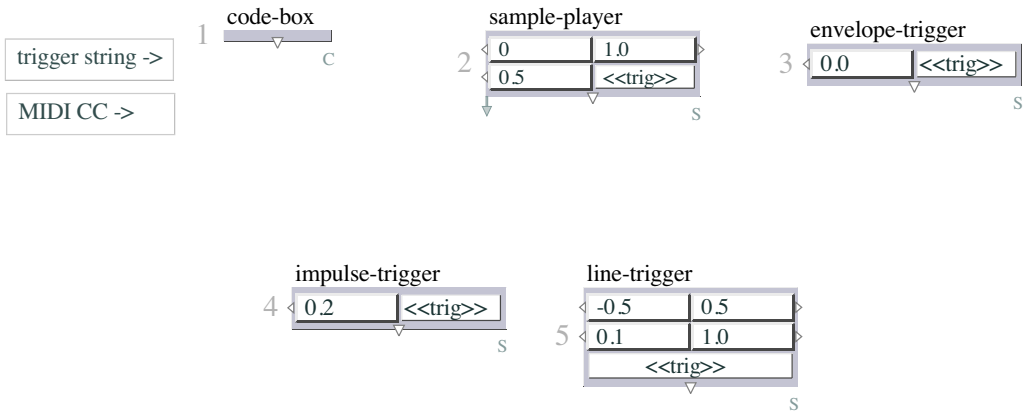

Figure 2.5: 04b-trigger-boxes

2.1.6 Sliderbank

This patch gives some slider-bank examples. The properties of individual slider input-boxes can be edited by selecting the 'edit slider...' option from the input-box popup-menu item.
The patch contains also a slider-bank with predefined slider setups (1). A setup can be recalled by clicking one of the buttons below the slider-bank box. Individual slider setups can be stored by pressing 'a'. This will store the current slider positions. Slider setups can also be defined/edited by opening the box editor using the 'edit slider...' option from the box popup-menu item. The setups are stored as a list of lists of slider values.
In (2) the slider-bank responds to MIDI continuous control (CC) messages. This is reflected by using green color at the slider handles; also there is a small numbered green triangle at the low-left corner of the slider-bank box. The CC assignments are given by opening the box editor and entering a list of controller numbers (one number for each slider) using the MIDI CC input (in this case we use controllers 1, 2 and 3).

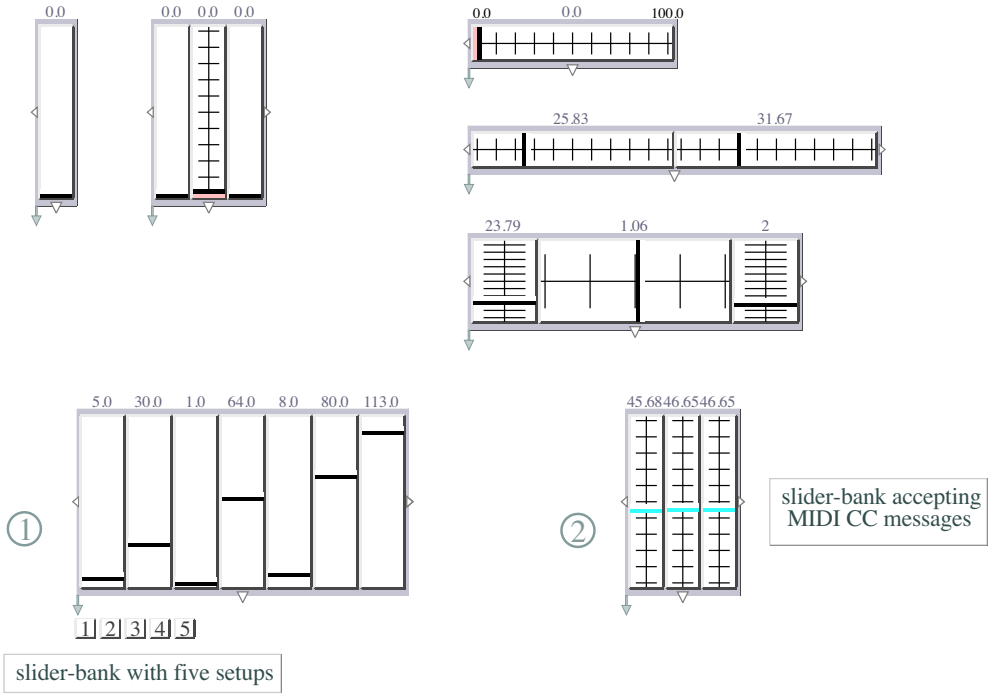

Figure 2.6: 05-sliderbank

2.1.7 Application-Input-Boxes

This patch shows five editor boxes (labeled with 'E'). Each of these boxes contain one application input-box. An application-input box contains a dedicated editor-window that can be opened by double-clicking the application input-box.
The input-box shows the contents of the editor-window. This view can be panned or zoomed locally using the scroll-wheel button of the mouse.

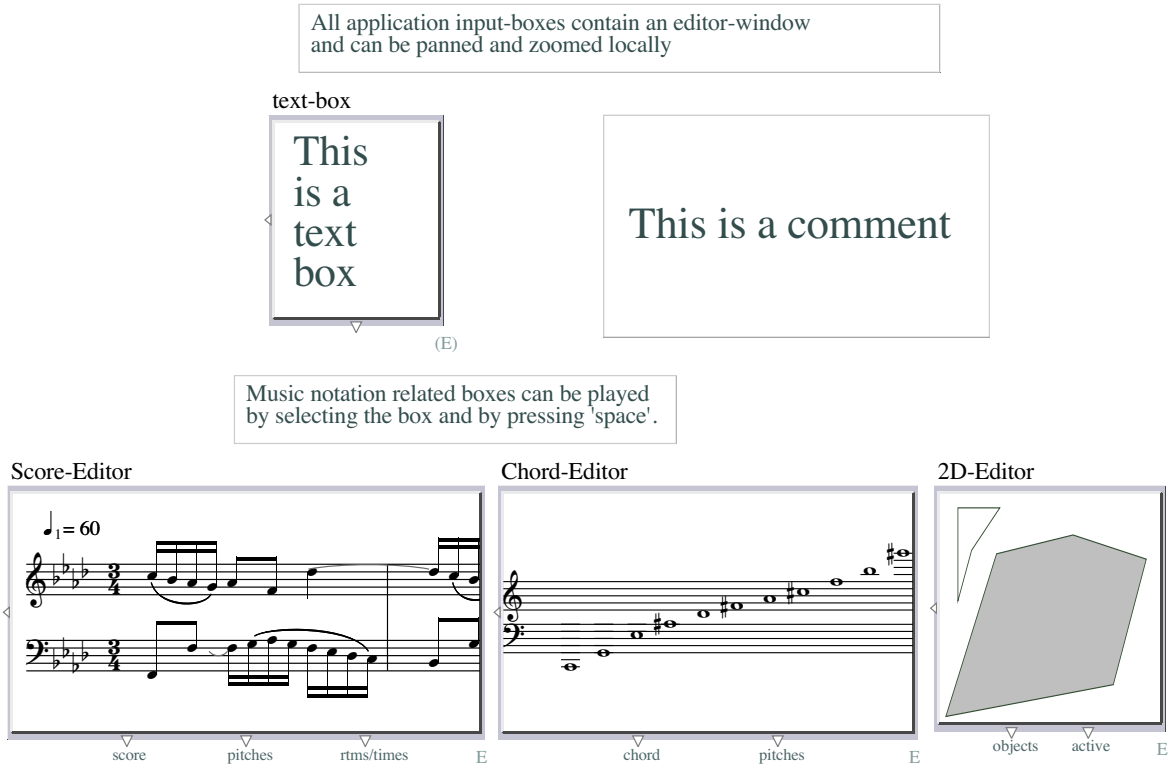

Figure 2.7: 06-application-input-boxes

2.1.8 Database-Input-Boxes

PWGL contains special extendable boxes that change their appearance and behaviour according to the first input-box of the main-box. By changing the state of the first input-box the box will change the number of inputs, input-types, extension patterns and default values.
This scheme allows to define within one box complex applications that can be used to build various structures, represent databases, etc.

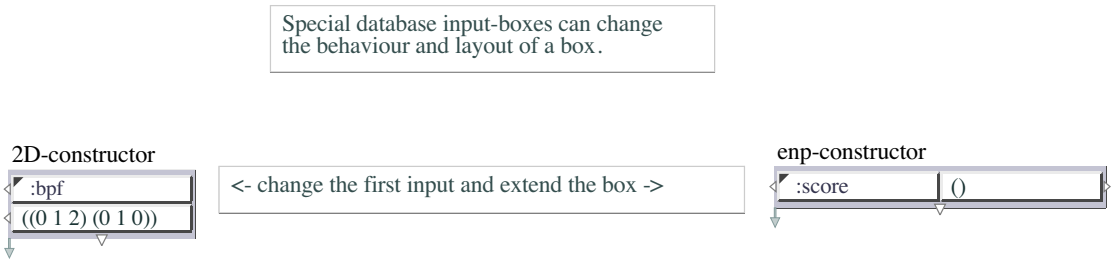

Figure 2.8: 07-database-input-boxes

2.1.9 Constructor-Boxes

This patch demonstrates two constructor boxes that are used to build objects for the 2D-Editor, Chord-Editor and Score-Editor.
The type of the object is defined by the first input-box. Changing the object type will change the layout, the input-box types and default values of the box (see also the previous tutorial).

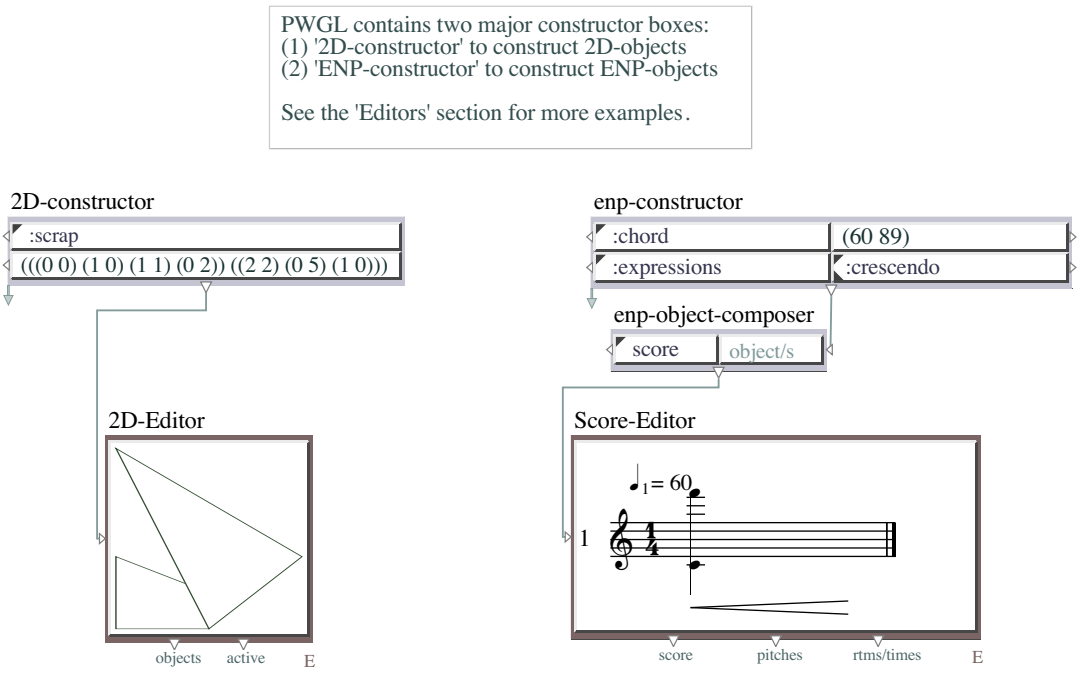

Figure 2.9: 08-constructor-boxes

2.1.10 Programming

PWGL contains three basic textual tools that allow Lisp programming: text-box, Lisp-code box, and code-box. The main differences between these boxes are the following.
The contents of a 'text-box' can be read into the patch (using the output), and thus the box is used besides code also for data and rules.
The contents of a 'Lisp'-code box cannot be accessed from a patch and thus it is used mainly for code. The contents is compiled automatically when the patch is loaded.
The 'code-box' allows the user to express in textual form complex Lisp expressions and it is one of the most important tools to interface Lisp with the graphical part of PWGL. The user can open a text-editor by double-clicking the box. In the text editor, while the user writes the code, the text is simultaneously analysed. The appearance of the box is calculated automatically based on this analysis. There is a special section dedicated to the code-box in this tutorial.
Note, that when working with code with the text-box or the Lisp-code box, the first line should always be a package declaration, for instance: '(in-package :cl-user)'.
Important: avoid using the system package (':ccl' or ':system') for your personal code, as this may cause name conflicts with the underlying PWGL system.

Note: the code-box is different as it works always in package ':ccl'.

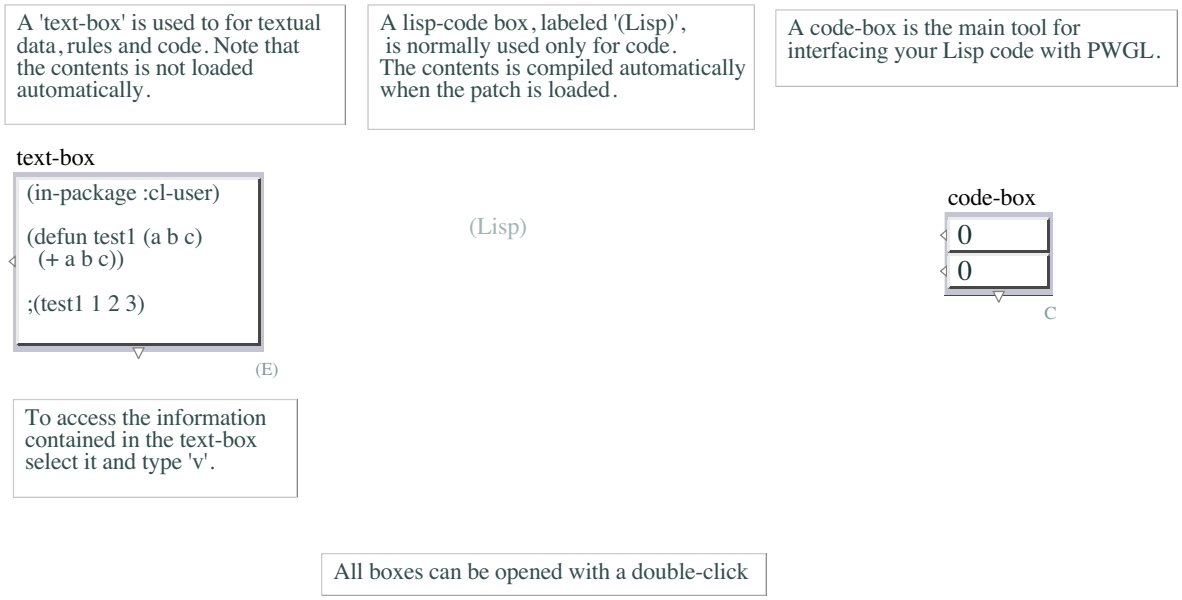

Figure 2.10: 09-programming

2.1.11 Box-Creation

This patch gives information of how to create your own boxes in PWGL.
The patch contains four text-boxes that contain box definitions with increasing complexity: (A) simple 'defun' case (B) 'PWGLDef' case (C) Extended case with layout options (D) Box definition, complex version
Open a text box and compile it. After this choose a PWGL window and enter the box names in the 'Lisp function' dialog.
Note: the paper 'icmc2003_box-design.pdf' dealing with PWGL-boxes is found in the 'Documentation' entry in the 'Overview' section of this tutorial.
The 'documentation/programming/box examples' folder gives some more complex examples that demonstrate how input boxes can interact within a PWGL-box.

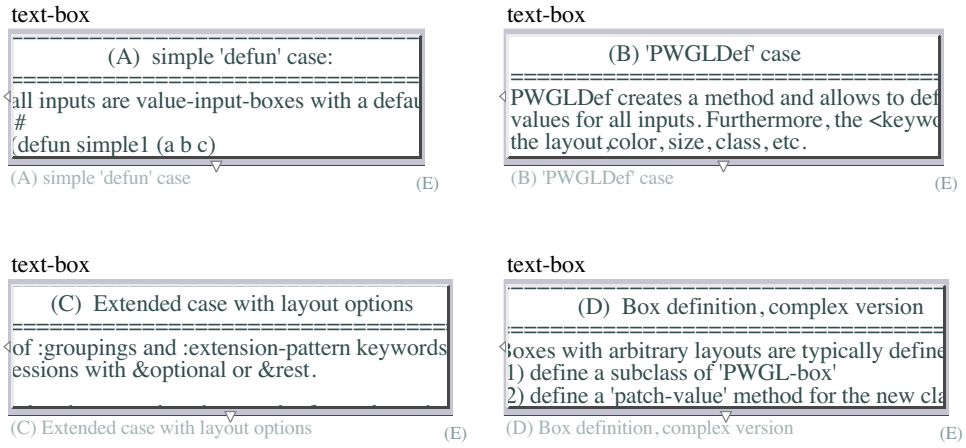

Figure 2.11: 10-box-creation

2.2 Control

2.2.1 PWGL-Map1

This patch and the companion patch called 'PWGL-map2' demonstrates how list handling loops can be realized in PWGL using the PWGL-map loop. A PWGL-map loop reminds somewhat the PW-map module of PW. PWGL-map has, however, been completely redesigned in order to facilitate loop handling routines in PWGL.
The PWGL-map loop consists of two boxes, PWGL-enum and PWGL-map, which are always used together. PWGL-enum is used to initialize the loop with initial lists to be treated by the loop. PWGL-enum is an extendible box: for each extended input there is also a corresponding output. PWGL-map, in turn, is used during the loop as a collector of the incoming patch evaluation at the second input 'patch'. PWGL-map has three optional arguments: 'test', 'accum' and 'endtest'. See the box documentation of PWGL map for further details.
This patch contains four basic examples: (1) a simple iteration of one list (2) an example with three parallel lists (3) the result list is filtered with the lisp function 'oddp' (4) a

loop where the result is not collected (i.e. this kind of loop is used only for a side effect, such as printing)

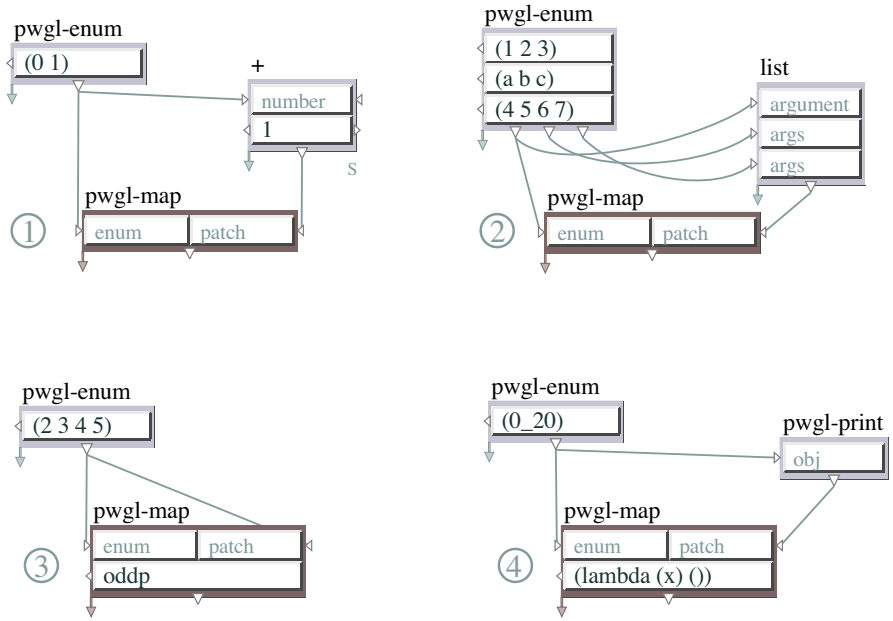

Figure 2.12: 01-PWGL-map1

2.2.2 PWGL-Map2

This patch demonstrates more PWGL-map loop examples. Each example uses the optional PWGL-map box arguments 'test', 'accum' and 'endtest'.
This patch contains five examples: (1) a simple iteration that collects all elements (2) an example that accepts only odd numbers and returns the sum of the result list (3) like example (2) but the minimum value of the result is returned (4) like example (2) but the maximum value of the result is returned (5) a loop that accepts only values that are not in the result list. The 'endtest' stops the loop if the result length exceeds 12.

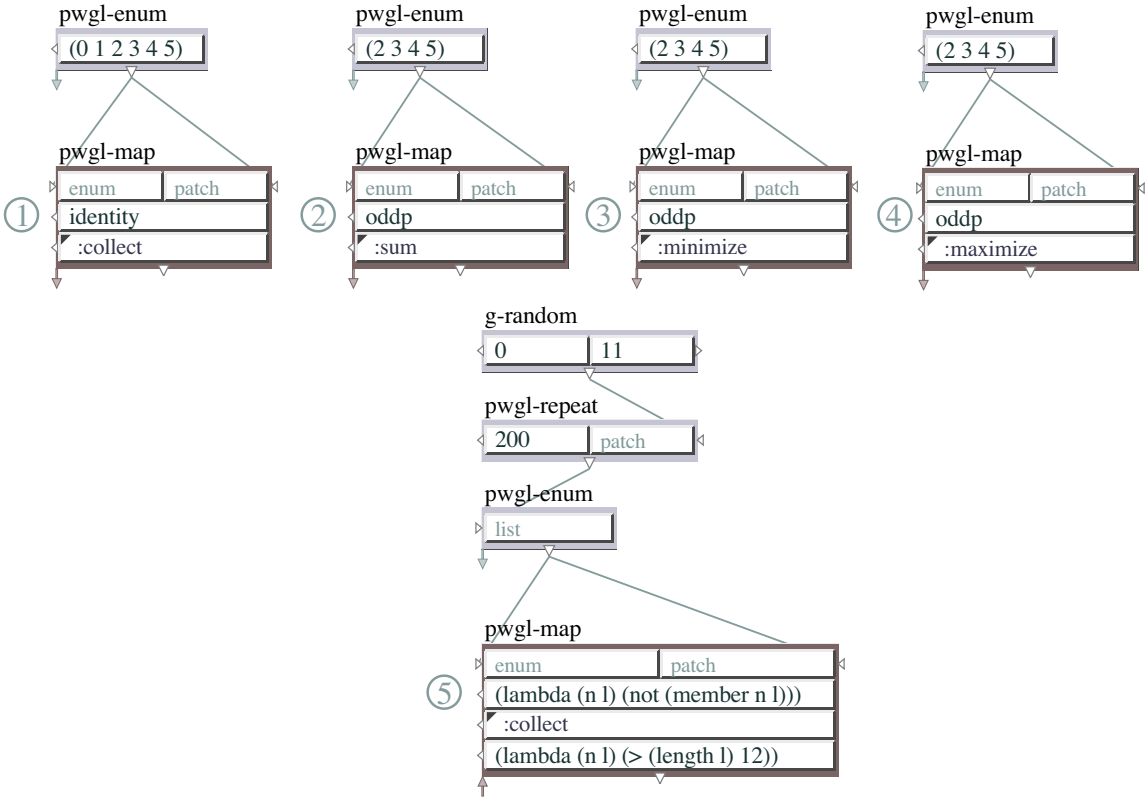

Figure 2.13: 02-PWGL-map2

2.2.3 Circ

This patch demonstrates some features of the 'PWGL-circ' box that is used for circular lists.
The 'reset' button allows to reset the circular list – i.e. the box will start with the first item of the list –(1). The list can also be reset automatically using the second input–with the option ':yes'–(2) and (3). In these cases the list will be reset each time the patch is evaluated.
(4) and (5) demonstrate how to use the PW 'expand-list' format for the 'clist' input.
In (6) and (7) the 'PWGL-circ' has been extended with extra 'clist' inputs. In these cases all given 'clists' are merged to one circular list of lists. (7) shows how these sublists can be circulated individually.

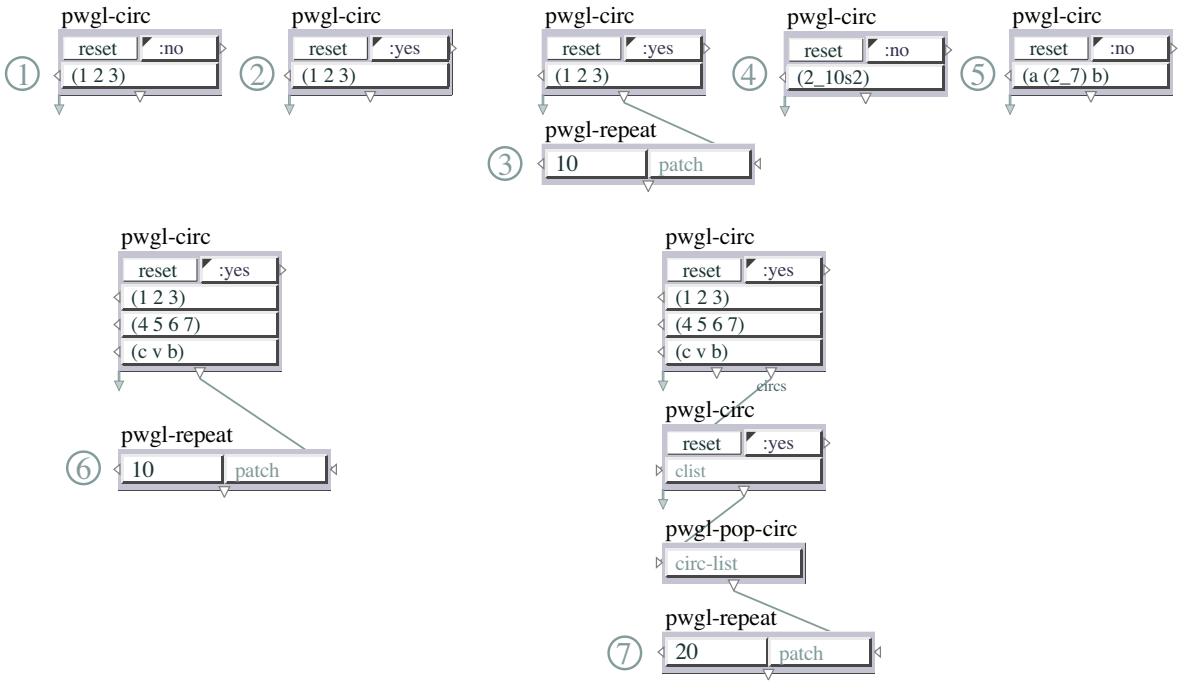

Figure 2.14: 03-circ

2.2.4 Switch

This patch gives some examples of how to use switch and merge boxes in PWGL.
(1) shows a simple PWGL-switch box where the incoming patch can be chosen by selecting one of the buttons of the PWGL-switch box. If no input is connected then only the index is returned.
(2) shows a PWGL-merge box where the incoming patches can be chosen by selecting any combination of buttons of the PWGL-merge box. This box returns all selected inputs as a list. If no input is connected then only the index is returned.
(3) shows a master switch box ('MSW') that has 2 simple switch boxes ('a' and 'b') as slaves. The box-string of the master switch (here 'sw') controls all simple switch boxes with the same box-string.

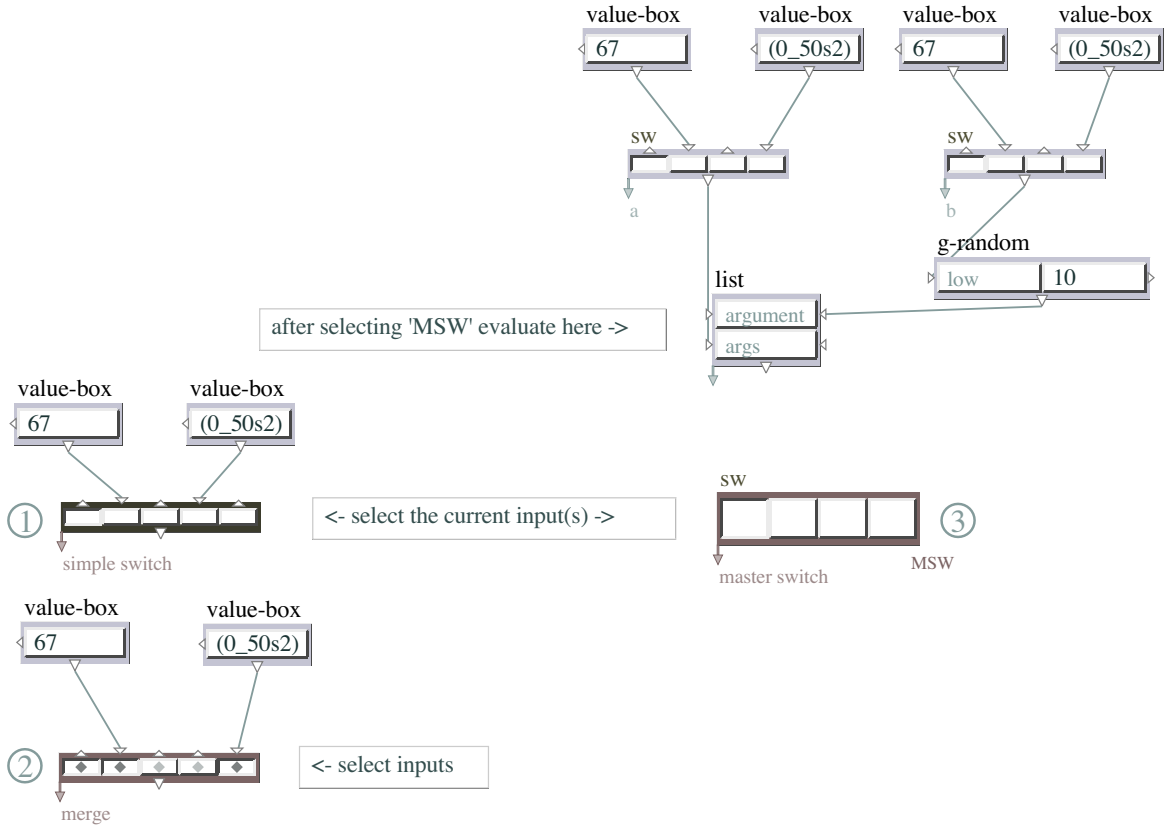

Figure 2.15: 04-switch

2.2.5 Const-Value

This patch gives some 'const-value' box examples, where the idea is to keep some data constant even when several boxes are connected to an output of a box (in normal situations this would cause multiple evaluations of the latter box).
(1) a simple case where the random data is kept constant each time the 'const-value' box is evaluated.
(2) and (3) demonstrate different behavior in a loop context using the: ':once', ':loop-init', and ':eachtime' options of the second optional input.

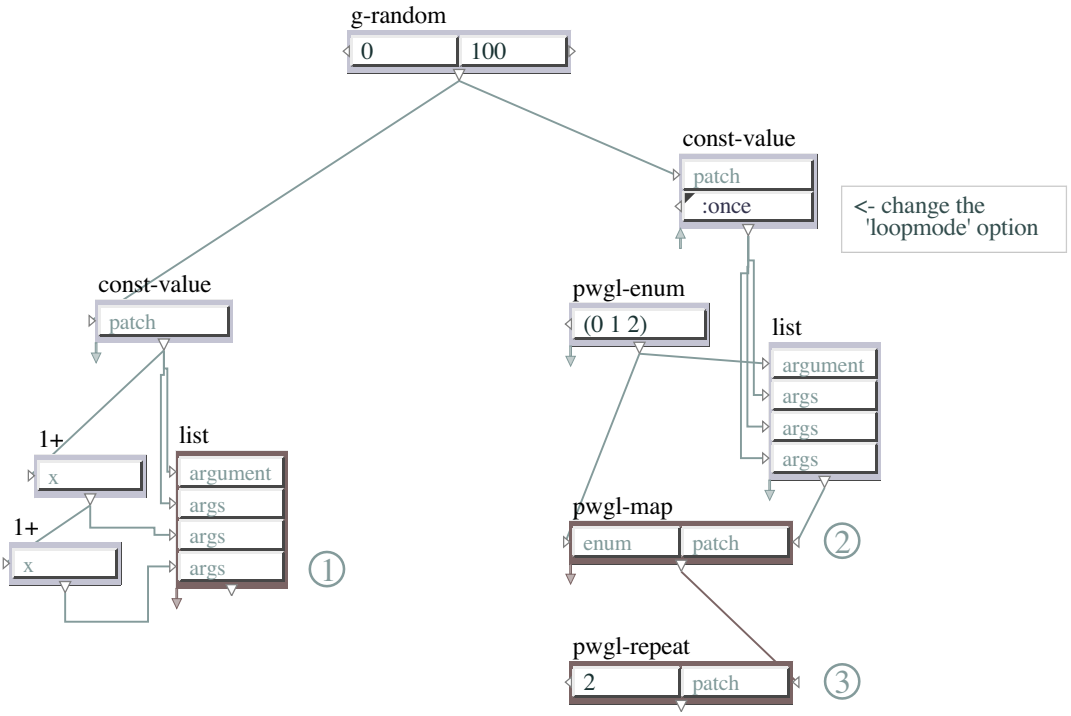

Figure 2.16: 05-const-value

2.2.6 PWGL-Value1

'pwgl-value' allows to use pseudo-local variables or functions in PWGL. The values are stored in a hash table and they can be accessed anywhere in a patch. Note that the hash table is cleared with every top-level patch evaluation.
The 'value-key' parameter is a keyword. If the 'init' or the 'write' input is not given then the value stored under 'value-key' is returned.
An initial value can be stored under 'value-key' using the optional argument 'init'. After this the value can be accessed by other 'pwgl-value' boxes or from textual code in a patch (for instance in scripting or constraints rules). If the initial value needs to be updated after the initialization then use the 'write' argument.
(1) a basic example where a random list is kept static during several 'pwgl-repeat' calls.
(2) shows how initial values defined 'pwgl-value' can be referred to inside an abstraction that is in 'lambda' mode.
In (3) 'pwgl-value' is used to initialize a lambda expression, which acts later in the patch as a Lisp closure.

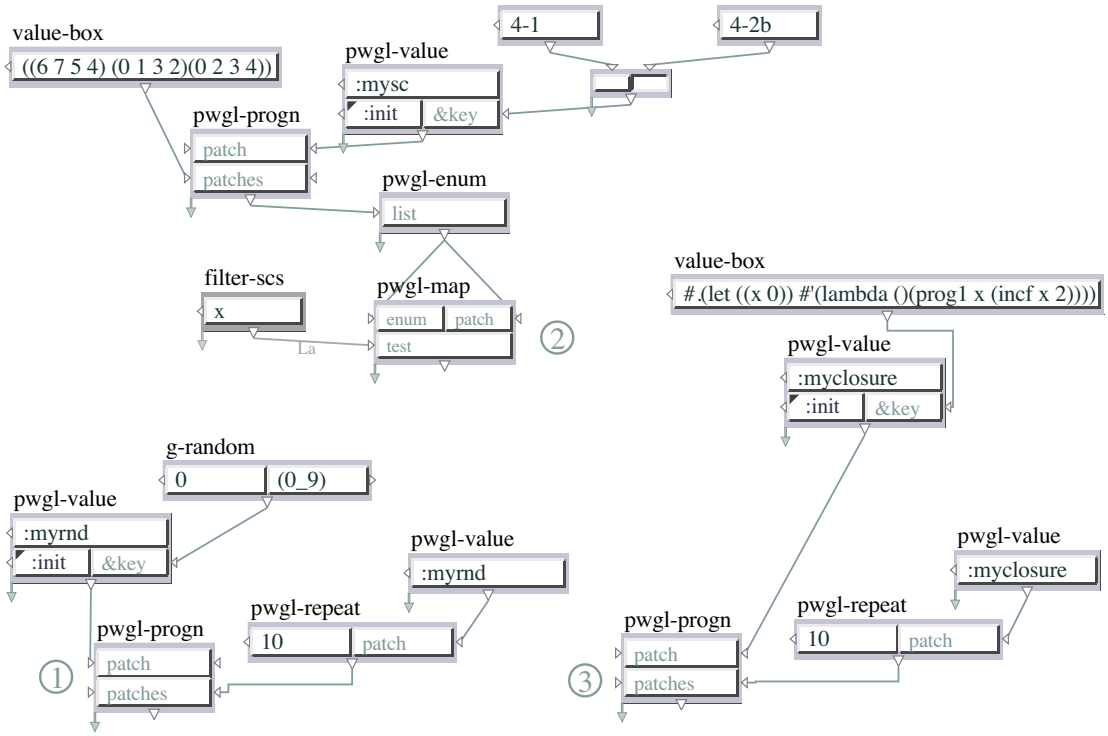

Figure 2.17: 06-PWGL-value1

2.2.7 PWGL-Value2

This more complex example demonstrates how the PWGL-value box can be used to initialize, read and write information in a loop.
An initial list, '(4 0 5 2 3 1)' , is gradually transformed to another list, '(6 7 9 10 11)', using the PWGL-value and the PWGL-map loop boxes.
The initial list is created and stored under the keyword 'subst-list'; (1); within the loop the current state of this list read (2); and after the substitute operation the modified list is written back under 'subst-list' (3). Thus 'subst-list' is modified at each loop step.

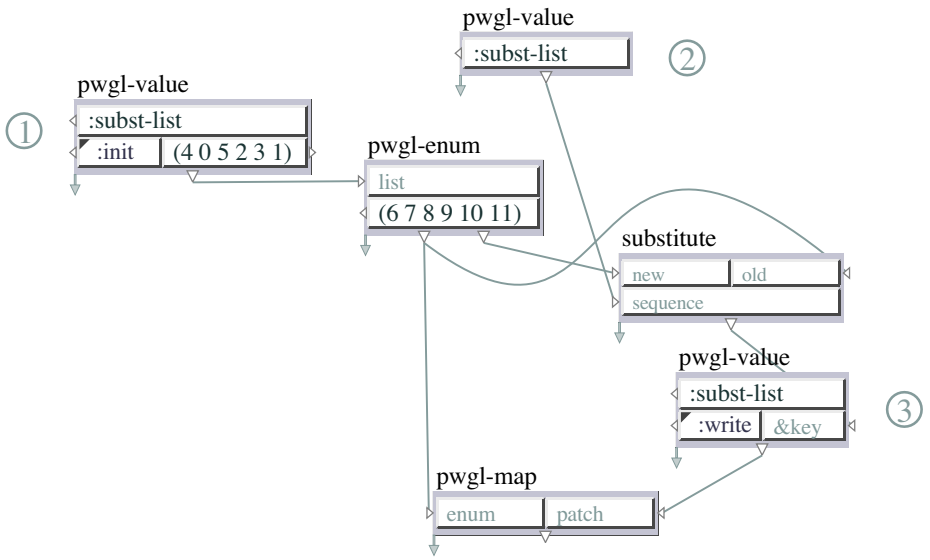

Figure 2.18: 07-PWGL-value2

2.2.8 PWGL-Value3

Here we use PWGL-value to calculate an interval distribution for a chord (1).
In (2) we store the input chord under ':init'. Then we pass this list (except the last element) to a pwgl-map loop. Inside the loop we read again our ':init' list (3), but we also remove each time the first element of this list (note that we use here the ':write' operation). Thus the ':init' list becomes shorter at each iteration step.
Finally, in (4), we calculate and sort the final interval distribution.
In (5) we use the code-box to define the left-part visual patch definition in textual form.

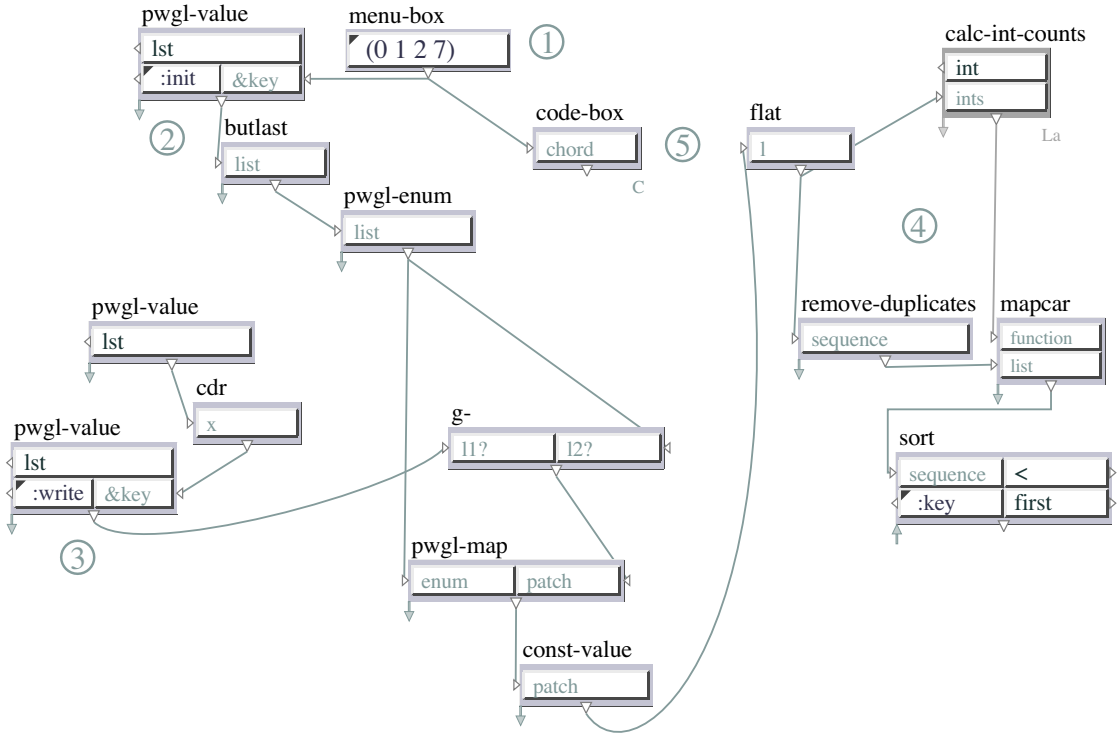

Figure 2.19: 08-PWGL-value3

2.2.9 Reduce-Accum

This patch presents the standard 'reduce' Common Lisp function (equivalent to the OpenMusic 'accum' function).
This tutorial is roughly based on the on-line OpenMusic 'accum' tutorial, but here we use 'reduce' instead of 'accum'.
In (1), (2) and (3) we use 'reduce' in conjunction with simple lisp functions (list, + and *).
In (4) and (5) we utilize the PWGL abstraction scheme (note that the abstractions are in 'lambda' mode) to define the functions for the first input visually.

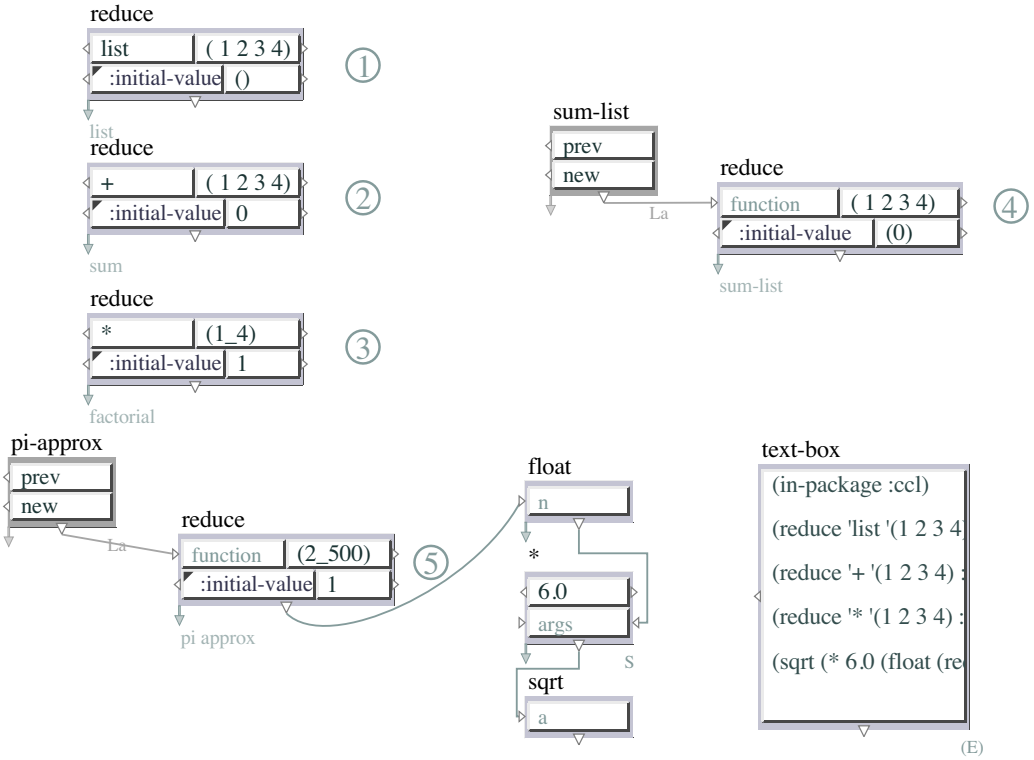

Figure 2.20: 10-reduce-accum

2.3 Editors

2.3.1 Introduction

The 'Editors' chapter of the tutorial consists of five subsections. The first three subsections are dedicated to each main editor in PWGL, i.e. 2D-Editor, Chord-Editor and Score-Editor. The last subsections deal with our scripting language that allow to make various side-effects to a score. We also have a subsection that demonstrates some ways how to create beat objects in PWGL.

2.3.2 2D

2.3.2.1 Spiral

This is a basic patch that demonstrates how a '2D-constructor' box can be used to create a bpf (break-point function) object.
The patch uses two 'sample-fun' boxes that calculate the x- and y-coordinates. In order to get the final spiral result, we scale the y-coordinates with the help of a 'interpolation'

box. The final result can be seen in the ’2D-Editor’.

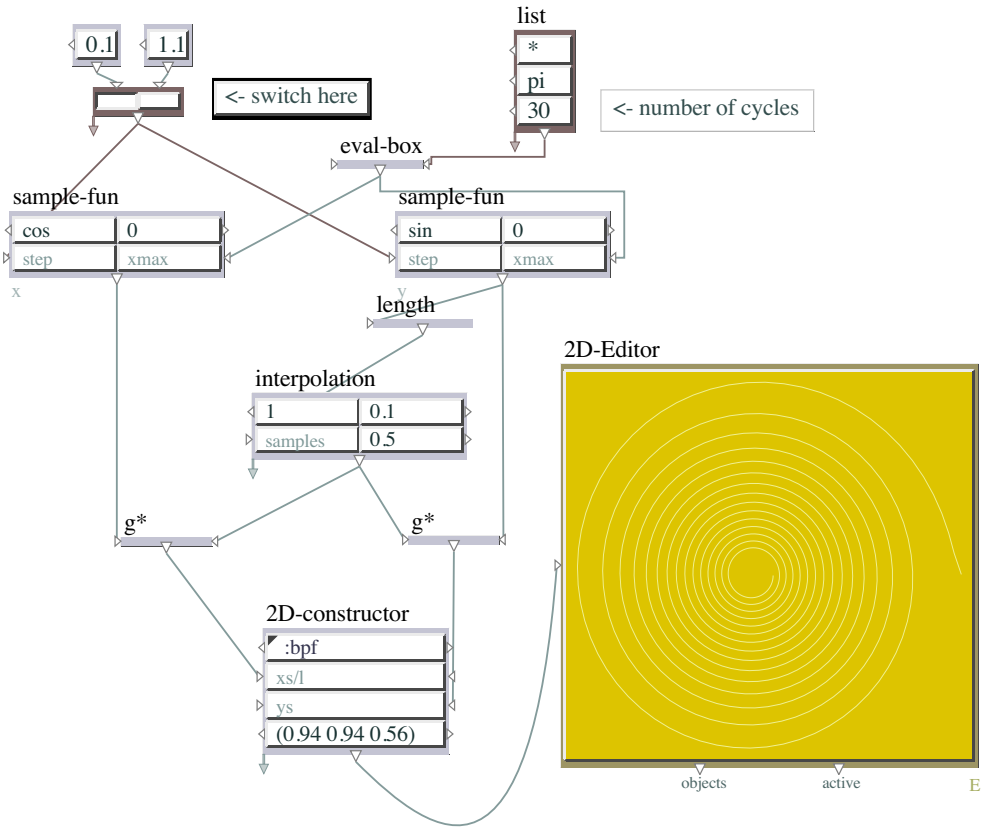

Figure 2.21: 01-spiral

2.3.2.2 Interpol-Bpfs

In this patch we generate 50 bpfs using a interpolation process. The starting point is always a sine function. The end point for the interpolation can be specified with a switch box that has five options: ’sin’, ’cos’, ’log’, ’tan’ and ’random’.
Also the colors for the resulting bpfs are calculated with a ’interpolation’ box. The red portion of the RGB values are gradually changed from 0.4 to 1.0.

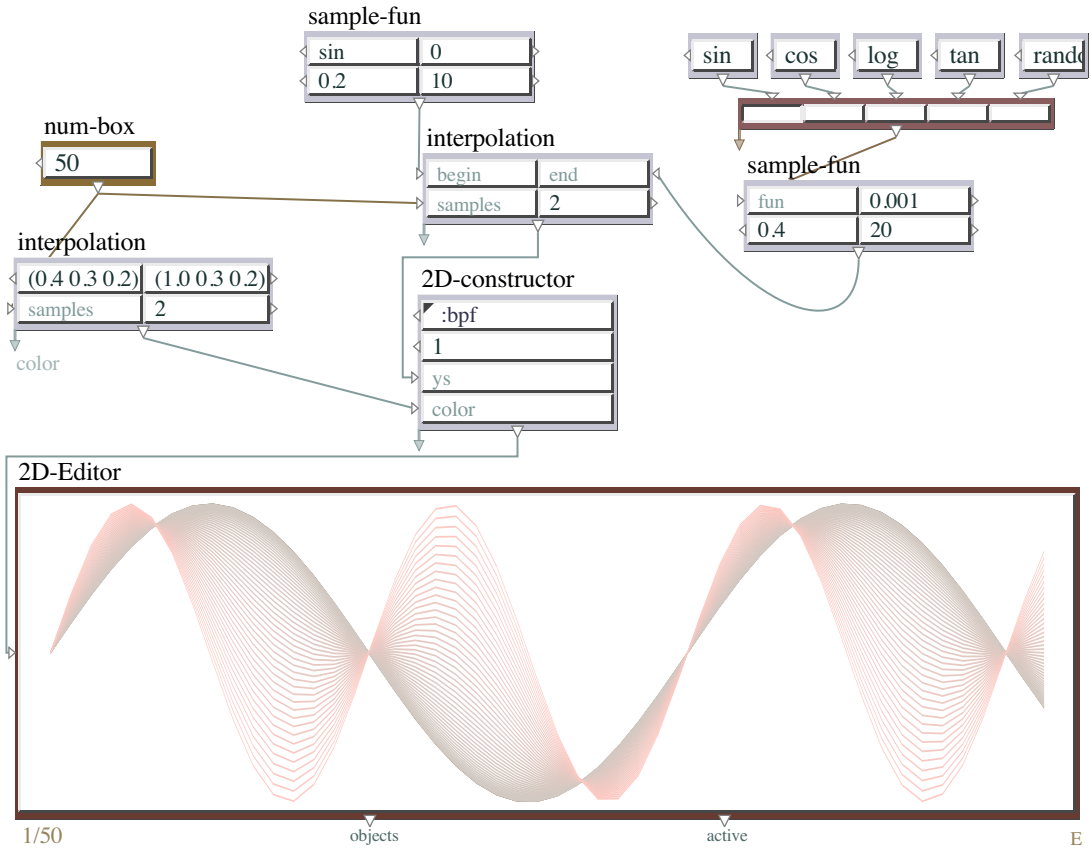

Figure 2.22: 02-interpol-BPFs

2.3.2.3 Bezier

A '2D-constructor' box is used here to generate 50 bezier functions (see the first input that is ':bezier'). Also we use several interpolation boxes to achieve the final result which is given in the '2D-Editor'.

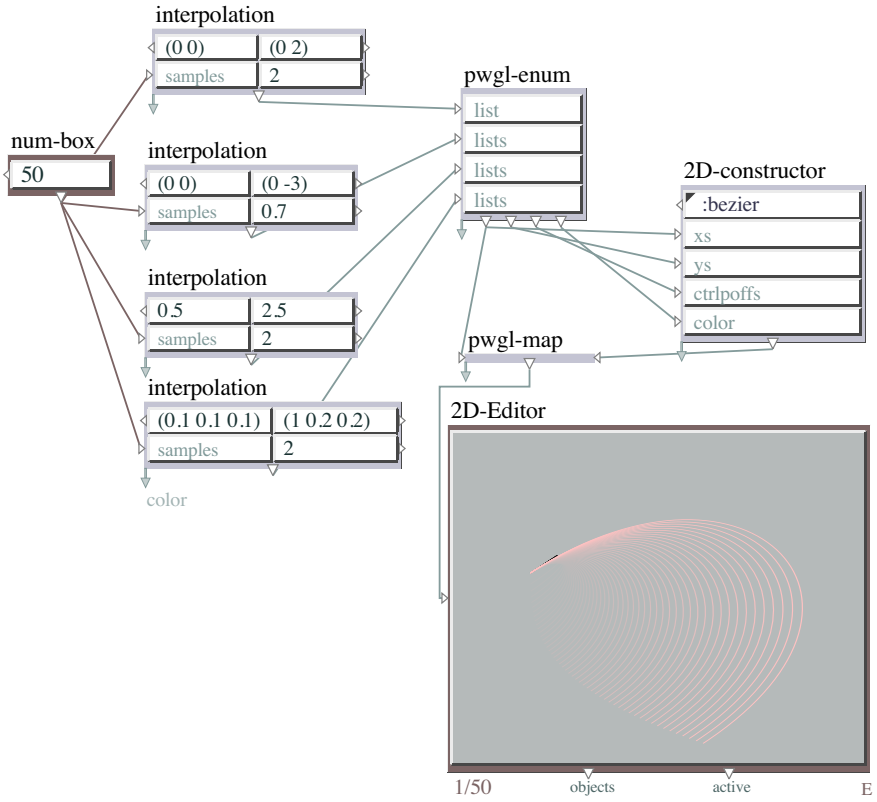

Figure 2.23: 03-bezier

2.3.2.4 Bezier-to-BPF

This patch is similar to the previous ones: we generate 50 bpfs. The difference is however that the starting point and end point of the interpolation process are given as two bezier functions that are found in the upper part of the patch. The conversion (from bezier to bpf) and the interpolation are found in the abstraction 'interpol-bps'.

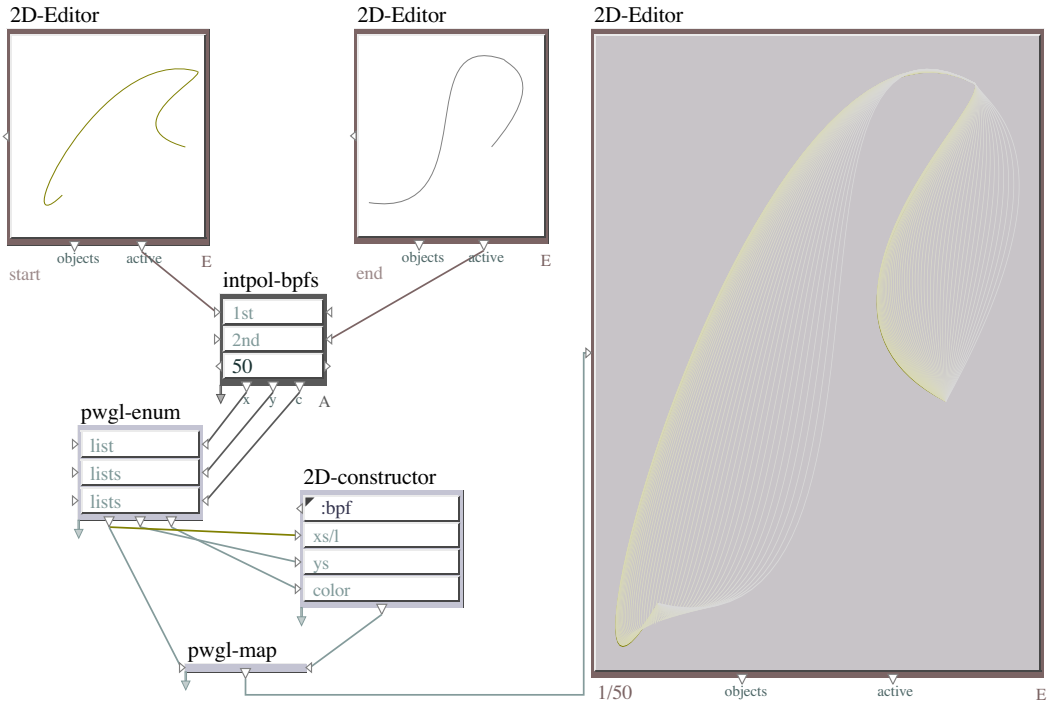

Figure 2.24: 04-bezier-to-bpf

2.3.2.5 2D-Constructor

This patch contains several 2D-object types situated in '2D-Editor' boxes (upper row). The x and y values are accessed using the optional 'x' and 'y' outputs. These outputs can be added to a '2D-Editor' box using the box-editor (the box-editor opens either with the 'edit box...' box popup menu-item or by a double click on the main box area).
The x and y values can be manipulated and fed to '2D-constructor' boxes (middle part of the patch). The lowest row contains '2D-Editor' boxes that show the final results.

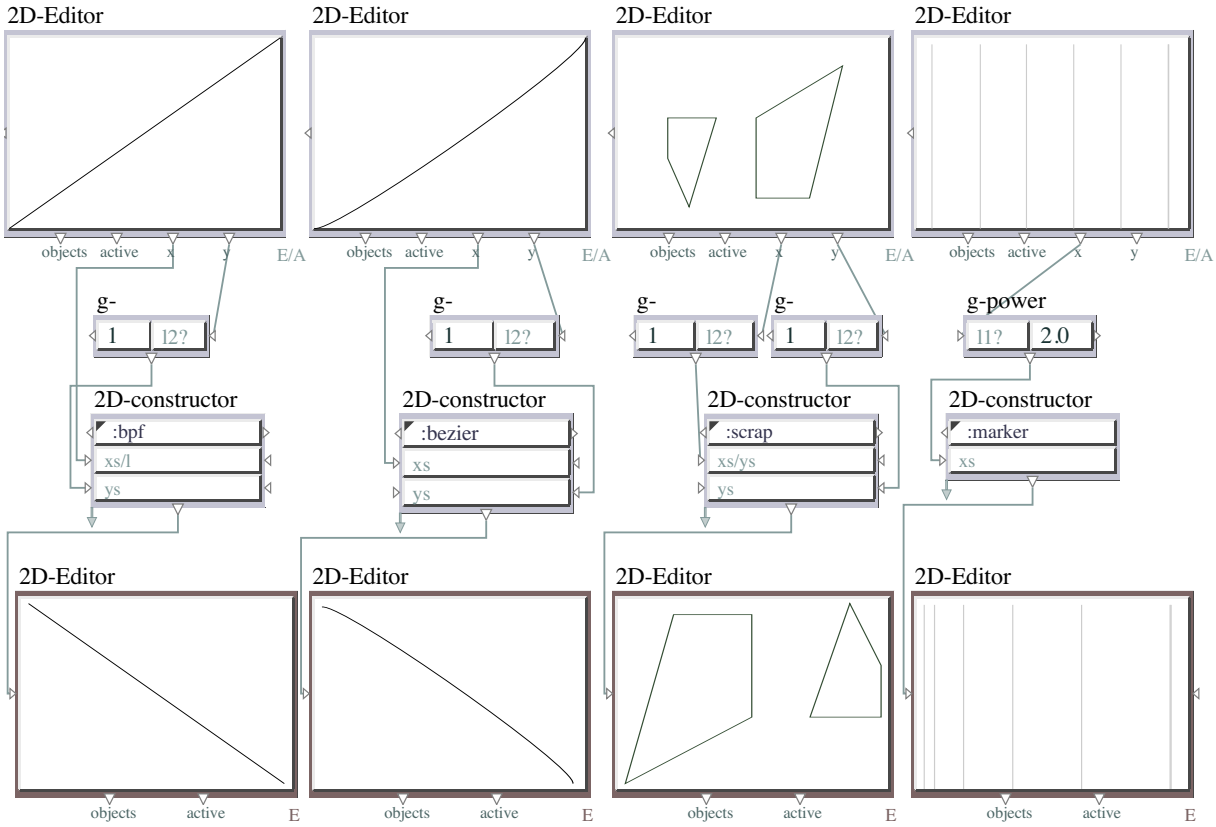

Figure 2.25: 05-2D-constructor

2.3.2.6 2D-Chord-Seq

This patch shows how music notation related objects can be fed to a 2D-Editor. We have here four input options to the '2D-Editor' box that can chosen with a switch box '(1) (2) (3) (4)': (1) a metric 2-part score; (2) a non-mensural score; 3) a chord; (4) a list chords (here the chords are calculated algorithmically in the abstraction 'gen-chords').

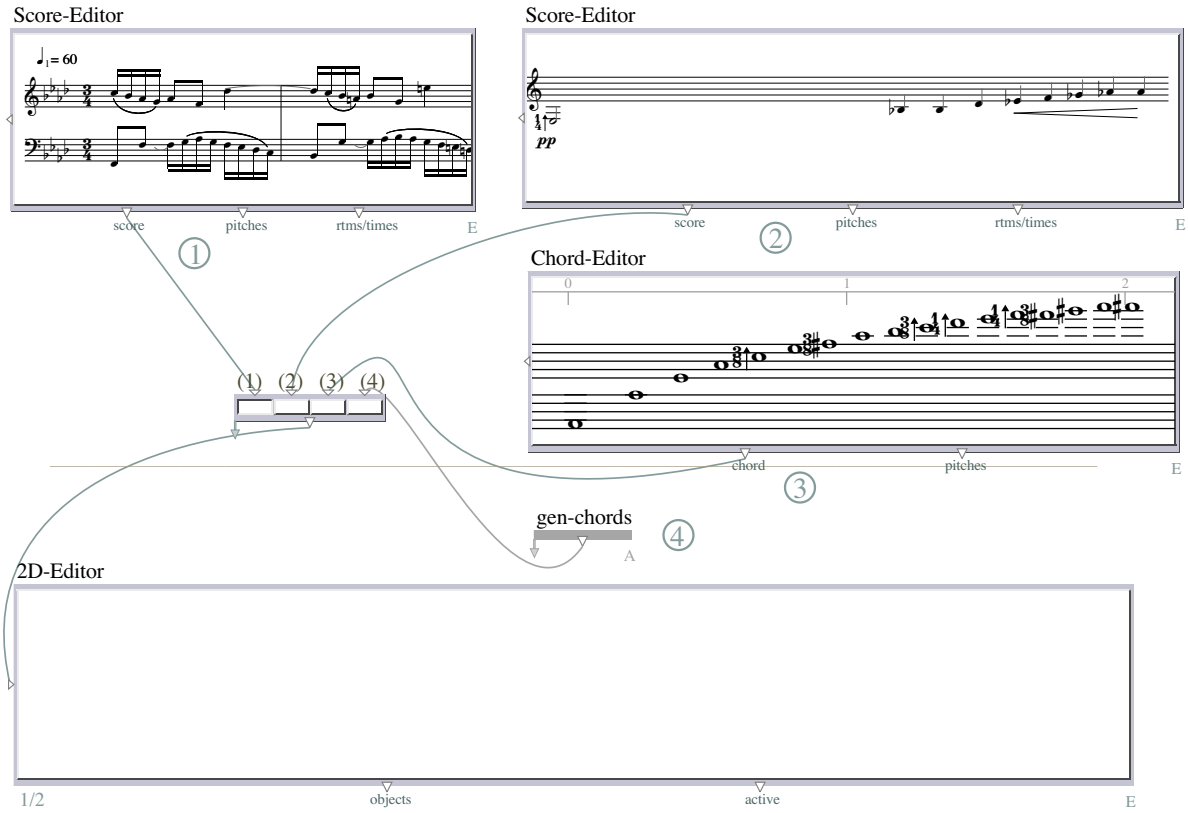

Figure 2.26: 06-2D-chord-seq

2.3.2.7 PWGL-Sample

This patch demonstrates the 'pwgl-sample' box which can be used to sample following 2D-objects: bpf, bezier, sound-sample and scrap collection.
'pwgl-sample' first calculates an internal sampling interval according to the min and max x-values and the 'no-of-points' argument of the 2D-object in question. After this a train of sampling pulses are generated and the respective y-values are read at each pulse (x) value.
The upper '2D-Editor' contains three 2D-objects: a sound sample, a bpf, and a bezier. The current 2D-object can be selected using the master switch box 'samp/bpf/bez'. The result of the sampling process is a bpf that is shown in the lower '2D-Editor' box.

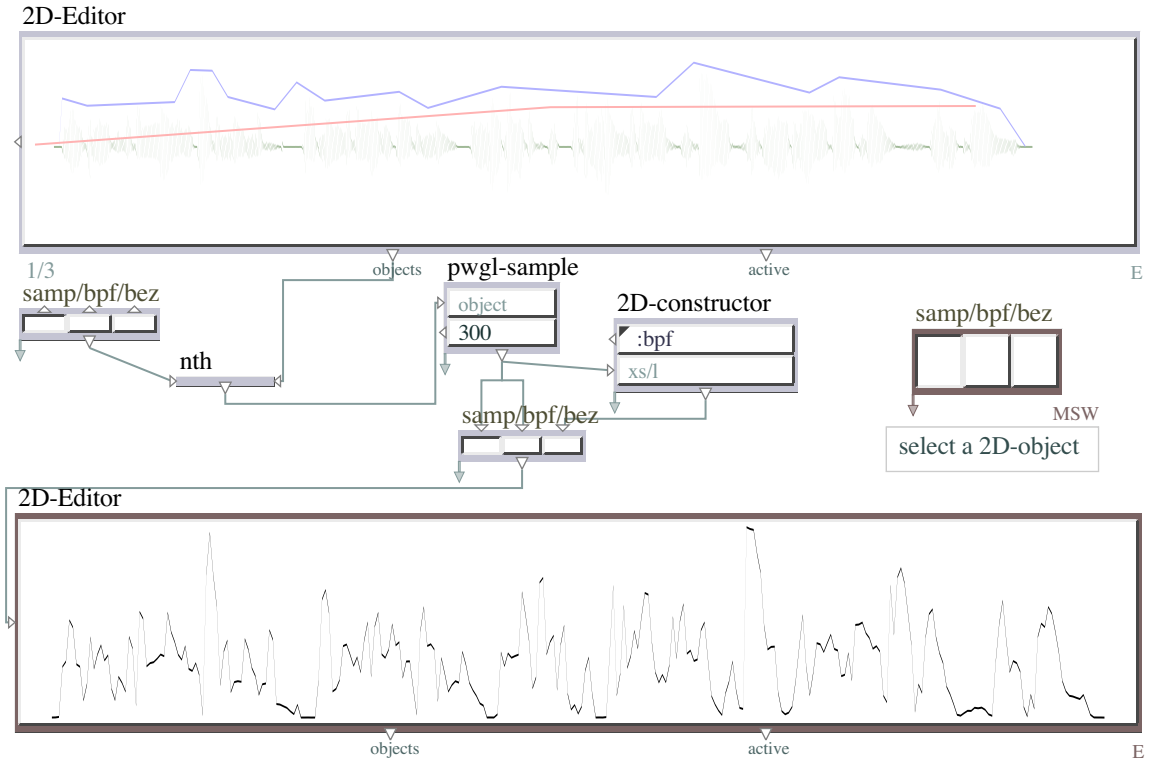

Figure 2.27: 07-pwgl-sample

2.3.2.8 BPF-Arithmetic

This patch shows how some generic arithmetic boxes ('g+', 'g-', 'g*', etc.) are able to work also with bpfs. The first argument is here always a single bpf (sine function) and the second argument is a list of bpfs (sine functions). The user can choose different options with a switch box. The result is shown in lower '2D-Editor' box.

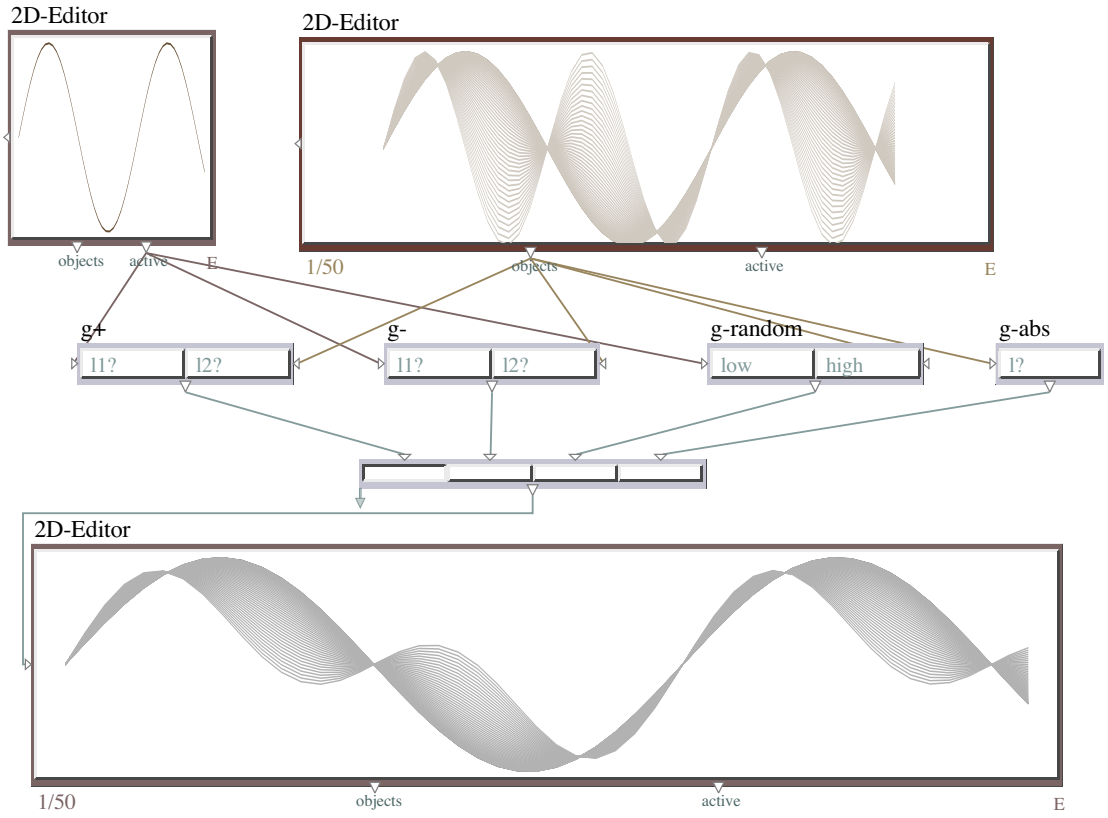

Figure 2.28: 08-BPF-arithmetic

2.3.2.9 Marker

Here the starting point is a complex bpf that is found in the upper '2D-Editor' box. In the 'filter-points' abstraction we choose points from the original bpf according to some criteria (here we use a filter that accepts only points that have a y-value that is greater than the 'limit' input). The x-values of all selected points are used to generate markers (see the '2D-constructor' box where the first input is ':marker'). Both the resulting marker-collection and the original bpf are fed to the lower '2D-Editor' box.
Note that the 'limit' input will change continuously at each evaluation due to the 'pwgl-circ' box (this will result in a more sparse marker-collection).

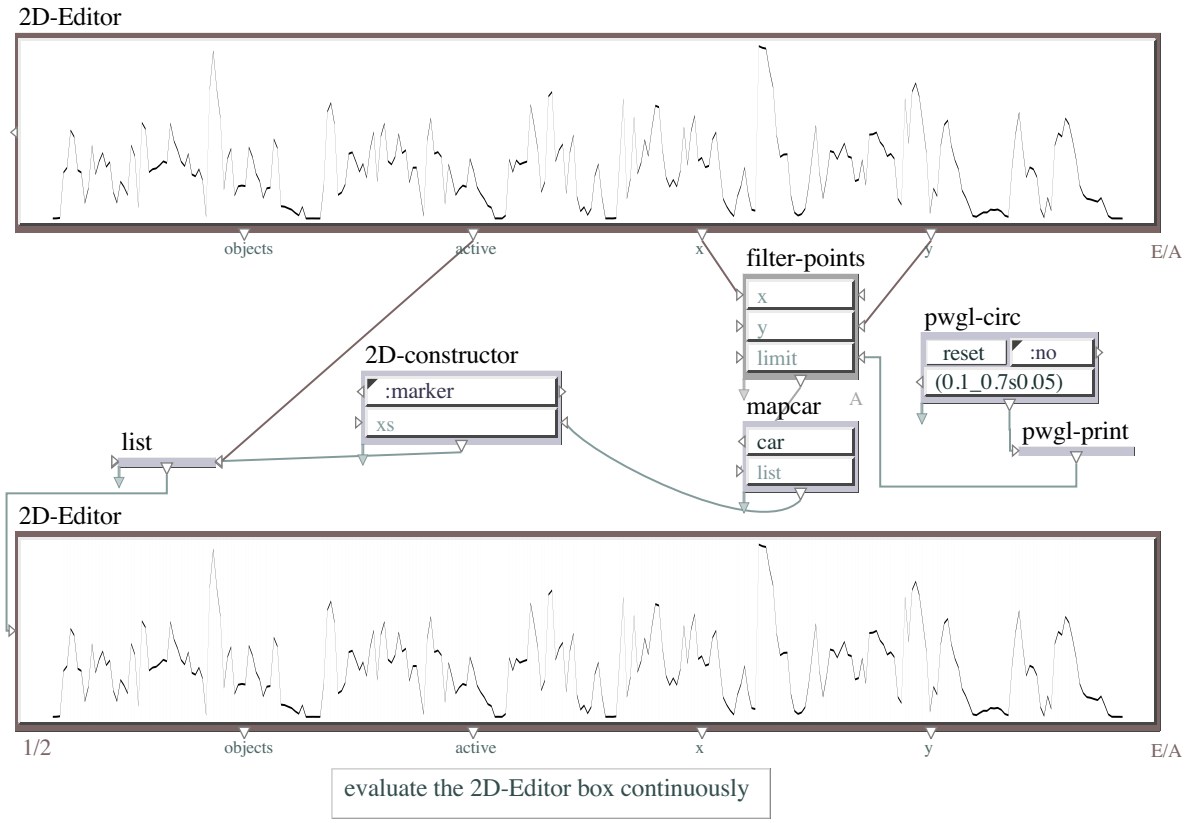

Figure 2.29: 10-marker

2.3.3 Chord-Editor

2.3.3.1 Overtone-Arp

This patch contains three options to create overtones (see the three 'f->m' boxes): (1) harmonic series; (2) compressed series; (3) stretched series. The result is converted to midi-values with an 1/8 tone approximation). After this the midi list is looped and at each iteration step a note object is created with a pitch and offset-time value (the latter parameter comes from an 'interpolation' box which generates an accelerando gesture). Finally the resulting list of notes is given to the 'enp-object-composer' box that creates a chord that is shown in the 'Chord-Editor' box.

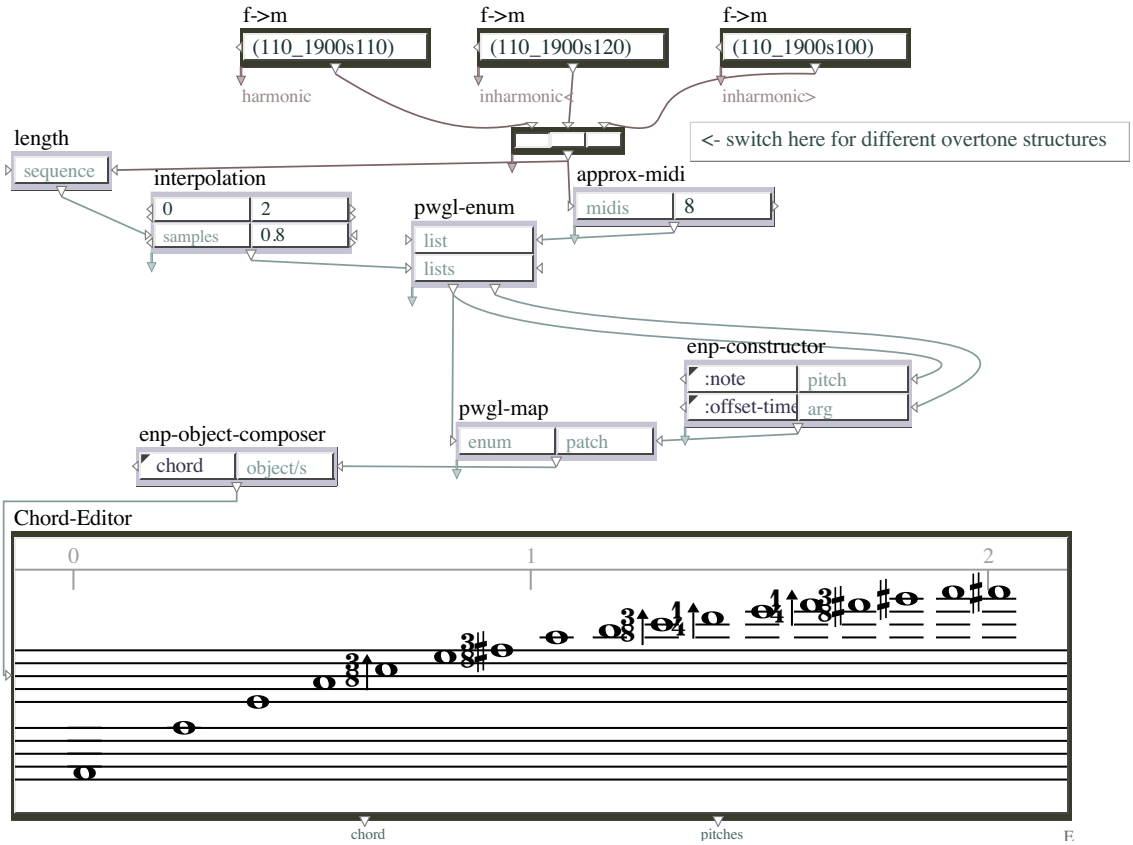

Figure 2.30: 01-overtone-arp

2.3.3.2 Chord-Matrix

A Chord-Editor can contain also several chords. In this example a list of four midi-value lists (see the 'text-box') is given to the 'Chord-Editor' box. This creates a 1*4 chord matrix where chords can be inspected, played and edited in the 'Chord-Editor' .

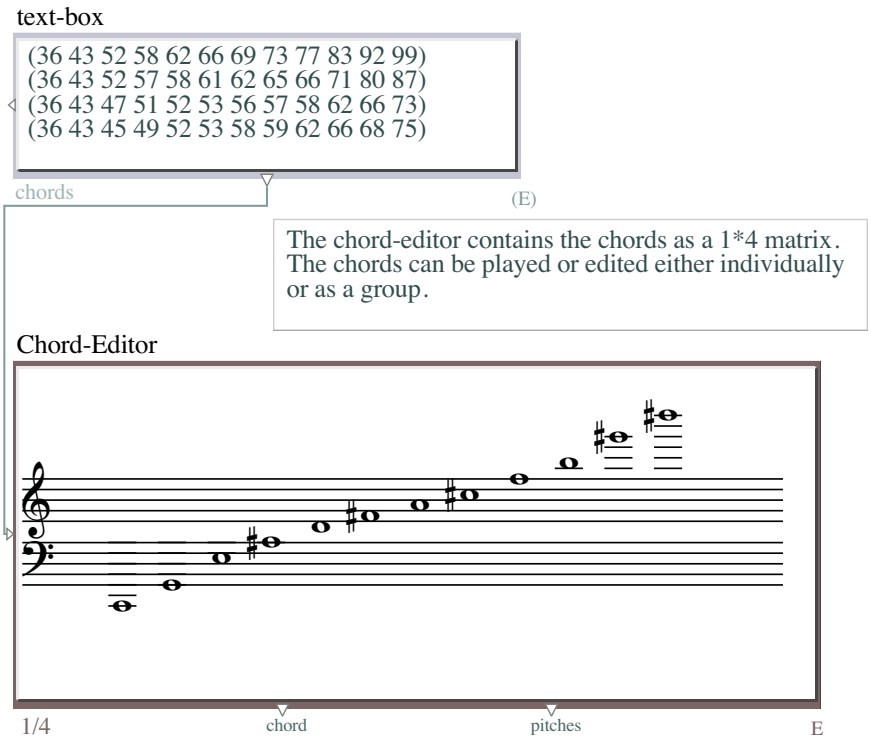

Figure 2.31: 02-chord-matrix

2.3.3.3 Circ-Chords

This example is similar to the previous one except that we have inserted here a 'pwgl-circ ' box before the 'Chord-Editor' box. The user can inspect one by one the chords in a circular fashion by re-evaluating the 'Chord-Editor' box.

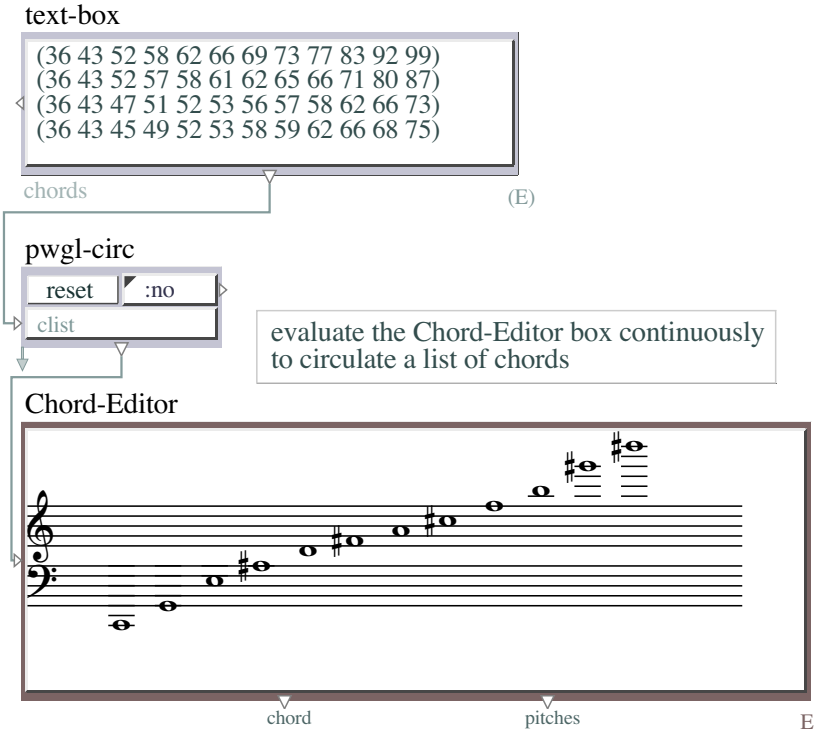

Figure 2.32: 03-circ-chords

2.3.3.4 Constructing-ENP-Objects-1

Here we demonstrate the use of the 'enp-constructor' -box. The first three constructor boxes are used to create notes with specific attributes (enharmonic spelling). The last one is, in turn, used to create a chord containing the notes created before. Furthermore, in this box, an accent expression is assigned to the chord.

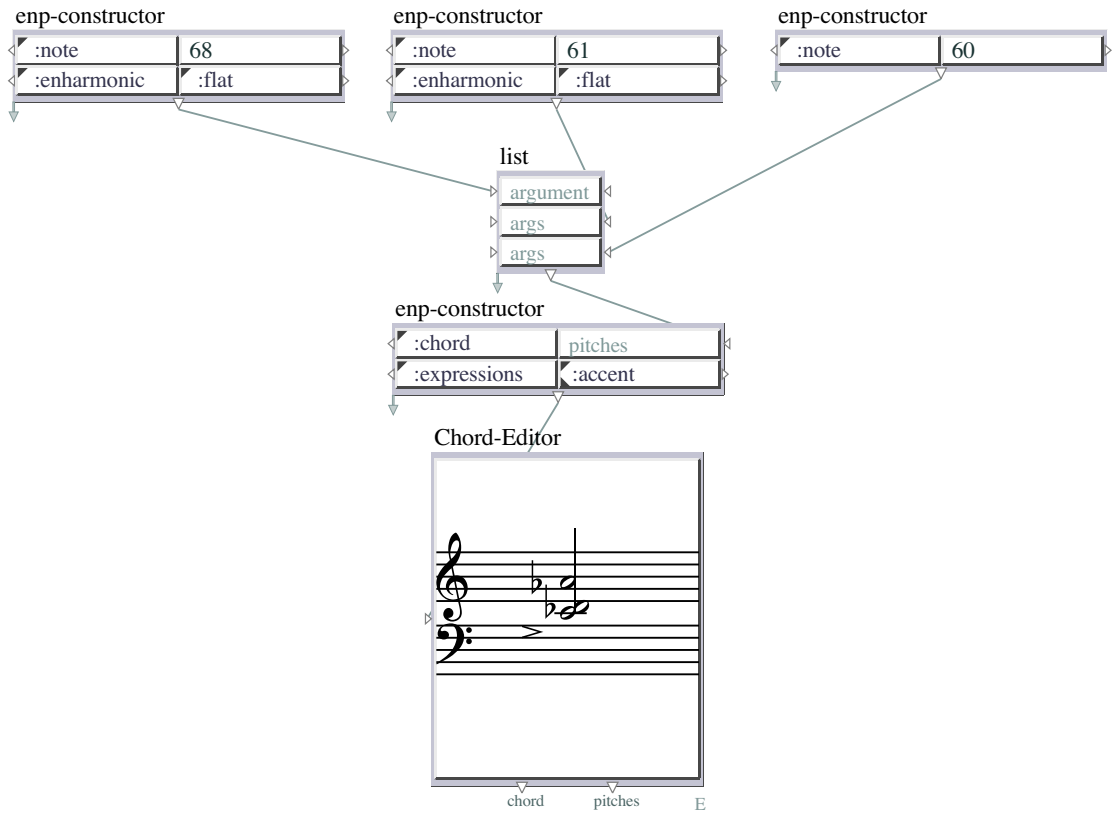

Figure 2.33: 09-constructing-enp-objects-1

2.3.4 Score-Editor

2.3.4.1 Transpose-Chords

A PWGL patch that generates chord sequences. The pitches are calculated by combining two chords ('chord1' and 'chord2'). The second chord is kept untransposed while the first one is transposed with intervals ranging from 0 to 12. The user can choose (by selecting one of the options of the master switch box called 'a tempo/acc') whether the resulting chord sequence will have static delta-time values or whether the sequence forms an accelerando gesture. The non-mensural result is shown in the 'Score-Editor' box.

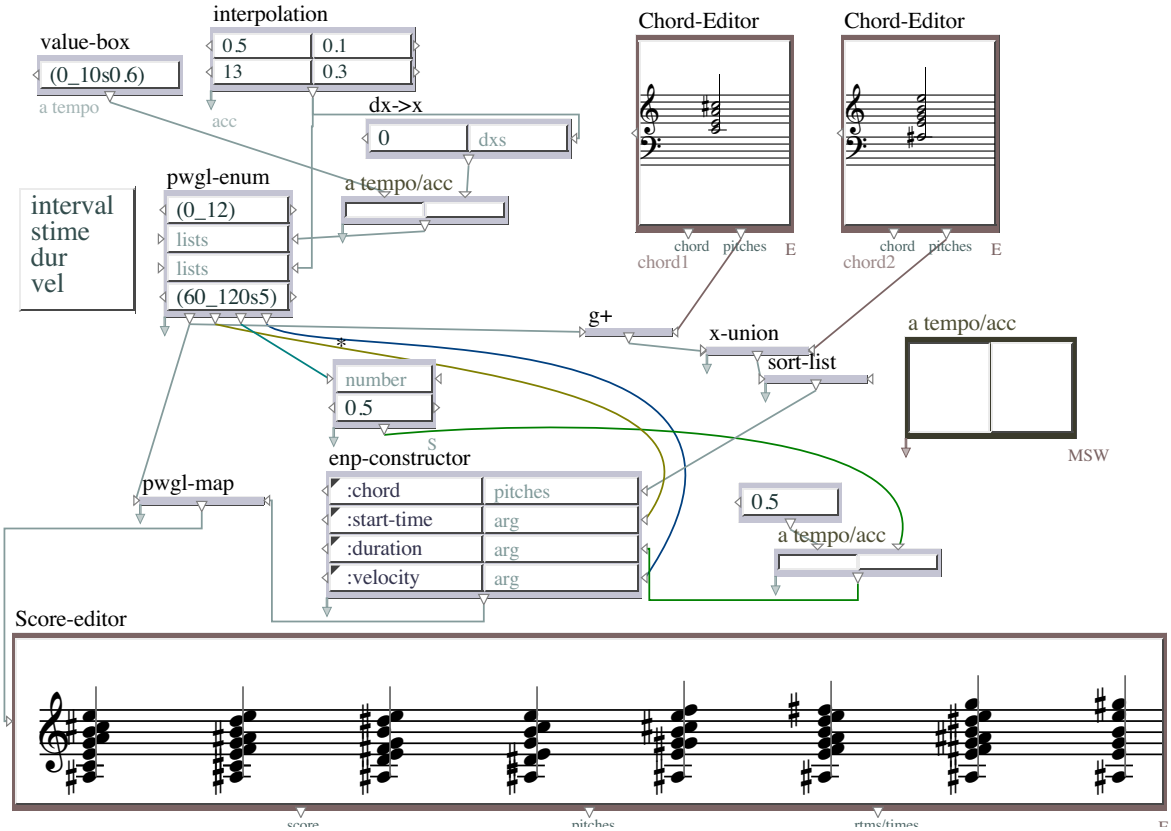

Figure 2.34: 01-transpose-chords

2.3.4.2 ENP-Constructor

ENP-score-notation is a score format to describe in textual form various score elements like beats, measures, voices and parts. This basic format can be extended using keywords to represent other score attributes such as pitch, enharmonics, expressions, and so on, resulting in complete scores.
Our example is based on the following two-measure rhythmic structure, see the 'text-box' (1): (((((2 (1)) (1 (1.0 1 1 1)) (1 (1 1 1 1 1)) (1 (1 1 6)))
((1 (1.0 -1)) (2 (-1 1 1 1 1 1)) (1 (1 1)) (1 (1)))))) that is extended with keyword/value pairs to produce the final score shown in the 'Score-Editor' (3).
The ENP-score-notation format is converted to a score using the box 'enp-constructor' (2).

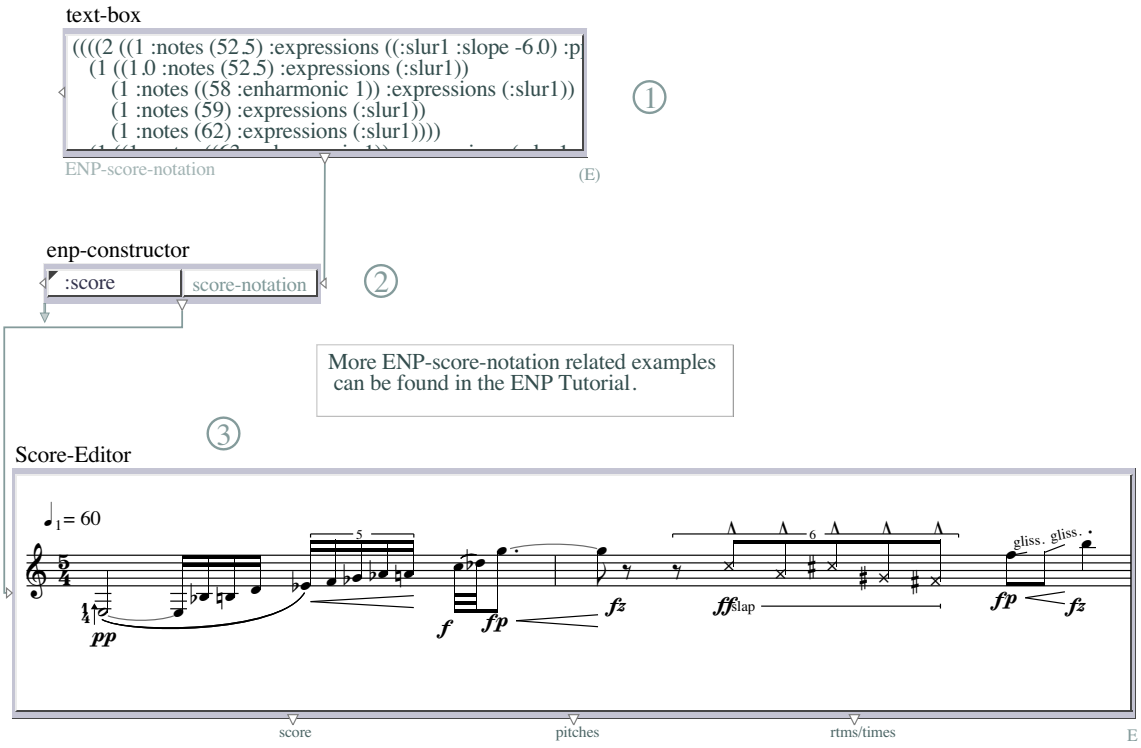

Figure 2.35: 02-enp-constructor

2.3.4.3 ENP-Constructor-Mix

In this example we first construct a chord object using the 'enp-constructor' box (1). After this we read the chord object from the 'Chord-editor' box and pass it to a box called 'enp-score-notation' (2). This box converts any ENP object to ENP-score-notation format. Here we can use various filters to include or exclude properties. In our case we exclude nothing, i.e. we get a duplicate of the original chord without modifications (3). You can apply a filter by double-clicking the right-most input just beside the ':exclude' input (4). A dialog appears giving you several options which information you can exclude from the result. Select, for instance, the 'expressions' option. If you now reevaluate the patch at (3), the expressions contained in the original chord will disappear.
See also the related patch called 'ENP-score-notation-filter'.

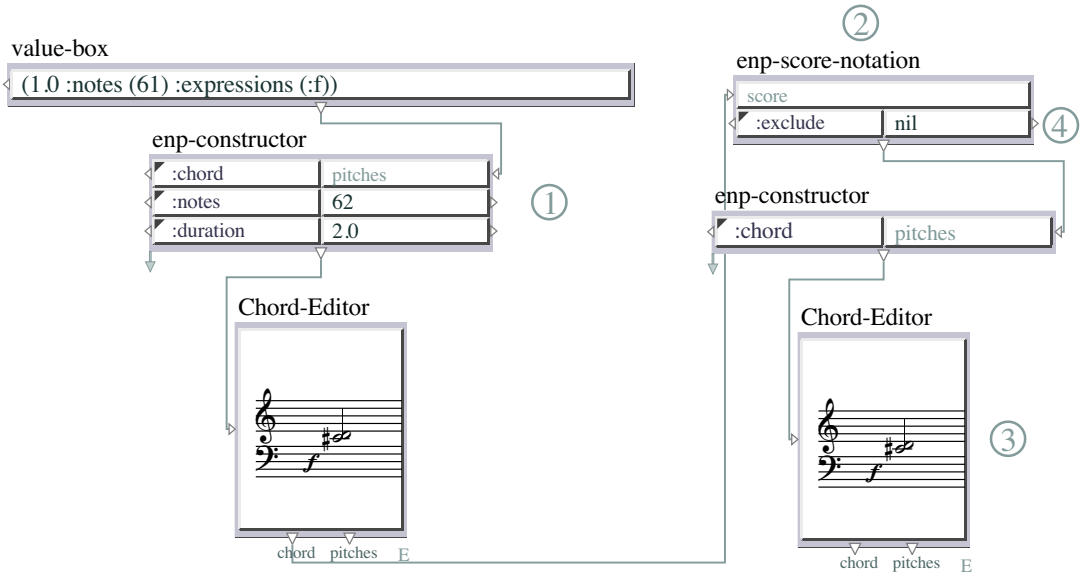

Figure 2.36: 03-ENP-constructor-mix

2.3.4.4 ENP-Object-Composer

This patch demonstrates the use of the 'enp-object-composer' box. We start with a flat list of two note objects that are generated in (1). In the resulting four scores we add from left to right each time a new list around the previous list expression. Thus we gradually imbed the two notes deeper and deeper in the ENP score hierarchy resulting in: (2) two parts (3) two voices (4) two measures (5) two beats.

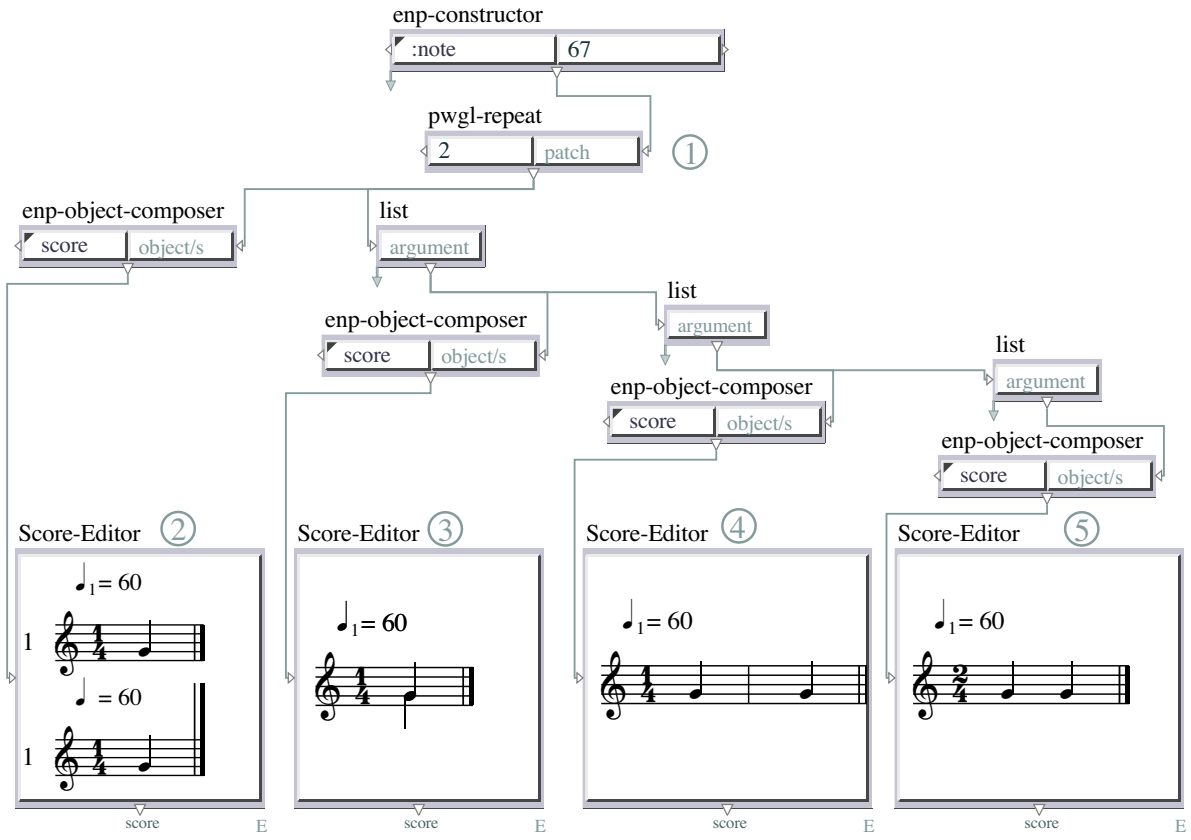

Figure 2.37: 04-ENP-object-composer

2.3.4.5 ENP-Score-Notation-Filter

This patch uses as a starting point a score given in (1). We use here two 'enp-score-notation' boxes.
To the left (2) we use the ':exclude' option with 'nil', and thus the resulting score is an exact duplicate of the original.
To the right (3) we use in turn the ':include' option with 'nil'. This means that we strip all other score information except the underlying rhythmic structure.

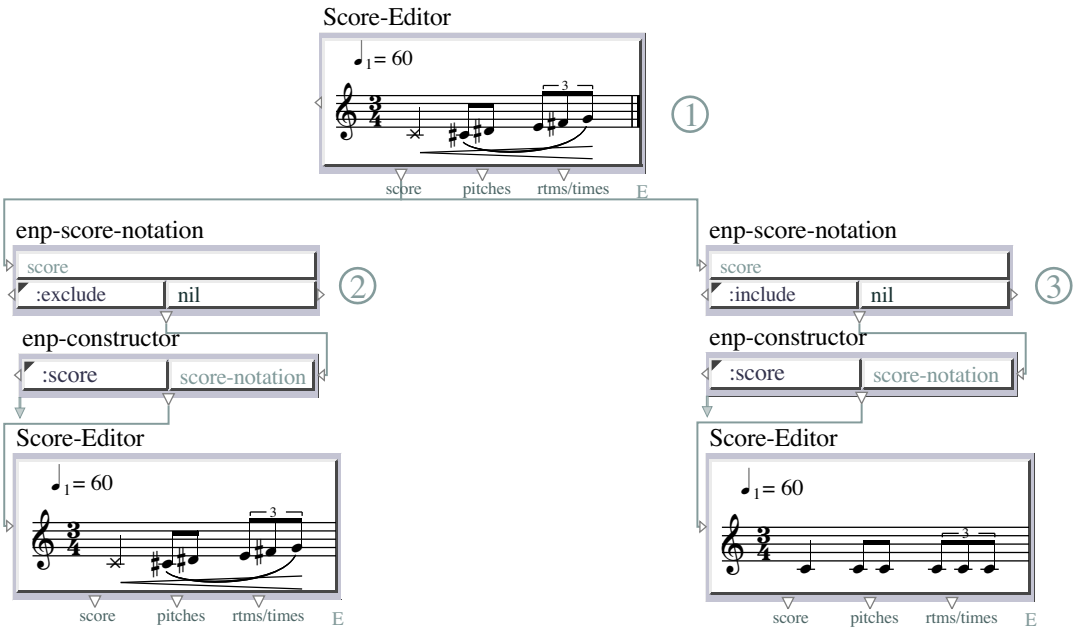

Figure 2.38: 05-ENP-score-notation-filter

2.3.4.6 Advanced-Topics

This patch contains two abstractions.
In the first one, 'Combo', we extract various substructures from the original score (1). In (2) you can use a switch box to choose the substructure you are interested in. The result can be found in (3).
In the second abstraction, 'Constructor', we generate scores using several'enp-constructor' boxes. You can use a switch box (1) to choose the resulting score which is found in (2).

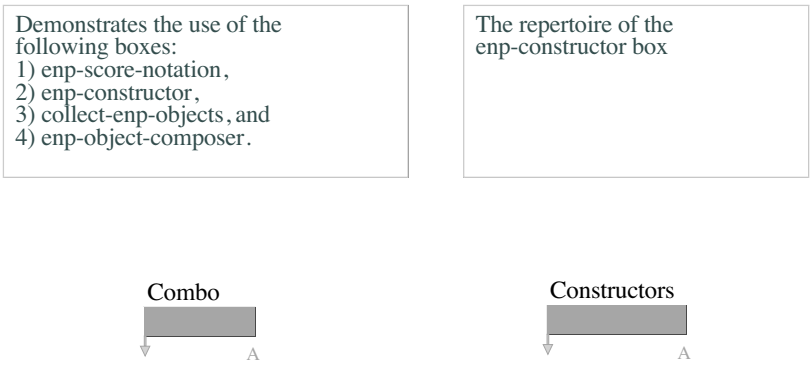

Figure 2.39: 06-Advanced-Topics

2.3.4.7 Adjoin-Voices

The two boxes, 'collect-enp-objects' and 'enp-object-composer' can in many cases be used in pairs as demonstrated in this example. Here, the two voices found in the input scores at the top are concatenated by collecting the measures (1) from both scores and then appending them together (2). The 'enp-object-composer' (3) creates a new voice using the measures collected from the input scores. This box fills in the missing structures between the source and target objects.
It is important to copy the objects as 'collect-enp-objects' returns the actual instances found in the input scores. This is done using the 'duplicate-instnce' box (4)

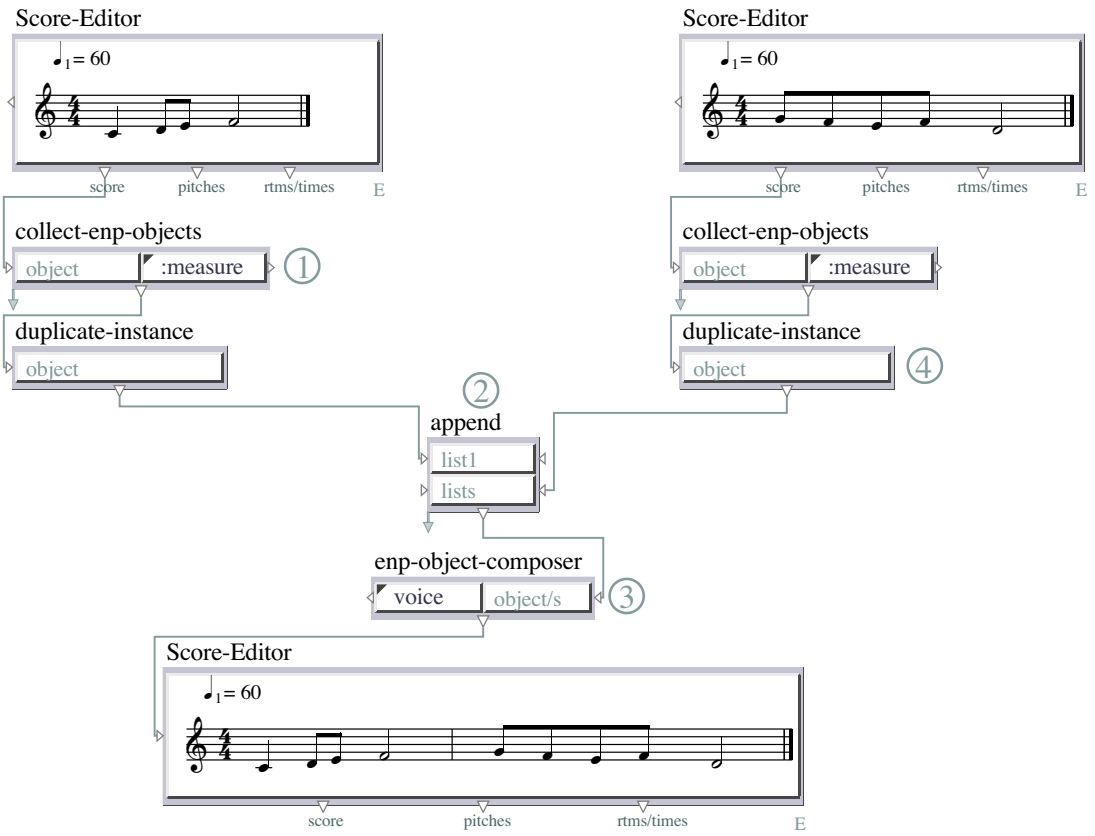

Figure 2.40: 07-adjoin-voices

2.3.4.8 Collect-Objects

This example demonstrates how to retrieve segments form a score. The input score at the top contains two parts. The 'collect-enp-objects' box (1) collects all the objects that are of the type indicated by the menu-box (2). The first of the collected objects is selected an converted into a score using the 'enp-object-composer' box (3). In effect, this patch defines a method to retrieve from the score, the first part, the first voice, the first measure, the first beat, the first chord, and the first note.

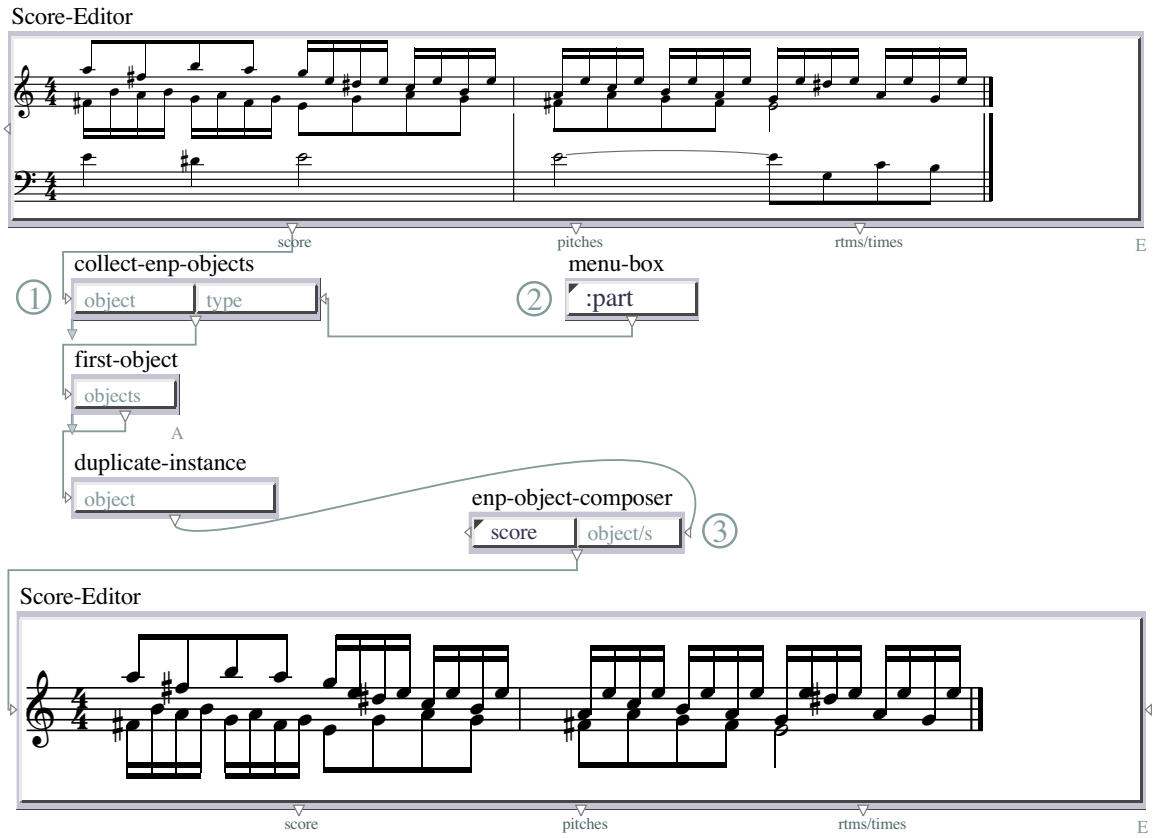

Figure 2.41: 08-collect-objects

2.3.4.9 Constructing-ENP-Objects-2

This is a more complex example utilizing the 'enp-constructor' -box. Here we demonstrate how to build a chord containing several notes with varying durations and dynamics. On the left, the two text boxes show partial analysis information of a bell sound (1). The first of them, gives the frequencies and the second one gives the initial amplitudes scaled between 0.0 and 1.0 according to the loudest partial.
The individual start-times of the notes inside the chord can also be manipulated by a switch (3). The first position sets all start-times to 0.0 and the second position, in turn, calculates the start-times in reverse order according to the initial amplitudes.
Note also the use of ':offset-dur' attribute that allows to give a note inside a chord an extra duration that is relational to the duration of the containing chord.

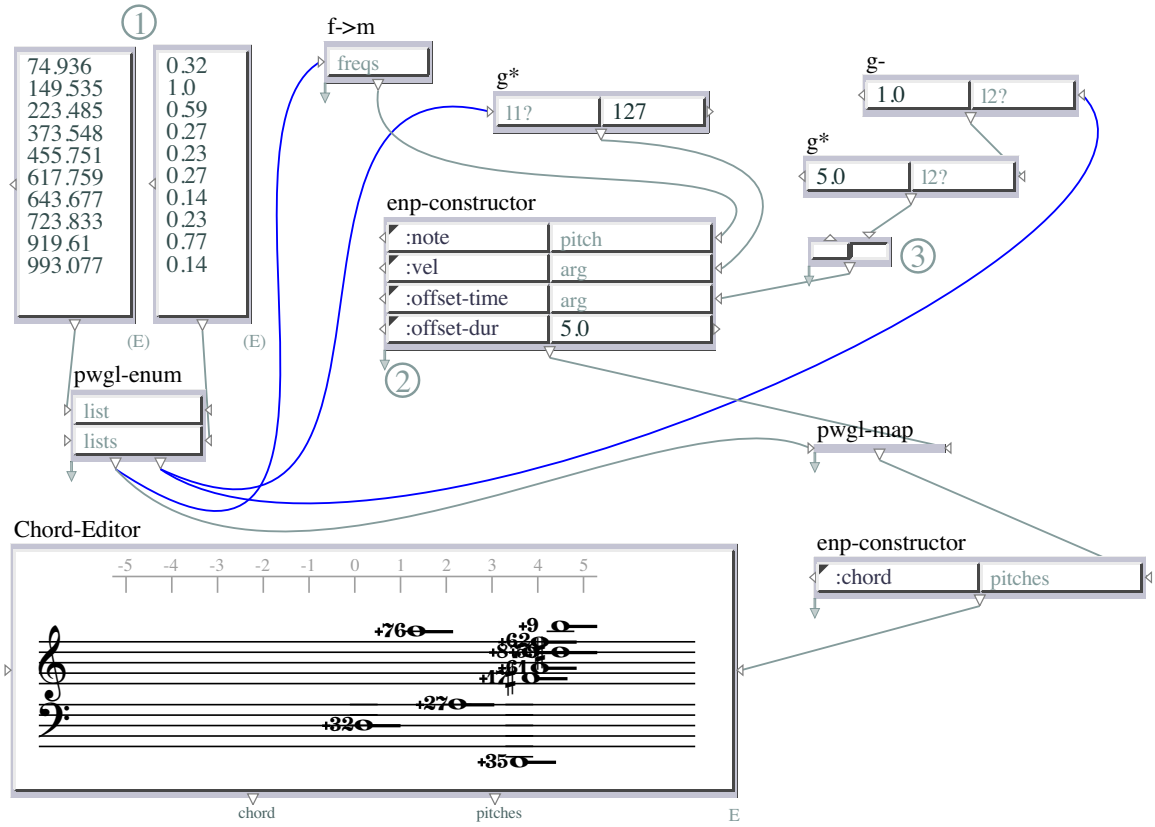

Figure 2.42: 09-constructing-enp-objects-2

2.3.4.10 Constructing-ENP-Objects-3

The 'pwgl-enum' box (1) gives two parallel lists that define pitch and offset-time respectively. The 'text-box' in (2) gives examples of different ways of defining ENP-expressions and their properties. Here we define, among other things, a Score-BPF expression containing some points (see the ':points' attribute). The points are given in a list format where each sublist is a pair consisting of a relative time and the corresponding (y)value. The start-times are automatically scaled inside the extent of the resulting expression. The 'enp-constructor' box (3) is used build the note objects. It can be used to define the properties of ENP-objects, such as, notes and chords.

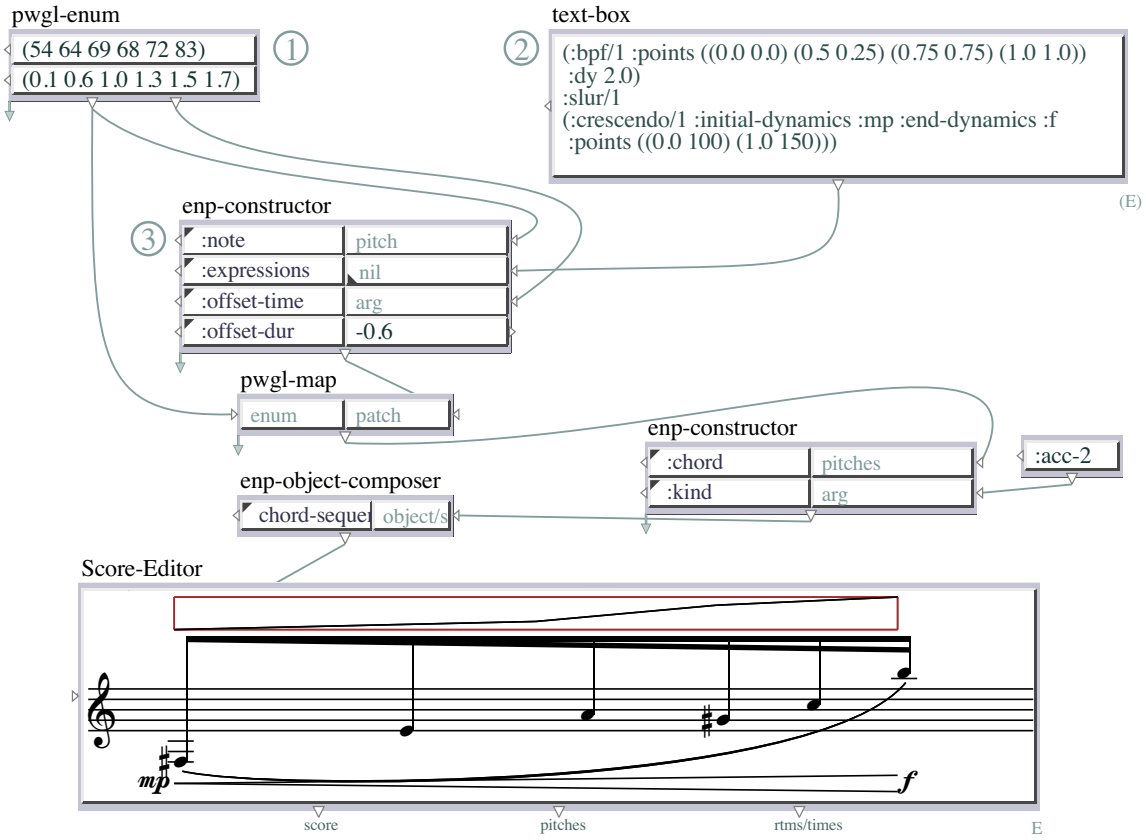

Figure 2.43: 09-constructing-enp-objects-3

2.3.4.11 Canvas-Expression

This patch shows how the Canvas-expression can be used with ENP-score-notation.
(1) Demonstrates how a specific expression is expressed textually and how the textual representation can be used to create an exact copy of the original object.
(2) Gives the current repertoire of graphical primitives inside Canvas-expression, which are: (a) :arrow (b) :filled-circle (c) :circle (d) :lineloop (e) :polygon (f) :linestrip (g) :line (h) :text
See the text-box at the bottom of the patch for possible attributes for each of the primitives.

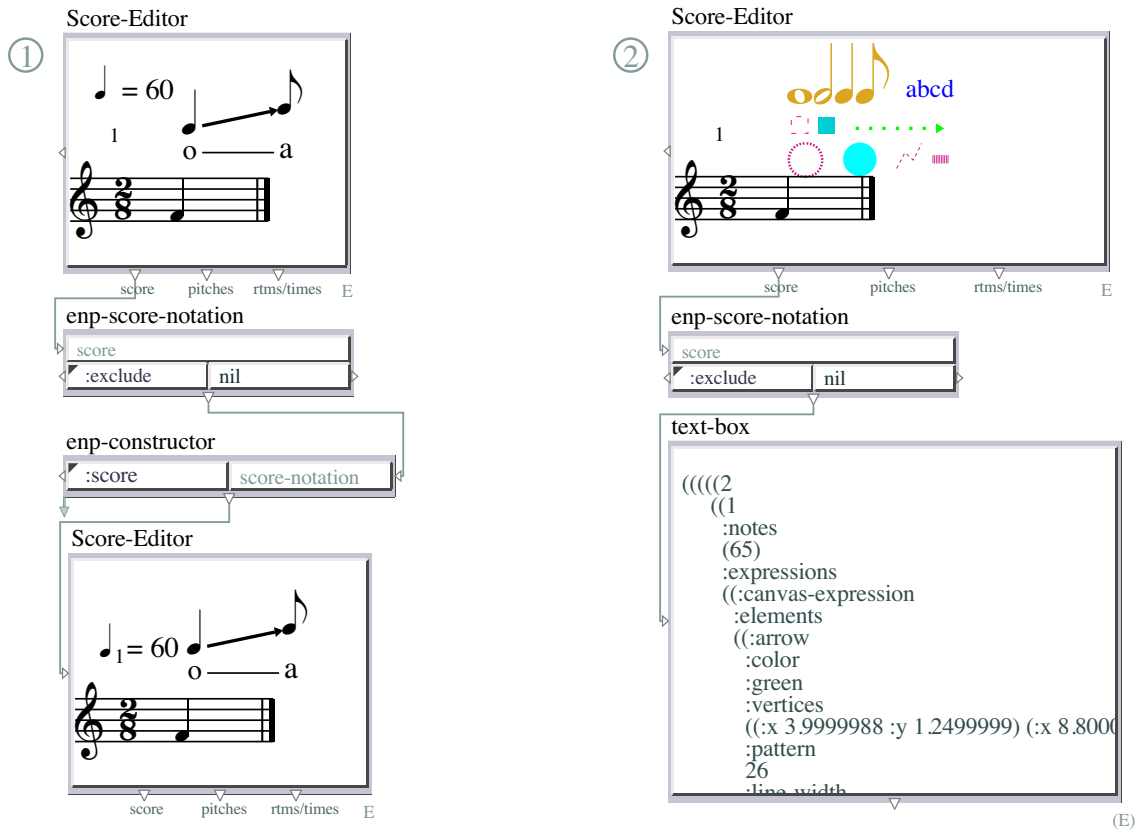

Figure 2.44: 20-canvas-expression

2.3.5 Scripting

2.3.5.1 Scripting Syntax

Scripting can be used to produce various side-effects (such as expressions, analytical information, timing, color, clef position, etc.) to an input-score using the 'enp-script' box. The position, outlooks and type of the side-effects are defined by the 'rules' input. A scripting rule uses a similar syntax than a Score-PMC rule. Both start with a pattern-matching part where the syntax is identical in both cases. The difference, however, deals with the Lisp-code part. In a Score-PMC rule the Lisp-code part typically returns a truth value, whereas in a ENP-script rule the Lisp code either returns an expression definition or nil.
In the former case the rule is used to insert expressions in the score, for instance:

```
(* ?1 ?2 ?3
   (?if  (add-expression 'group ?1 ?2 ?3
           :info (sc-name (list (m ?1) (m ?2) (m ?3)))))
   "analyse␣3-card␣melodic␣set-classes")
```

Here the rule adds an expression ('add-expression') of type 'group' to all adjacent melodic three-note formations ('?1 ?2 ?3'). The string appearing in the score is given after the ':info' keyword.
In the latter case the rule is used for setting slot values for music notation related objects such as notes, chords, beats, measures, and so on:

```
(* ?1 :chord
   (?if (when (m ?1 :complete? t)
          (dolist (n (notes ?1))
            (if (< (midi n) 60)
                (setf (clef-number n) 1)
                (setf (clef-number n) 0)))))
   "assign␣notes␣below␣60␣to␣bass␣clef")
```

This rule sets the clef number of all notes in a chord so that notes having a midi-value below 60 will be assigned to the bass clef.
For a more detailed discussion of the constraints syntax see the 'Constraints' chapter of this tutorial.
Also for more information see the paper 'ENP-Script-ISMIR05.pdf' that is found in the 'documentation/publications' folder.

2.3.5.2 Mark-Matchings

This patch demonstrates visually the behavior of basic rule accessor keywords that can be used in the pattern-matching part of a rule (see also the 'Constraints' section for more information).
The rules contain the Lisp-expression (?mark ?1 ?2 ...) which is used to temporarily store the indicated objects (?1 and ?2) so that they can later be displayed in the score. This information is not saved or copied along with the score or the patch but needs to be generated again if needed. This feature can be used for example for pedagogical or debugging purposes.
When evaluated the 'enp-script' box stores all cases found by '?mark' as a database that can be inspected by the user by opening a rule diagnostics dialog with a double-click.
The upper part of the dialog contains two columns. To the left we have names of all current scripting rules.
When a rule is selected a list of all matched positions can be found in the right part of the dialog. When one of these positions is clicked, the exact position is shown in the input score using various drawing devices (such as circles, connected shapes, bezier functions, etc.).

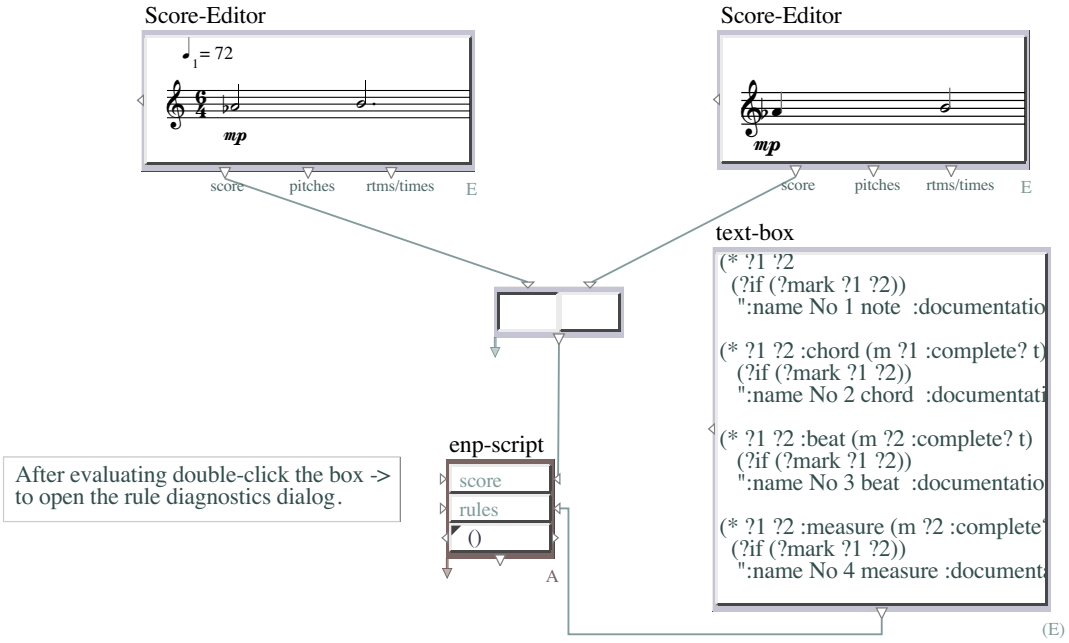

Figure 2.45: 01b-mark-matchings

2.3.5.3 Analysis

2.3.5.3.1 ENP-Script

A 'enp-script' box is used here to annotate a score with analytical information. The 'text-box' gives one scripting rule that analyzes all adjacent melodic 3-note groups according to their pitch-set-class content.
If the third input, 'selection?', of the 'enp-script' box is 'T', then the analysis will be applied only to the current selection.
Analysis information that have been added by enp-script can be removed using the 'enp script history' dialog.

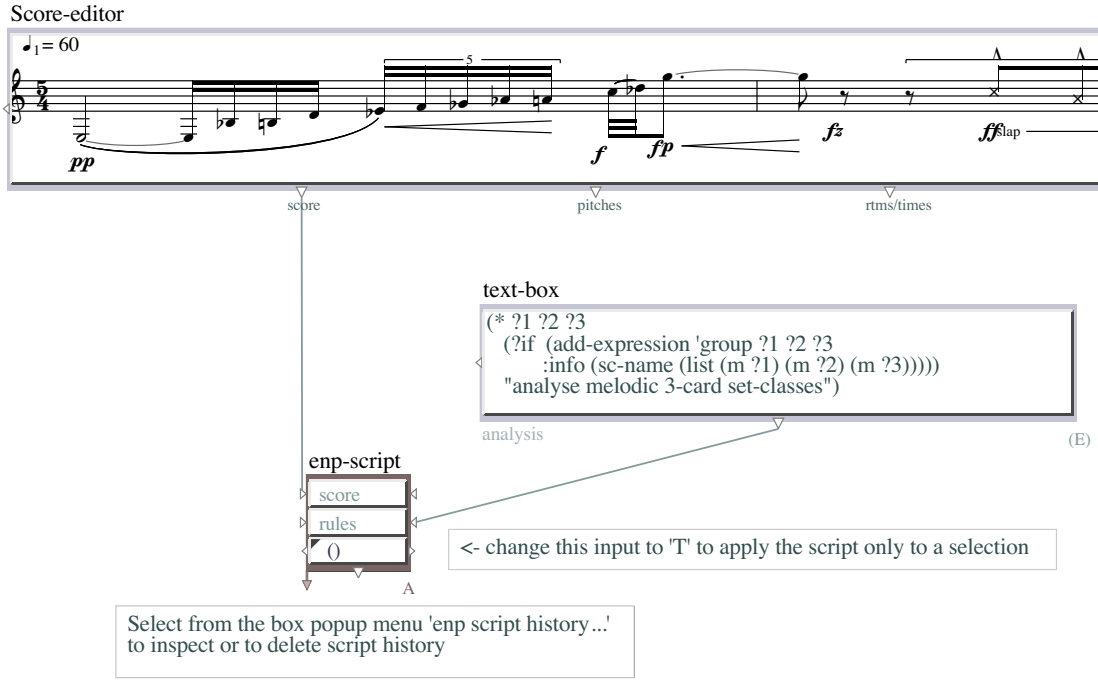

Figure 2.46: 01-ENP-script

2.3.5.3.2 Schoenberg-Op25

This example contains four scripting rules. The first three rules add standard expressions such as slurs, accents and crescendo markings. The fourth scripting rule annotates the score with row analysis information. The Lisp-code box ('row-forms') contains some Lisp code that is needed for the row analysis.
The 'rule-filter' box below the 'text-box' can be used as follows. First double-click the box. This action opens a dialog showing all the documentation strings of the incoming rules. Here you can select any subset that will be output from the box.

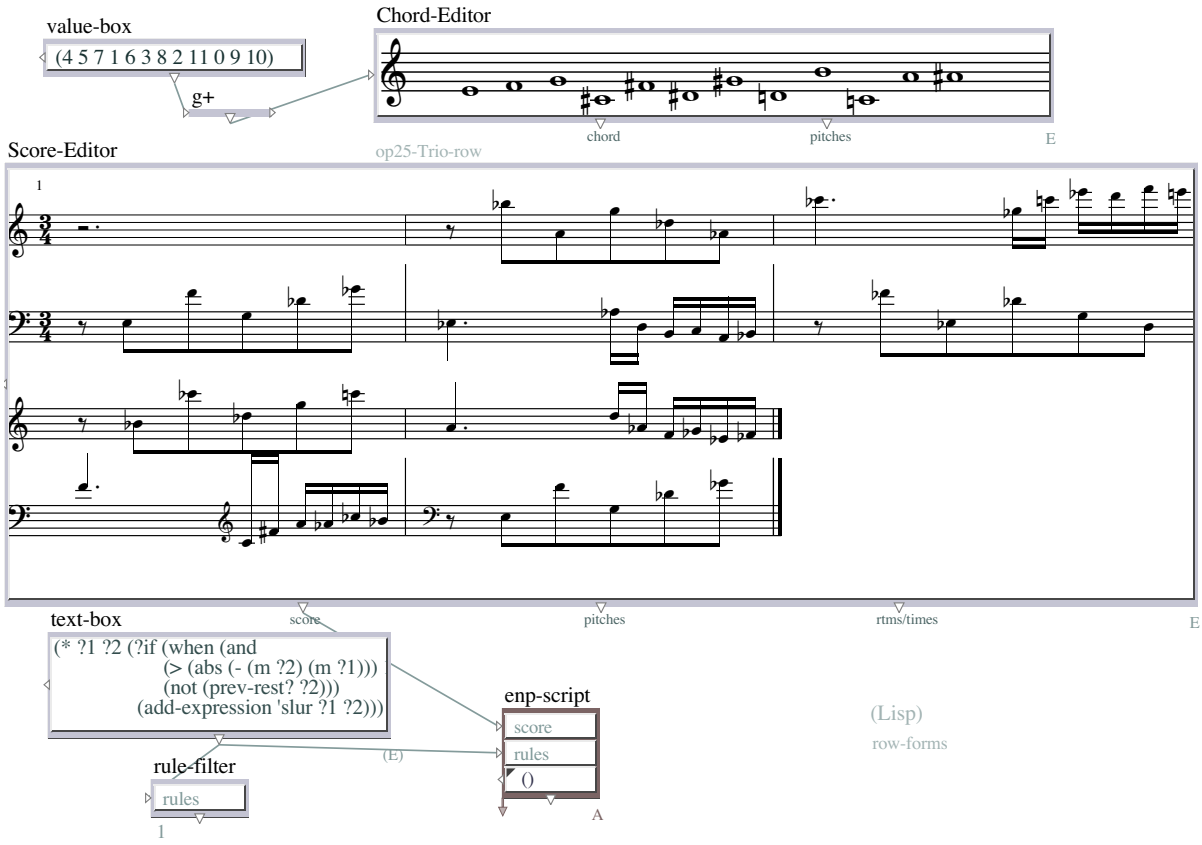

Figure 2.47: 02-Schoenberg-op25

2.3.5.3.3 Kuitunen-Vocal-Texture

This example utilizes two scripting rules. The first one finds all harmonic formations that consist of symmetric chords. These cases are marked with connected shapes that are cycled through three different shapes. The second one performs a motivic analysis of consecutive 3-note cells.
To see the complete analysis evaluate the 'enp-script' box.

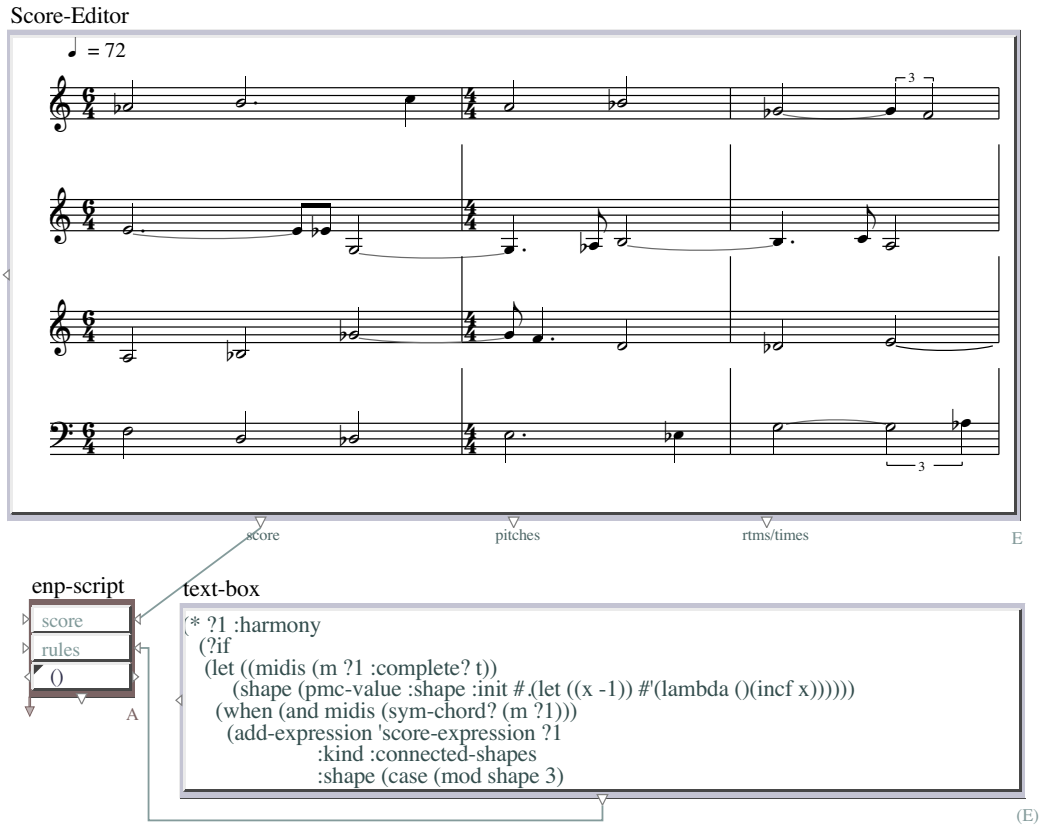

Figure 2.48: 03-Kuitunen-vocal-texture

2.3.5.3.4 Parallel-Fifths

This example shows how classical voice-leading mistakes can be marked in a score. This is a fairly complex analysis task, and we use here in the 'm-method' the special ':vl-martrix' (vl = voice-leading) keyword to access a matrix object consisting of notes that define both adjacent harmonic and melodic relations simultaneously. From this object we access a list of lists of notes, where each sublist forms a melodic (or horizontal ':h') movement. Using this data structure we can perform the actual analysis where we find all voice-leading cases that form parallel fifths.

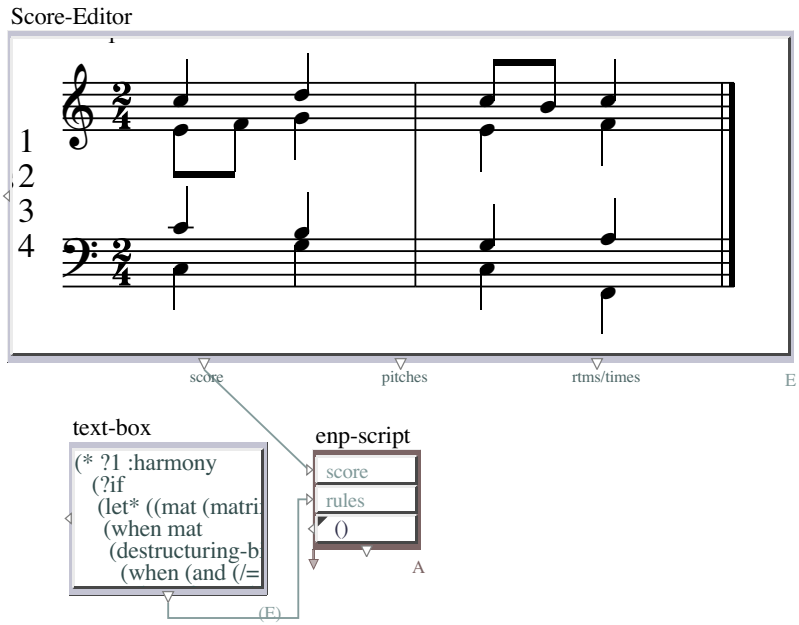

Figure 2.49: 05-parallel-fifths

2.3.5.4 Score Manipulation

2.3.5.4.1 Arpeggio-Chords

This patch demonstrates how simple midi-list values (given in the 'text-box' in the upper part of the patch) can be converted to a more meaningful and readable musical presentation with the help of ENP-script. The list input contains four 12-note chords, which are are transformed step-wise to a result that is shown in the 'Score-Editor'. These steps include setting the piano staff and assigning the lowest notes to the bass clef; adjusting the start time of each chord; and finally manipulating the offset-time of individual notes within the chords so that the result contains either upward or downward arpeggios with an accelerating gesture.

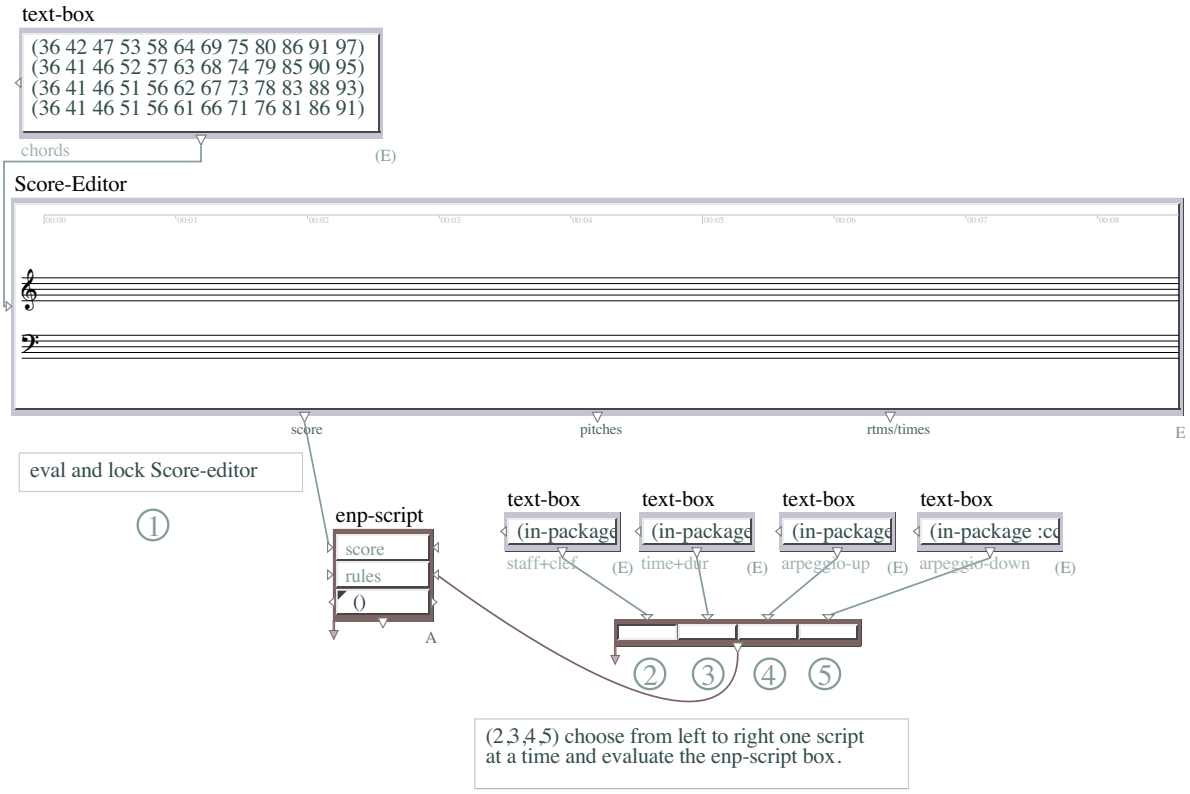

Figure 2.50: 01-arpeggio-chords

2.3.5.4.2 Beethoven-Expressions

Here two basic scripting rules are used to add automatically slurs and staccatos to the score.

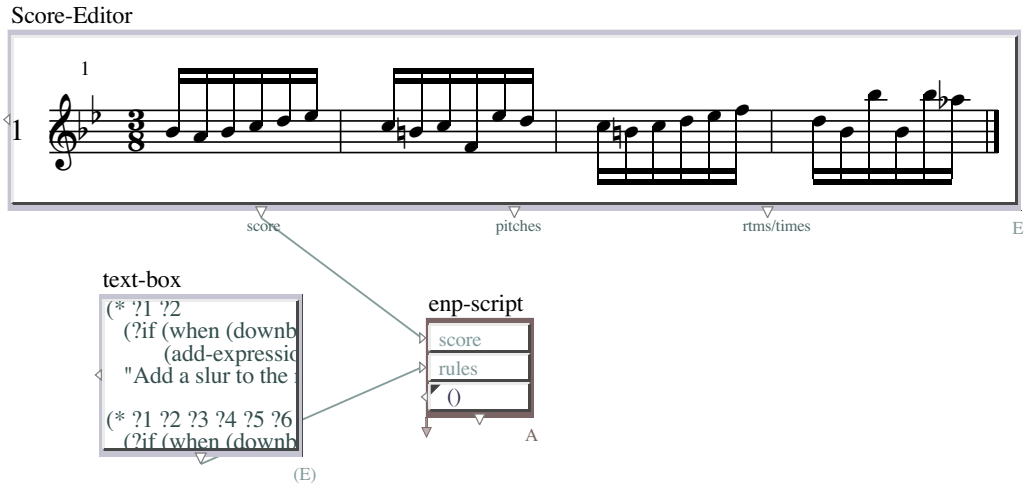

Figure 2.51: 02-Beethoven-expressions

2.3.5.4.3 Chopin-Octaves

This example differs from the previous scripting patches as it contains two sets of rules that are given in two separate text-boxes. The first contains a rule that adds octaves to all notes in the score. The second set contains a rule that assigns all notes below middle-C to the bass clef.
Normally the scripting engine executes all rules in parallel (i.e. all rules are applied to the first note, then all rules are applied to the second note, etc.).
Here we cannot use this basic scheme as the first rule makes side-effects to the basic score structure (i.e. adds notes). Thus we must ensure that the first rule has been completely executed before the second one starts.
This is why we give the rules as a list (see the 'list' box). When the script engine encounters this list, it executes the first rule for all notes in the score before starting with the next rule.

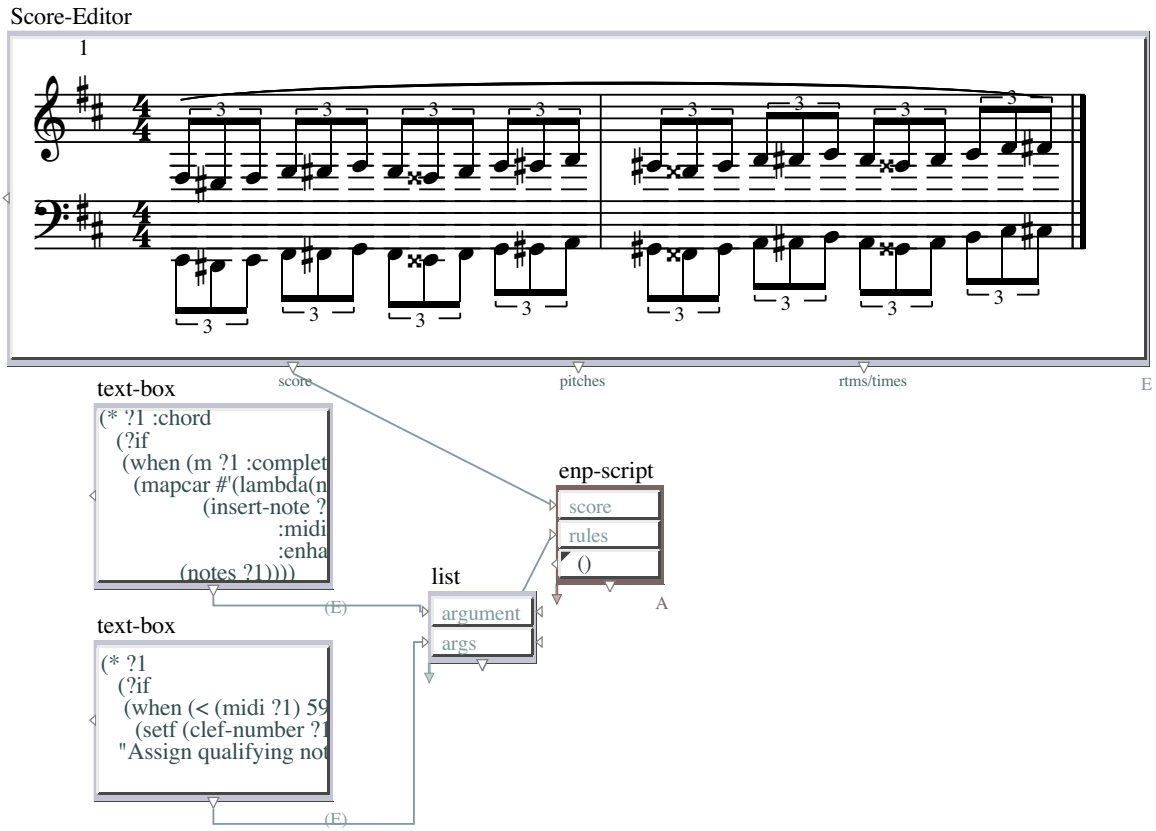

Figure 2.52: 03-Chopin-octaves

2.3.5.4.4 RTM-Modification

Scripting can also be used to change the metric structure of a score. Here we halve the denominators of time signatures and halve the unit-length of main-beats. This results in a score where all chords have exactly equal timing information when compared to the original score, although the measure and beat information is written differently in the modified score.

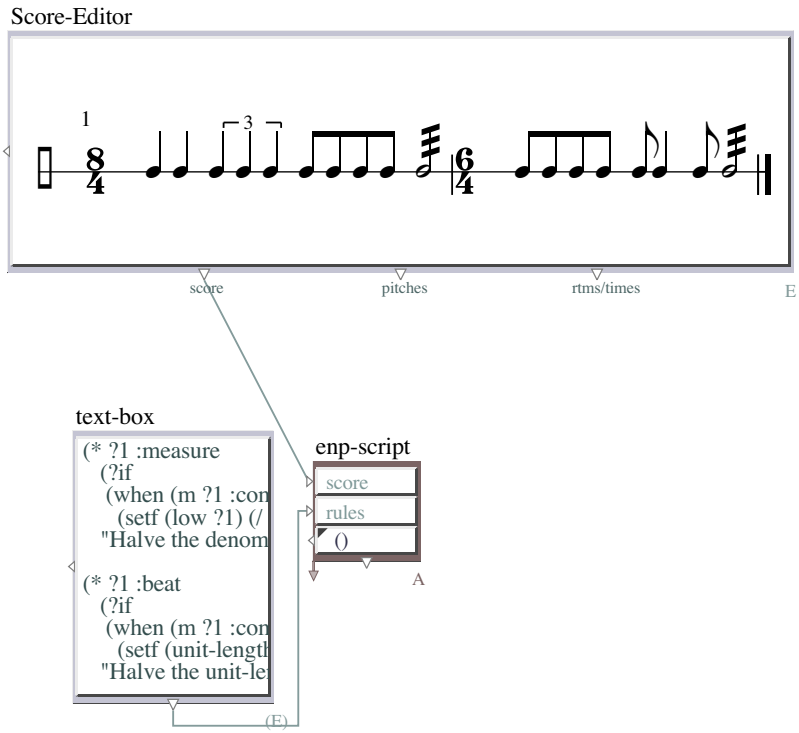

Figure 2.53: 04-rtm-modification

2.3.5.4.5 Chopin-Layout

In this patch we use scripting to change some layout features of a score. In the first rule we diminish the beat-size of a beat of the upper voice (this results in a smaller note-head size) and in the second one we change x-offset value of the chords to ensure that the stems are aligned with the modified note-heads.
The patch contains a switch box that allows to toggle between the original layout and the modified one.

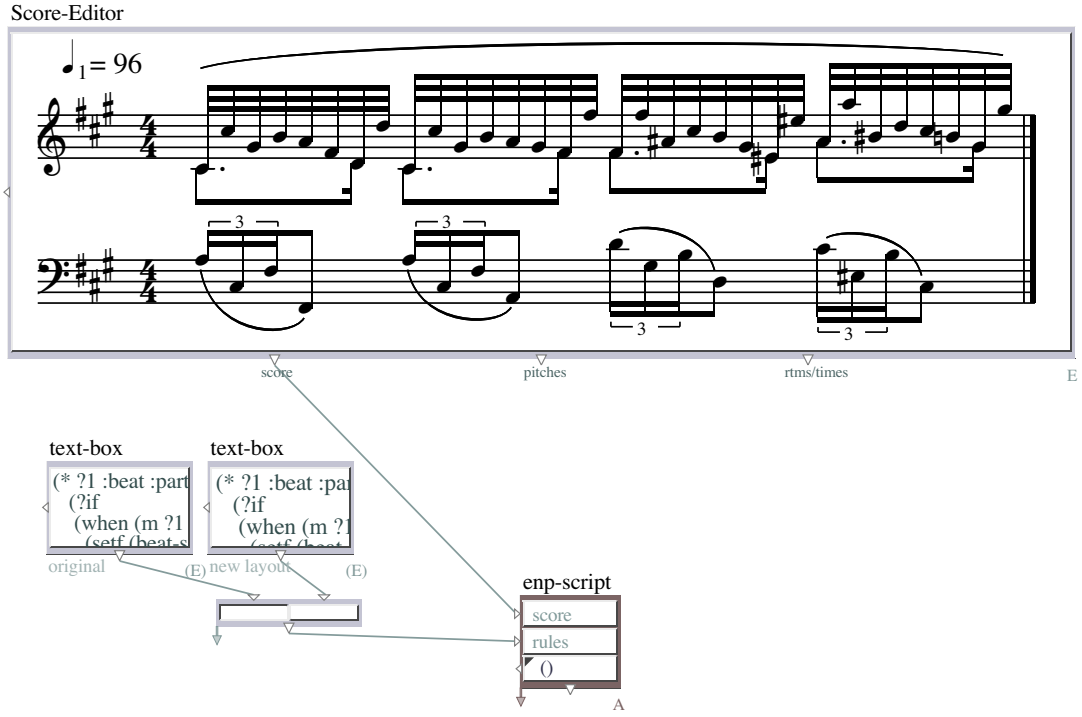

Figure 2.54: 05-Chopin-layout

2.3.5.4.6 Reassigning-Pitches

In this example we duplicate a score (1), and assign new pitches for a result score (2). The new pitches are given in (3). Note that all other score information of the original score are kept in the result score.
The actual pitch assignment is realized with enp-script that has one scripting rule (4). The 'update-pitches' abstraction (5) has two purposes: it defines one 'pre' operation and one 'post' operation that are fed to the 'prepare-fns' input of the 'enp-script' box (the 'pre' operation is run before enp-script, and the 'post' operation after enp-script). First, the 'pre' operation stores the pitch list under the keyword ':pitches' (this list is then accessed inside the scripting rule). Second, the 'post' operation restores correct midi-values for the tied notes after running enp-script.

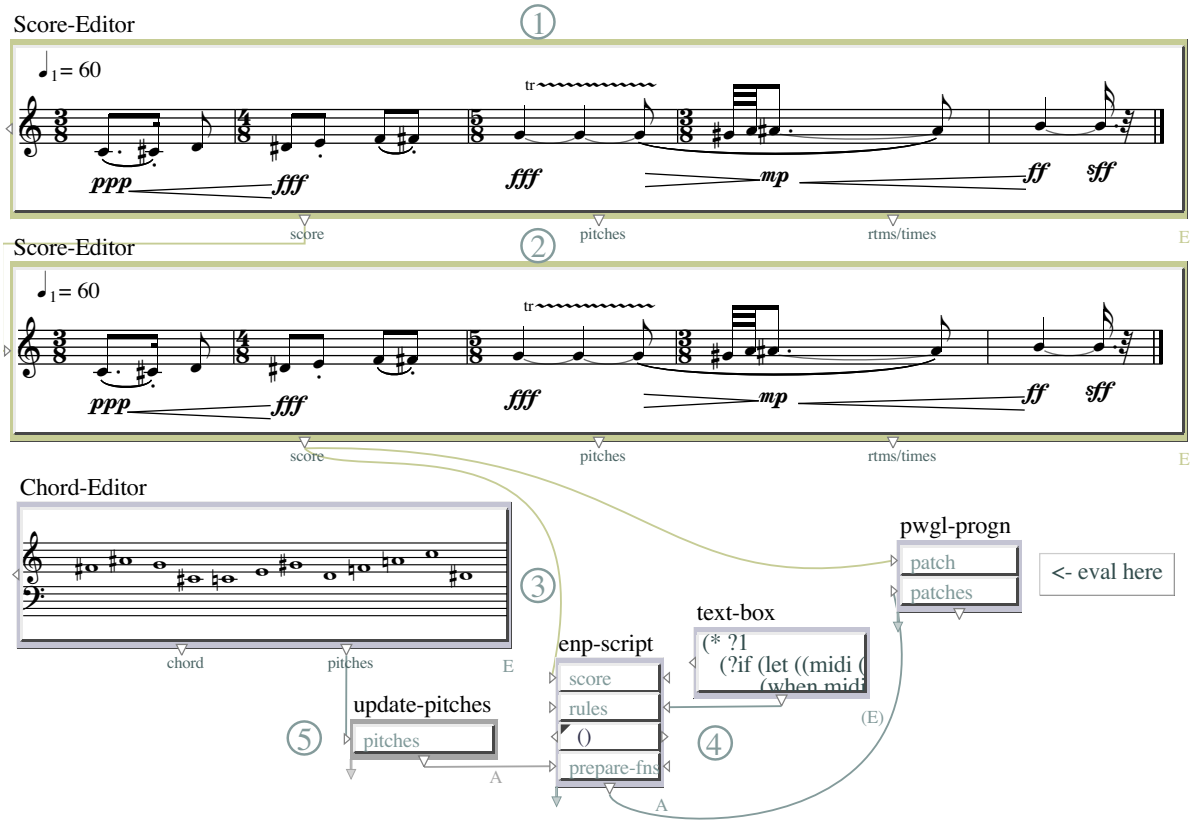

Figure 2.55: 06-Reassigning-pitches

2.3.6 Rhythm

2.3.6.1 Basic

This section demonstrates how to create beat objects in a patch for ENP. A beat consists of a RTM-list and a unit-length.
The RTM-list defines the internal proportions of a beat and it can consist of positive numbers (attacks), negative numbers (rests), or positive floating point numbers (ties).
The unit-length, in turn, gives the number of units that are used for this beat (the actual unit, e.g. 1/4 note, 1/8 note, etc., is defined in the time-signature of the measure that owns the beat) .
In (1) and (2) a RTM-list is created and the result is shown in (3). In (4) the user gives a unit-length. This information is collected in an 'enp-constructor' box (5) that creates the beat object. This result is in turn fed to an 'enp-object-composer' box (6) that creates the final score object (7).

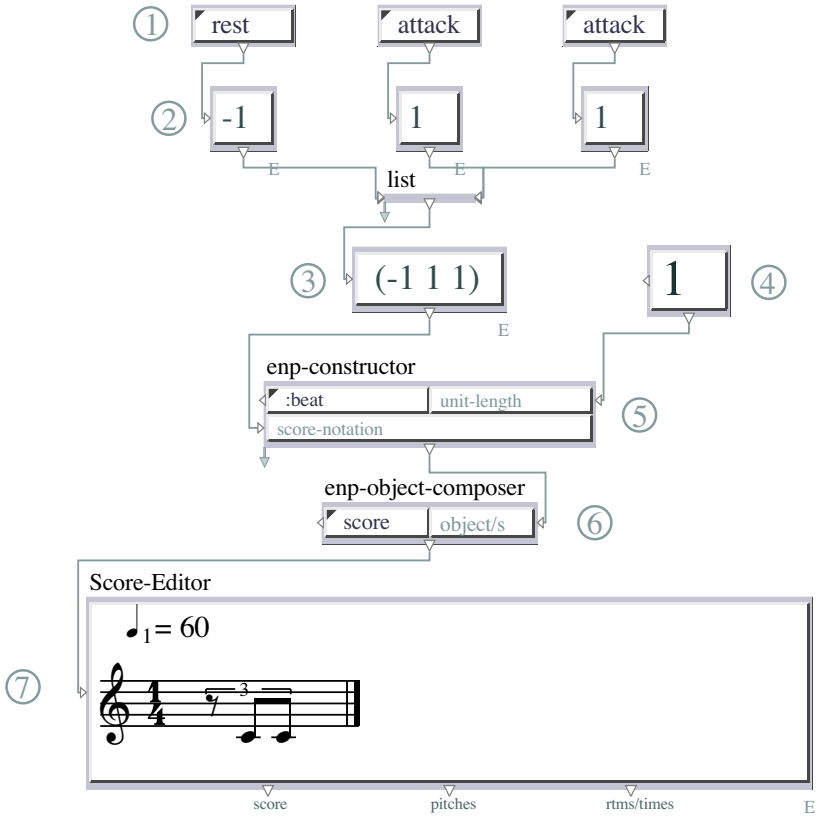

Figure 2.56: 01-basic

2.3.6.2 Random-Rhythms

This patch demonstrates how to create random beats. First we create a RTM-list (1). After this we simplify the RTM-list with the 'gcd' (greatest common divisor) function (2). (3) shows the resulting RTM-list. Finally in (4) we generate a random unit-length.

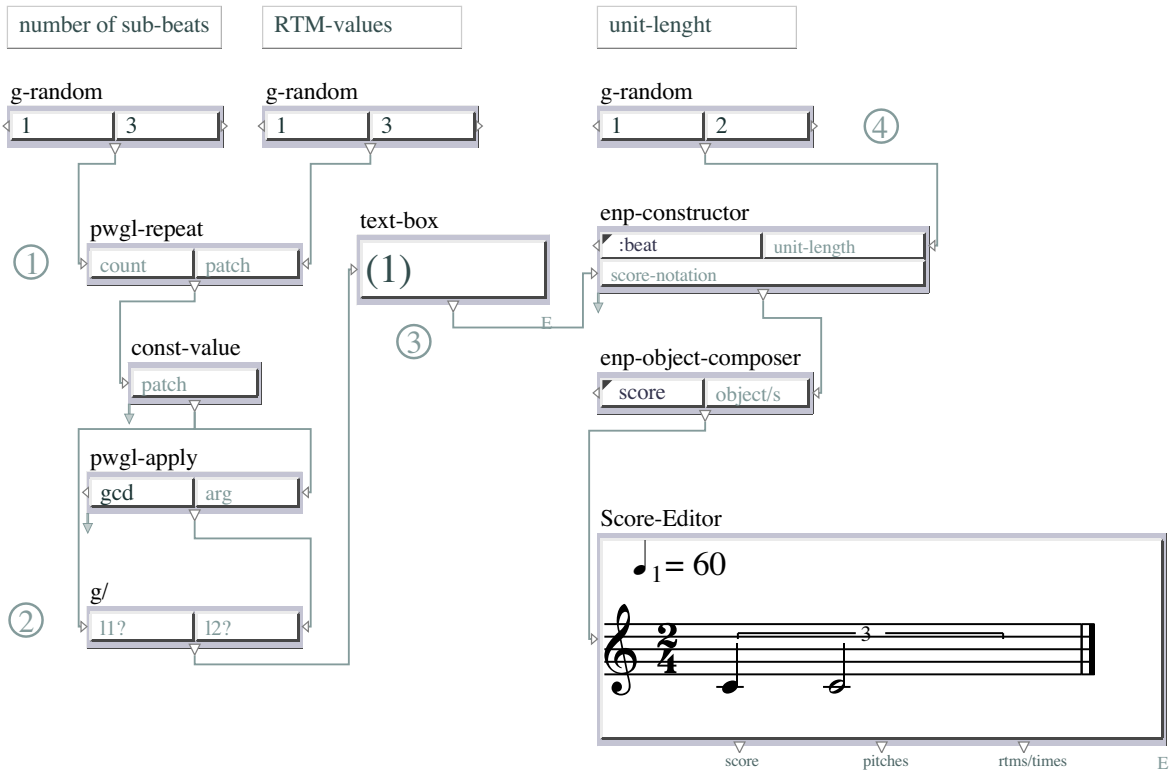

Figure 2.57: 02-random-rhythms

2.3.6.3 Pulses

In this example the shape of a bpf controls the number of elements of 10 RTM-lists (1). The current bpf is first sampled, scaled and truncated between 1 and 5 (2). After this the final RTM-list is created in a loop (3). Here we create also the beat objects with the help of a 'enp-constructor' box.
The 2D-Editor contains 4 bpfs and the current one can be selected using the arrow up/down keys.

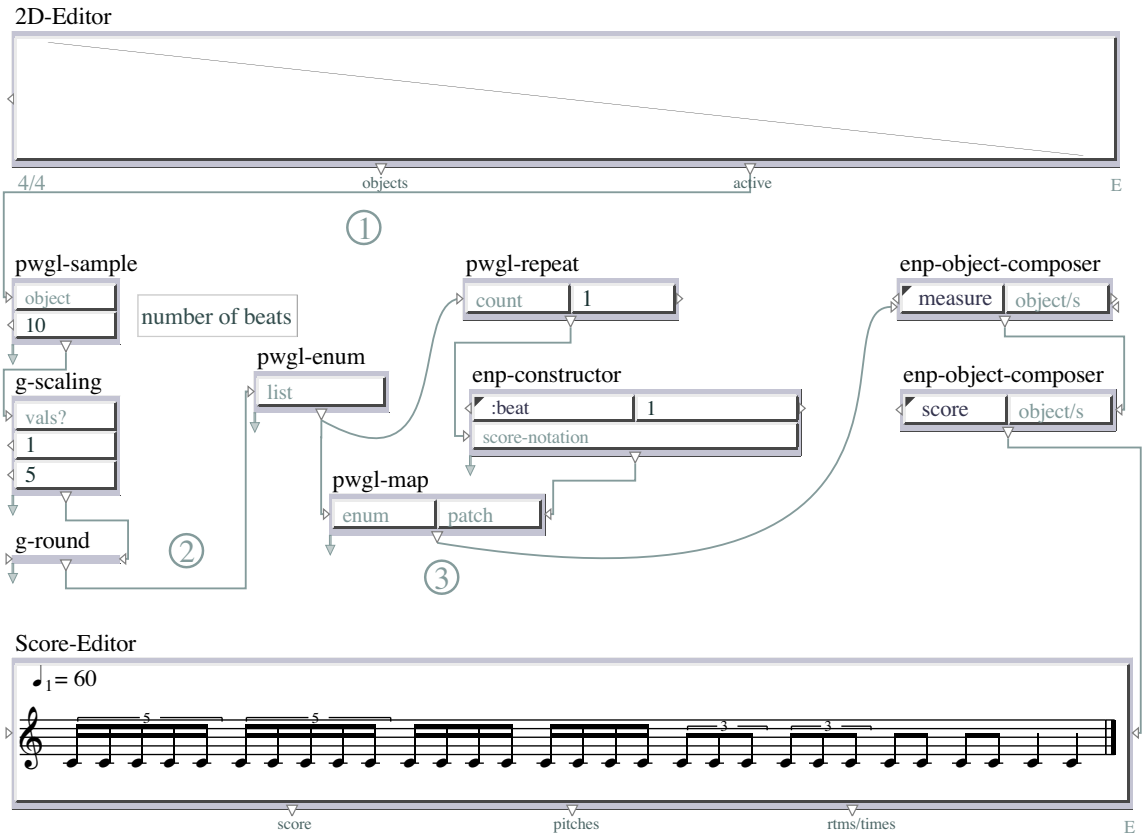

Figure 2.58: 03-pulses

2.3.6.4 Rhythm-Database

A menu-box is used as a database of different RTM-lists (1). This database can inspected and edited by double-clicking the menu input.

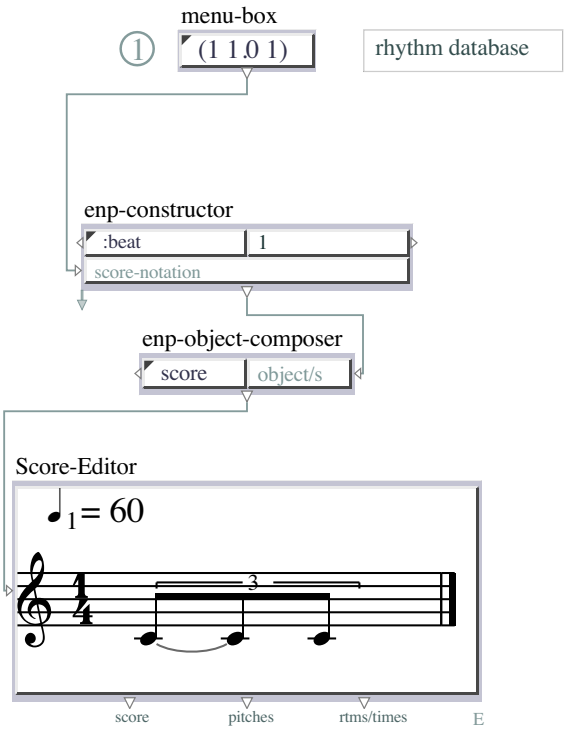

Figure 2.59: 04-rhythm-database

2.4 Special-Boxes

2.4.1 Display-Box

2.4.1.1 OpenGL Macros

There are special macros that can be used to draw graphical primitives, such as, lines, polygons, circles, etc. The following list enumerates the current set. The names should give appropriate hints as to their usage (the required parameters are shown in parenthesis):

(1) `draw-2D-arrow (x1 y1 x2 y2 &key (xsize 0.5) (ysize 0.3))`

(2) `draw-2D-box (x1 y1 x2 y2 &key filled-p)`

(3) `draw-2D-circle (x y r &key filled-p)`

(4) `draw-2D-line (x1 y1 x2 y2)`

(5) `draw-2D-lines (&rest 2D-coordinates)`

(6) `draw-2D-point (x y &key (size 1.0))`

(7) `draw-2D-polygon (&rest 2D-coordinates)`

(8) `draw-2D-quads (&rest 2D-coordinates)`

(9) `draw-2D-text (x y string font scaler &optional (justification t))`

(10) `draw-2D-triangles (x1 y1 x2 y2 x3 y3 &key filled-p)`

(11) `with-2D-object (type)`. Type can be one of the following keywords: :polygon :points :line :lines :line-loop :line-strip :triangles :triangle-strip :triangle-fan :quads :quad-strip. Vertices can be added to the object with `add-2D-vertex` (see below).

(1) `add-2D-vertex (x y)`

There are also some OpenGL specific macros that can be used to perform, for example, various matrix operations, or to change the graphics attributes, such as, transparency or current line width. The following enumerates the available macros arranged in groups according to their usage:

(1) `with-GL-translate (dx dy)`

(2) `with-GL-scale (factor)`

(3) `with-GL-rotate (degree)`

(4) `with-GL-matrix`

(1) `with-GL-color (color)`

(2) `with-GL-color-and-alpha (color alpha)`

(3) `with-GL-color-3f (r g b)`

(4) `with-GL-color-4f (r g b alpha)`

(1) `with-GL-line-width (width)`

(2) `with-GL-point-size (size)`

(3) `with-GL-line-stipple (pattern)`. Pattern is a two digit number defining the pattern, e.g., 88 means 8 points on, 8 points off; and 44 means 4 points on, 8 points off.

2.4.1.2 Colors

Color keyword
:ALICEBLUE
:ANTIQUEWHITE
:AQUAMARINE
:AZURE
:BEIGE
:BISQUE
:BLACK
:BLANCHEDALMOND
:BLUE
:BLUEVIOLET
:BROWN
:BURLYWOOD
:CADETBLUE
:CHARTREUSE
:CHOCOLATE
:CORAL
:CORNFLOWERBLUE
:CORNSILK
:CYAN
:DARK-BLUE
:DARKGOLDENROD
:DARKGREEN
:DARKKHAKI
:DARKOLIVEGREEN
:DARKORANGE
:DARKORCHID
:DARKSALMON
:DARKSEAGREEN
:DARKSLATEBLUE
:DARKSLATEGRAY
:DARKSLATEGREY
:DARKTURQUOISE
:DARKVIOLET
:DEEPPINK
:DEEPSKYBLUE
:DIMGRAY
:DIMGREY
:DODGERBLUE
:FIREBRICK
:FLORALWHITE
:FORESTGREEN

:GAINSBORO
:GHOSTWHITE
:GOLD
:GOLDENROD
:GRAY
:GRAY-BLUE
:GREEN
:GREENYELLOW
:GREY
:HONEYDEW
:HOTPINK
:INDIANRED
:IVORY
:KHAKI
:LAVENDER
:LAVENDERBLUSH
:LAWNGREEN
:LEMONCHIFFON
:LIGHT-BLUE
:LIGHT-BROWN
:LIGHT-RED
:LIGHTBLUE
:LIGHTCORAL
:LIGHTCYAN
:LIGHTGOLDENROD
:LIGHTGOLDENRODYELLOW
:LIGHTGRAY
:LIGHTGREY
:LIGHTPINK
:LIGHTSALMON
:LIGHTSEAGREEN
:LIGHTSKYBLUE
:LIGHTSLATEBLUE
:LIGHTSLATEGRAY
:LIGHTSLATEGREY
:LIGHTSTEELBLUE
:LIGHTYELLOW
:LIMEGREEN
:LINEN
:MAGENTA
:MAROON
:MEDIUM-BLUE
:MEDIUM-BROWN
:MEDIUM-GREEN

:MEDIUM-YELLOW
:MEDIUMAQUAMARINE
:MEDIUMBLUE
:MEDIUMORCHID
:MEDIUMPURPLE
:MEDIUMSEAGREEN
:MEDIUMSLATEBLUE
:MEDIUMSPRINGGREEN
:MEDIUMTURQUOISE
:MEDIUMVIOLETRED
:MIDNIGHTBLUE
:MINTCREAM
:MISTYROSE
:MOCCASIN
:NAVAJOWHITE
:NAVY
:NAVYBLUE
:OLDLACE
:OLIVEDRAB
:ORANGE
:ORANGERED
:ORCHID
:PALEGOLDENROD
:PALEGREEN
:PALETURQUOISE
:PALEVIOLETRED
:PAPAYAWHIP
:PEACHPUFF
:PERU
:PINK
:PLUM
:POWDERBLUE
:PURPLE
:RED
:ROSYBROWN
:ROYALBLUE
:SADDLEBROWN
:SALMON
:SANDYBROWN
:SEAGREEN
:SEASHELL
:SIENNA
:SKYBLUE
:SLATEBLUE

:SLATEGRAY
:SLATEGREY
:SNOW
:SPRINGGREEN
:STEELBLUE
:TAN
:THISTLE
:TOMATO
:TURQUOISE
:VIOLET
:VIOLETRED
:WHEAT
:WHITE
:WHITESMOKE
:YELLOW
:YELLOWGREEN

2.4.1.3 Examples

2.4.1.3.1 Basic

PWGL-display-box
PWGL-display-box can be used to visualize a wide range of information. This patch gives a simple example how the display-box can be used to draw graphics in the patch. Double-click the box to view and edit the code.

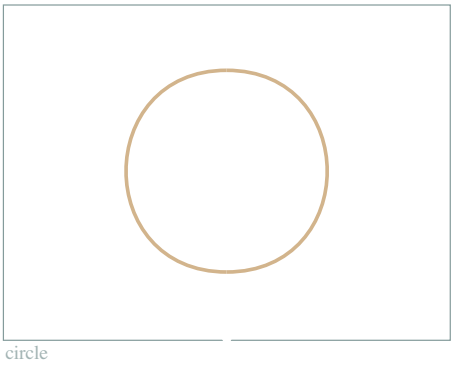

Figure 2.60: 02-basic

2.4.1.3.2 Using-Variables

PWGL-display-box
PWGL-display-box can be used to visualize a wide range of information. This patch gives a simple example how the display-box can be used to draw graphics in the patch. Double-click the box to view and edit the code.

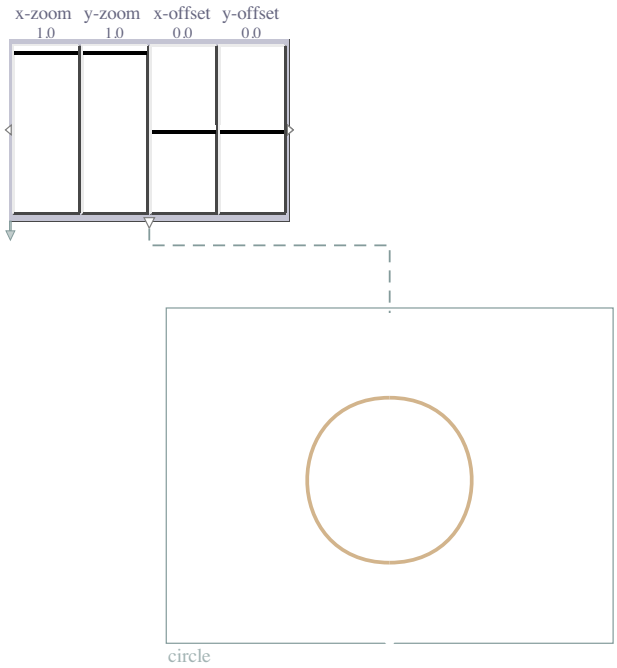

Figure 2.61: 02-using-variables

2.4.1.3.3 Macros

The collection of macros/primitives

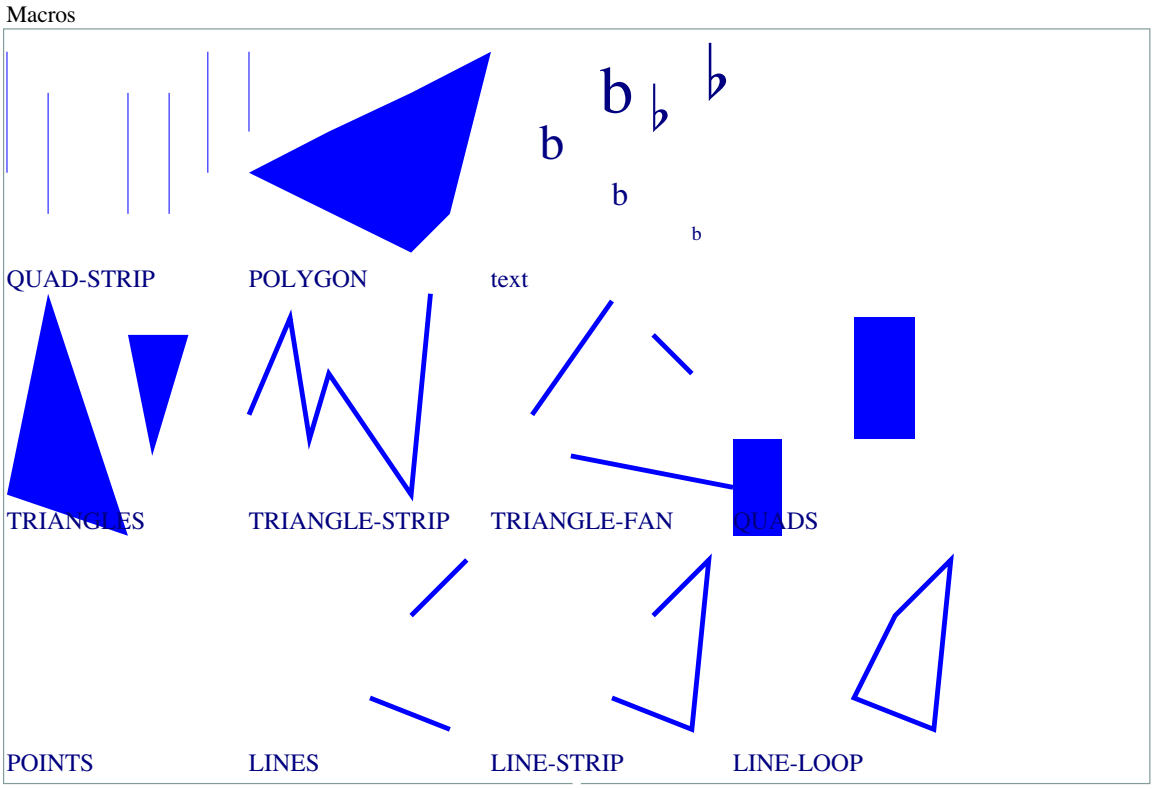

Figure 2.62: 03-macros

2.4.1.3.4 Lorenz-Attractor

See: http://en.wikipedia.org/wiki/Lorenz_attractor

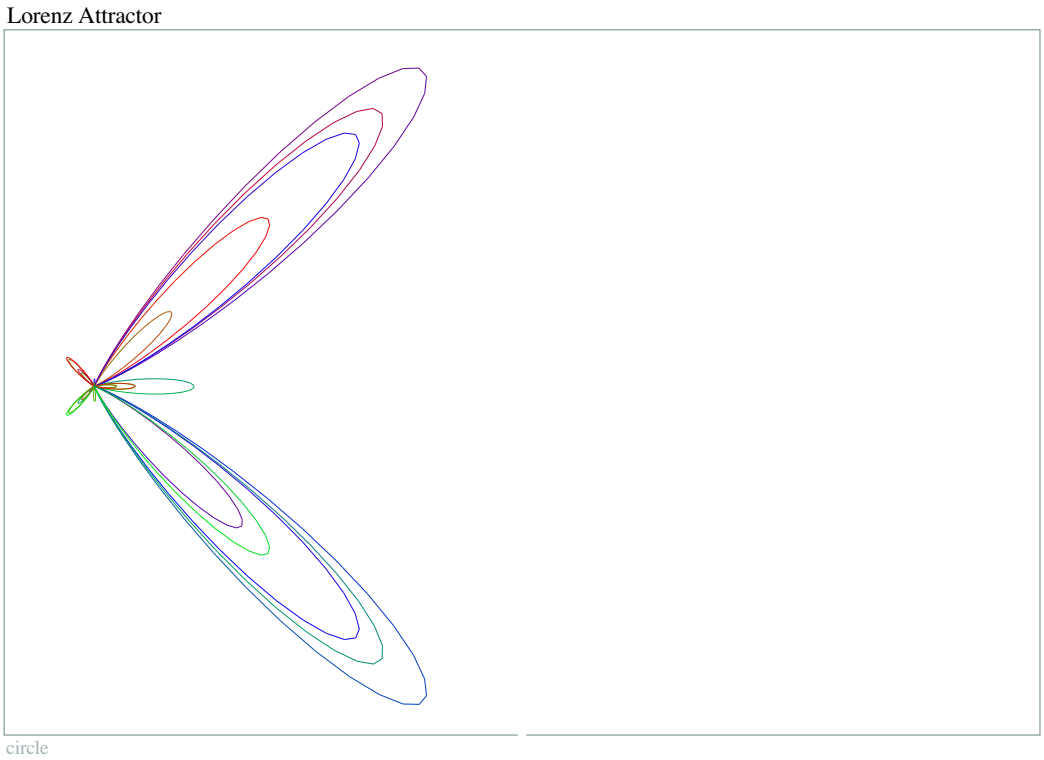

Figure 2.63: 05-lorenz-attractor

2.4.2 Shell

2.4.2.1 Introduction

The PWGL SHELL library provides a collection of boxes to interface with the UNIX shell. The SHELL library allows the user to call virtually any shell program, to redirect and pipe commands, and to input the results back to be used as a part of a normal PWGL patch. The SHELL library currently implements 4 main boxes:

(1) pwgl-shell-box

(2) pwgl-shell-redirect-box

(3) pwgl-shell-pipe-box

(4) pwgl-shell-execute-box

The documentation for the SHELL library API can be found here:

2.4.2.2 Basic-Principles

2.4.2.2.1 Basics

Editing
(1) The empty box by default does not have any shell commands assiciated to it. To enter the command you can either:
(a) double click the "?" button, or
(b) type '+', and enter the shell command name in the dialog.
In the dialog you have several options: you can write the name by hand, browse the file system, or use the menus on the rigth to access /usr/bin and /usr/loca/bin folders.
(2) When the box has a shell command it is also possible to extend/remove inputs with by typing "+" or "-" respectively. If you explicitely need to add an empty option use the '=' key.
(3) The options can be removed by selecing the option input box and typing "Cut" or "Delete".
(4) The inputs can be rearranged with the arrow keys (up/down)
(5) An option can be replaced with another one by double-clicking on the option input box and typing a new option name.
Getting Help
When a box has a shell command associated to it you can type 'h' to see the UNIX manual page.

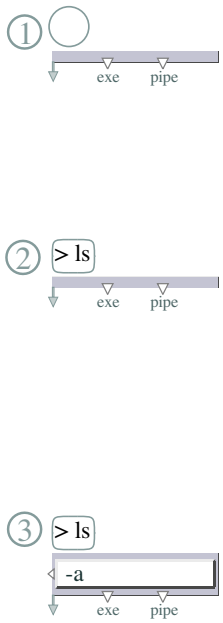

Figure 2.64: 00a-basics

2.4.2.2.2 Managing-Options

Some of the shell commands can include dozens of options and it is usually impossible to remeber them all.
By default the shell boxes are not aware of the applicable options.
If you select the 'ls' box (1) and type '+' to add an option you'll notice that an empty option input box is added. There is however, a simple way of creating databases for the shell commans.
(2) Shows a string that is copy pasted from the output of 'man ls' (the options section to be precise). By evaluating the box at (3) a database is automatically generated from the 'man' output. Now, if you again try to add an option to the 'ls' box shown at (4) you should be able to notice that things behave a little different.
Note, that it is possible to make a multiple selection. So, if you want the directory listing to display both the number of file system blocks actually used by each file as well as to show if a particular pathname names a file of a directory (e.g., ls -F -s) you can choose both options at the same time.
This makes it easier to manage the options and it also acts documentation to the boxes as well

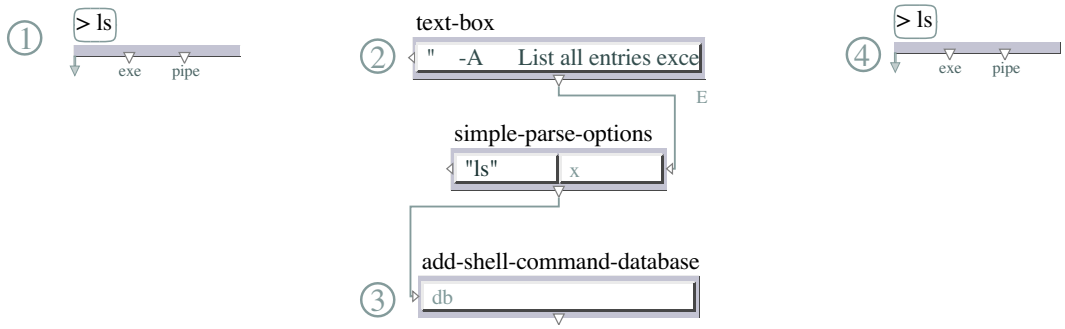

Figure 2.65: 00b-managing-options

2.4.2.2.3 Error-Handling

The shell-boxes read the return value of the executed program(s) and visually show if the execution resulted in an error.
Here, we use the UNIX command 'man' to try to locate the UNIX manual page for a program called 'foobar'. Evaluation of the box should - hopefully - result in an error.
When a shell box is in an uresolved (error) state a visual indication is shown. Now when moving the mouse over the failing box the UNIX error string and/or number is displayed the message area located at the bottom of the patch window.
Uresolved boxes will reset automatically when evaluate the next time (note that this may result in an error again). The box can be reset manually by selecting it and typing 'r'.
NOTE: this behavior cannot be observed in a tutorial patch! You need to open the patch in a separate window...

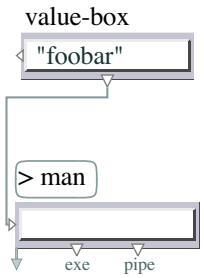

Figure 2.66: 01a-error-handling

2.4.2.2.4 Output

The shell-boxes can do some post-processing to the output. This patch shows the currently available post-processors:
(1) The default, returns the result as a string.
(2) :read-from-string calls the lisp read-from-string function to the result string. This is useful if you need to return numbers, for example.
(3) :list, takes the output line by line, returning a list of strings
(4) :void, just returns NIL. This is useful when the result is not interesting and also helps to avoid cluttering the PWGL output window with unimportant information.
(5) :boolean returns either NIL or the data returned by the system call.
The post-process can be set by typing 'p' on a shell-box.

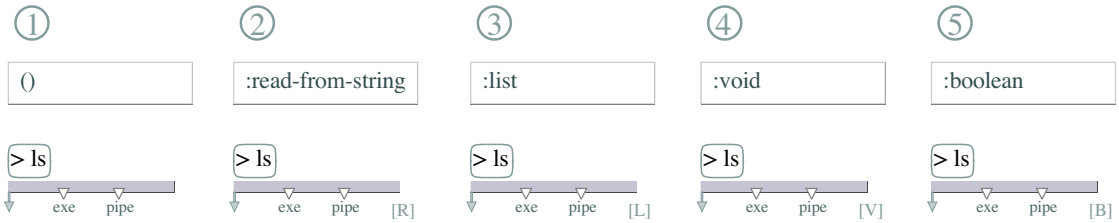

Figure 2.67: 01b-output

2.4.2.3 Examples

2.4.2.3.1 Simple-Io-Example

This is a simple example demonstrating how shell boxes can pass information to PWGL. Here, the 'cal' command is used to calculate the calendar for a given year. The result is stored in the 'text-box' at the bottom of the patch.

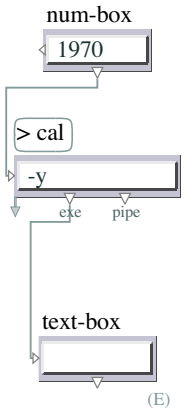

Figure 2.68: 01-simple-IO-example

2.4.2.3.2 Opening-and-Viewing

This patch demonstrates how the shell-box can be used display and examine the contents of a file.
in (1) we define a pathname pointing to a sampled sound file.
(2) open the file with user definable application – the application can be selected with the 'menu-box' shown in (3).
In (4) we use quicklook to preview the contents of the sound file (quicklook is a Mac OS X Leopard feature only).

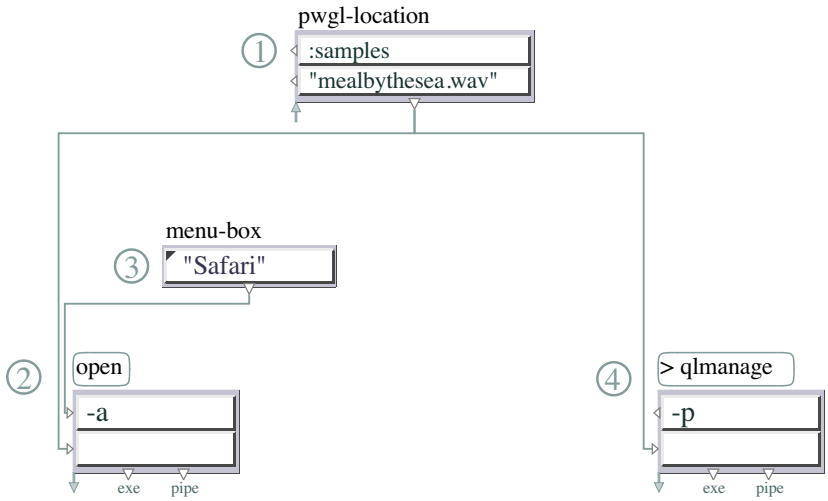

Figure 2.69: 02-opening-and-viewing

2.4.2.3.3 Executing

(1) Gives two options for locating lilypond.
In (2) the 'which' box is used to check if the application exists. Note, that the box is in 'BOOLEAN' output mode.
(3) 'pwgl-if' box, depending on the value returned by the 'which' box, either evaluates the lilypond script and opens the resulting PDF file (4) or displays a message (5) that the application is not found.
The patch is evaluated at (3)

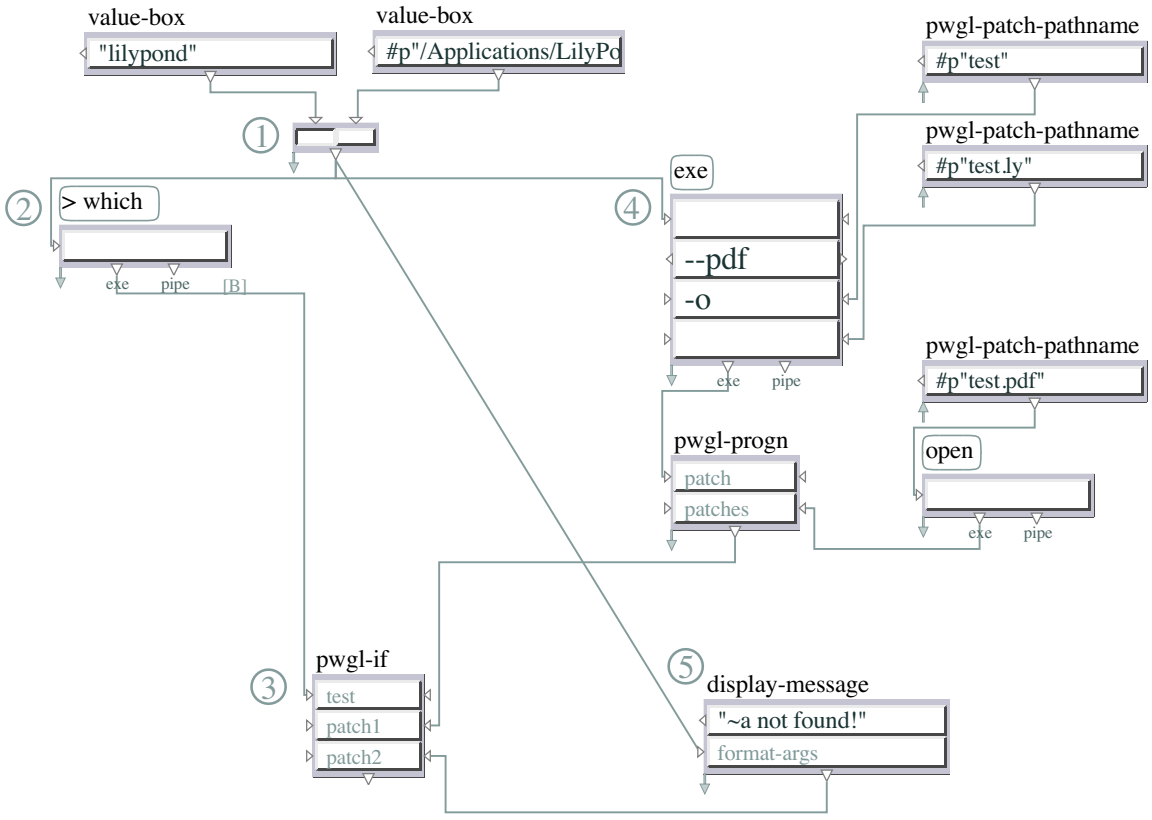

Figure 2.70: 02a-executing

2.4.2.3.4 Executing-Script

In this patch we use the 'exe' box (5) to call a script written in either php or ruby.
The two 'which' boxes (1) are first used to determine if one or both of the scripting languages are present in the system.
(2) if both scipting languages are found preference is given to ruby. This can be changed by changing the order of the input goin in the 'pwgl-or' -box
(3) names the script files (test.php and test.rb)
The two 'open' boxes (4) can be used to view and edit the scripts
Finally, by evaluating the 'text-box' at the bottom of the patch, the result of the script can be seen. These scripts just report the current installed version of the selected scripting language.

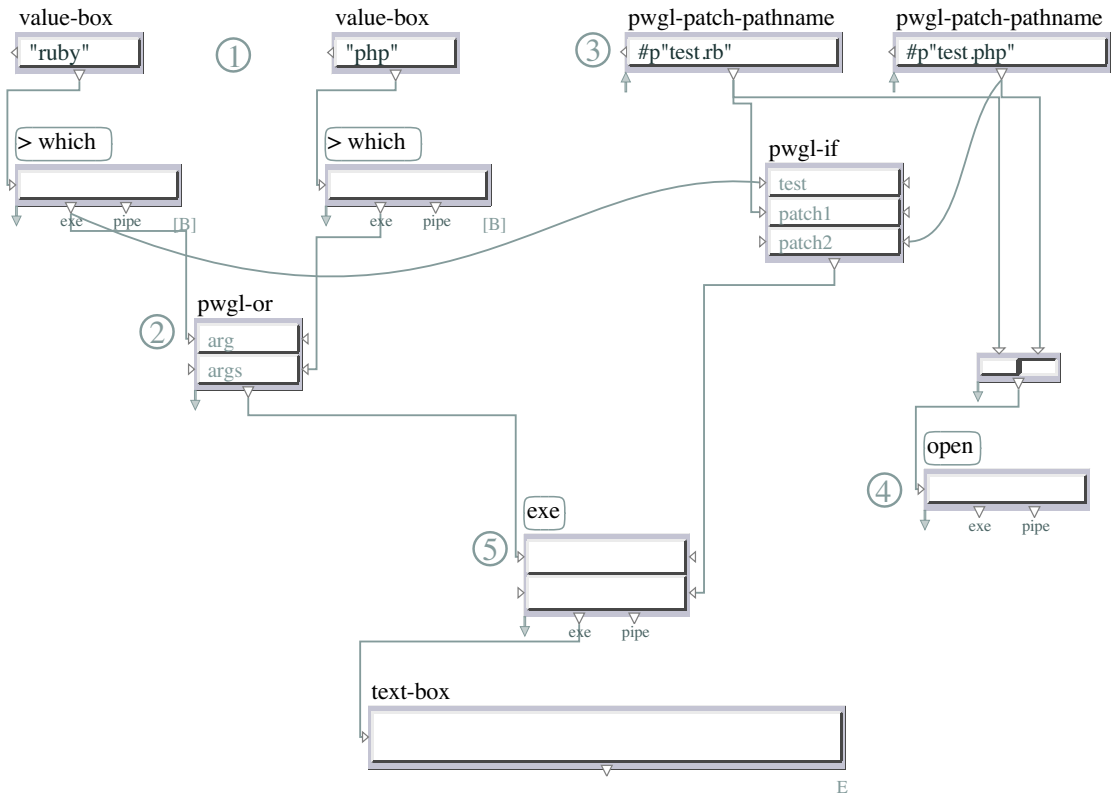

Figure 2.71: 02b-executing-script

2.4.2.3.5 Redirection

This example requires internet connection.
(1) 'curl' is a tool to transfer data from or to a server, using one of the supported protocols (HTTP, HTTPS, FTP, FTPS, GOPHER, DICT, TELNET, LDAP or FILE). Here the Wikipedia article of 'Shell computing' is retrieved. Here, the curl call is not executed, instead only the command line call is passed forward out of the second output.
(2) '>>' box implements the classic UNIX output redirection command. Here the result of the call in (1) is redirected to a file defined by the 'value-box'
(3) is used to open the HTML document in the default browser of the system.

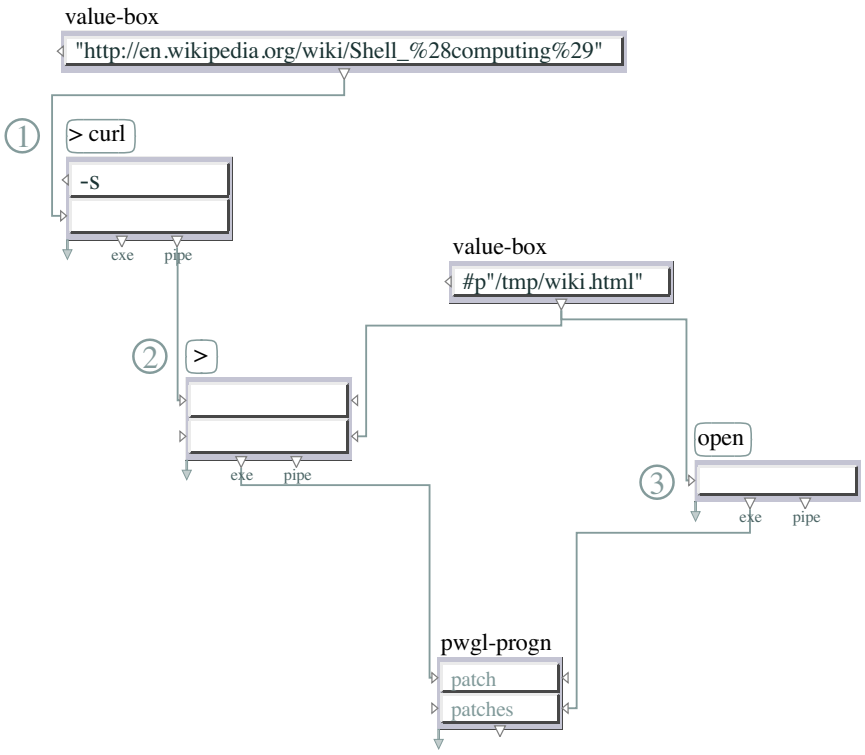

Figure 2.72: 03-redirection

2.4.2.3.6 Piping

With piping commands can be chained together.
(1) lists the contents of the users home directory
(2) the output is filtered to only contain lines which contain the string given by the 'menu-box'
(3) uses Unix sort to sort the output lexicographically
(4) the 'pipe' or '|' box is used to pipe all the aforementioned command together so that the next command uses the output of the previous command. Piping allows the commands to to be chained together to create complex commands.
Finally (5) shows the aggregate result of the piped commands.

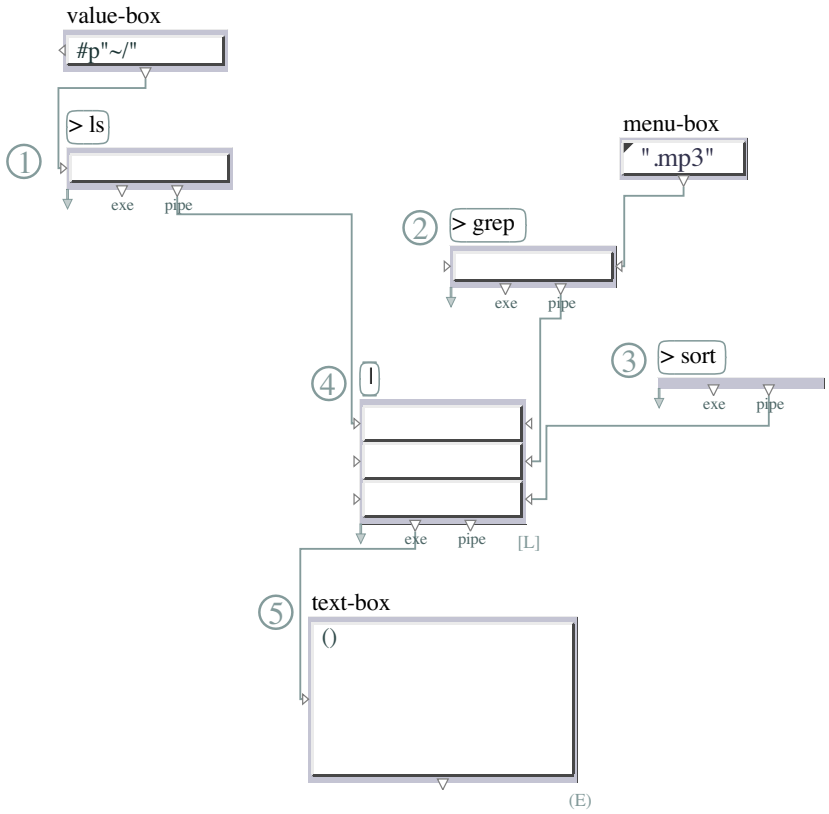

Figure 2.73: 04-piping

2.4.2.3.7 Hairy-Example

This patch demonstrates how the command line tools can be used to compile and run some C code and mix the results with ordinary Lisp boxes.
(1) If you change the c source code name given in the leftmost 'pwgl-pathname' box, you have to use 'touch' to create the file.
(2) You can open the c source code file in the default C code editor (e.g., Xcode) of your system.
(3) This box calls the UNIX C/C++ compiler and compiles the source file. The name of the output file (the product) is given with the flag -o.
(4) The 'EXE' box takes one mandatory argument which names the shell program to be run. The rest of the inputs, in turn, define the arguments to the shell program.
(5) Shows the result of the call of the box in (4).
(6) A standard 'g+' box is used to demonstrate that the output value of the shell-box can be used in Lisp as an argument to ordinary PWGL boxes.
By default the 'switch-box' in (7) forces the compilation of the C code every time the patch is evaluated in (6). This can be changed by selecting the second input of the 'switch-box'.

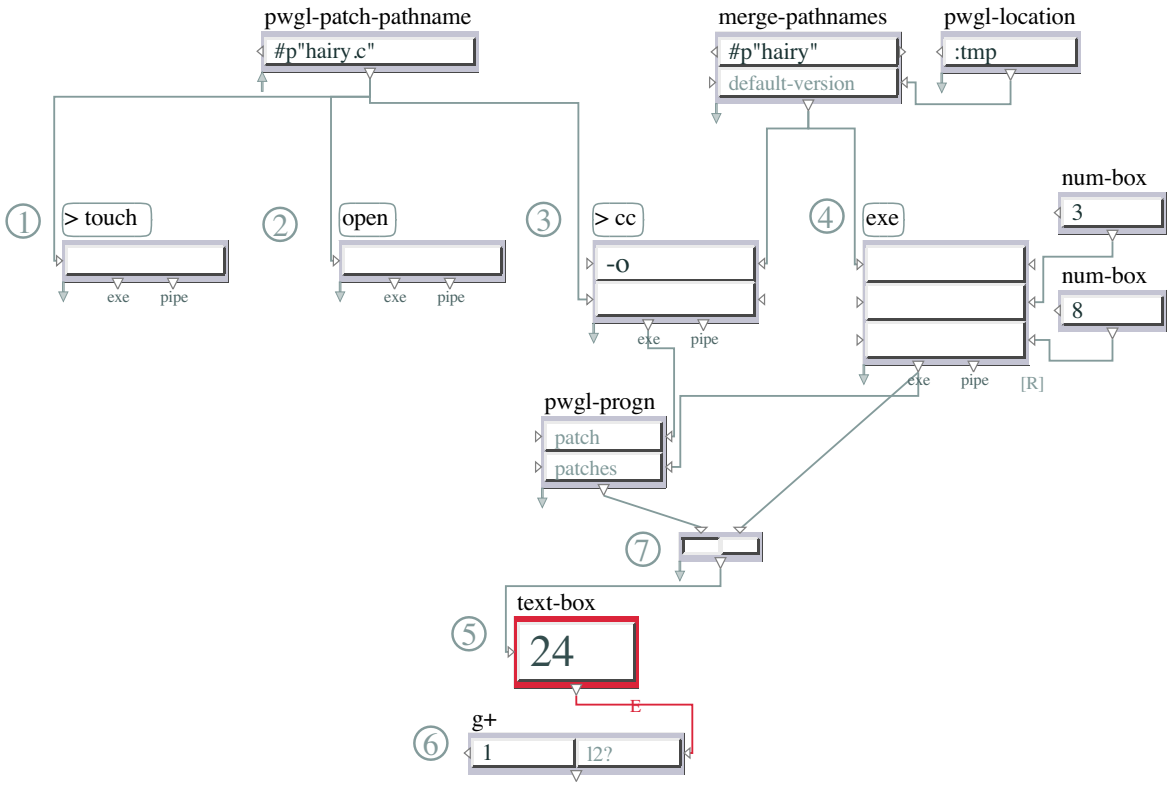

Figure 2.74: 05-hairy-example

2.4.2.3.8 Scripting

Here, we use 'grep' (1) to implement a simple file search script that displays a list of all files whose contents match a given string.
(1) Gives the location of the current patch.
(2) 'find' box is used to find all files in the directory that match the criteria, i.e., the file name ends with ".pwgl".
The grep box (3) is then used to search the files for the string shown in (4).
Finally the results are passed to a GUI box that prompts for a file name which is then opened for the user.

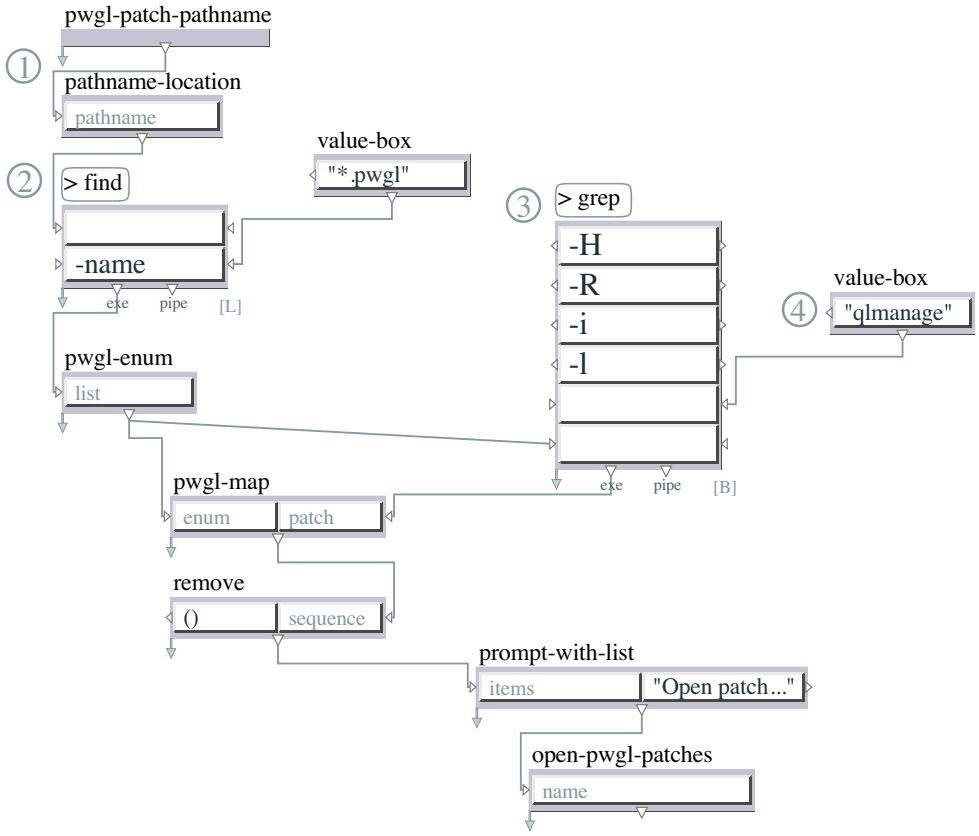

Figure 2.75: 06-Scripting

2.4.3 Code-Box

2.4.3.1 Introduction

This tutorial section presents a special PWGL box type, called code-box, which can be seen as an extension to our already elaborate collection of visual boxes. The code-box allows the user to harness the full power of the underlying Lisp language yet preserving the visual nature and functionality of a normal PWGL box. It allows the user to effectively express complex control structures, such as loops, thereby hiding those details from the patch. While the user writes the code in a text editor, the code is simultaneously analysed. This analysis consists of basic syntax checks and extraction of free variables and function names that result in a parameter list of the final Lisp expression. This scheme provides the main interface to PWGL and allows the user to access information from the visual part of the system. The appearance of the box is calculated automatically based on this analysis. By default the code-box is called 'code-box'. This name can be changed by the user. In order to distinguish this box from the ordinary ones, there is a label 'C' at the low-right corner of the box.

2.4.3.2 MIDI-List-to-Score

The target is to convert raw midi data into a more readable form before inputting the data into a score.
In (1) a list of chords (given as midi values) are passed to a code-box (2) 'construct-chords'. Here, using Lisp code, we create note and chord objects using the ENP-score-notation format. All notes that have pitch values below middle-C (60) are assigned to the lower bass-clef. We also use a piano-staff that will be seen in the final score and control the timing of the chords.
The result of the code-box calculation is given to the 'enp-constructor' box (3) that creates the final score object out of the ENP-score-notation format. The final result can be seen in (4).

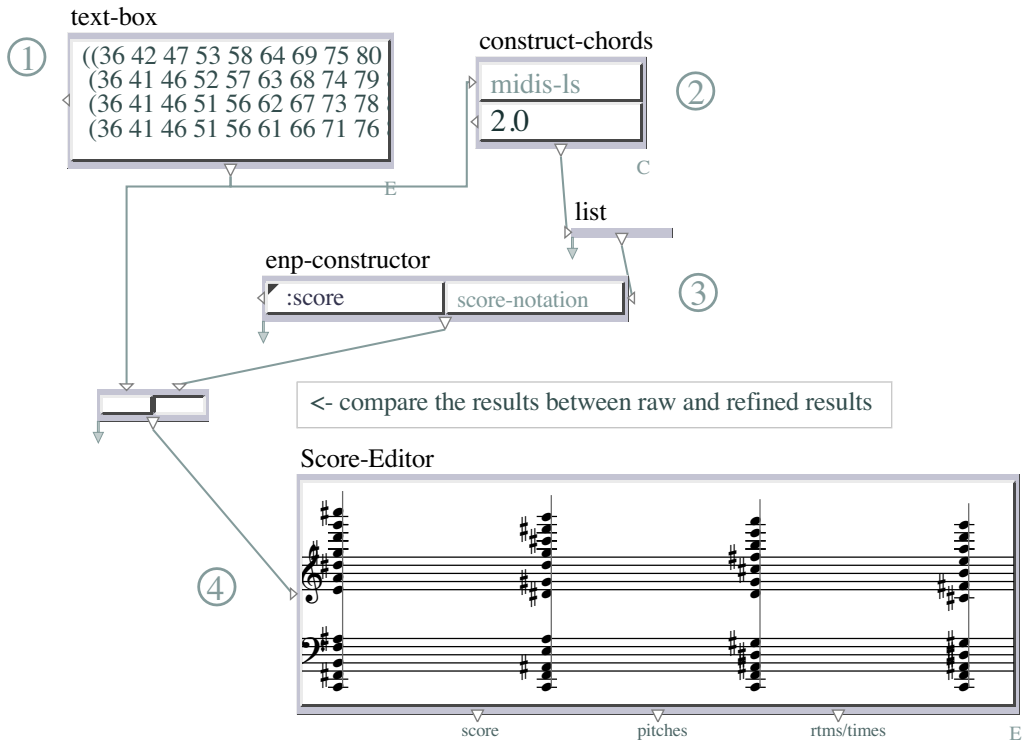

Figure 2.76: 01-midi-list-to-score

2.4.3.3 Create-Bpfs

Here we give two simple examples how to create a bpf object using the code-box.
In (1) we create a bpf with random y-values using three arguments: no-points, and low and high limits.

In (2) the y-values of a bpf (first argument) is inverted around an axis (second argument).

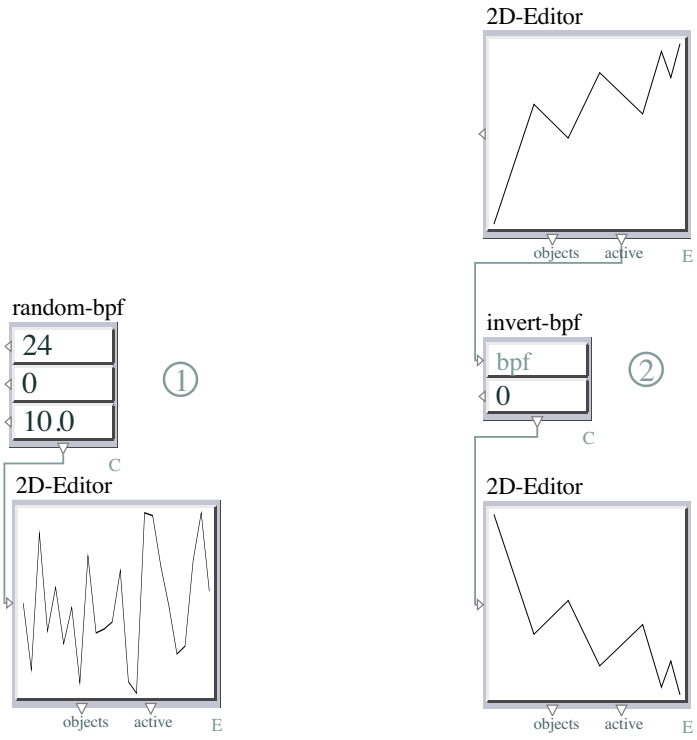

Figure 2.77: 02-create-bpfs

2.4.3.4 PMC-Examples

In this tutorial we use the code-box to solve some classical combinatorial problems.
In (1) and (2) we calculate subsets using two different approaches. In (1), 'subsets1', we simply check that all elements in a potential subset are in an ascending order. This scheme works if there are no duplicates in the incoming list and the elements are numbers (i.e. the 'list' argument is a set consisting of numbers). In (2), in turn, we operate with indices. This approach is more general and works even for the non-numerical elements and the 'list' argument can contain duplicates.
In (3) we generate all possible pitch-class (PC) supersets that contain the 'list' argument as a subset. This is done by generating a search-space out of all pitch-classes that are not contained in the 'list' argument.
Finally, in (4) we use PMC to make interval statistics out of a chord ('midis'). We collect all 2-note combinations. After this we make the final interval statistics.

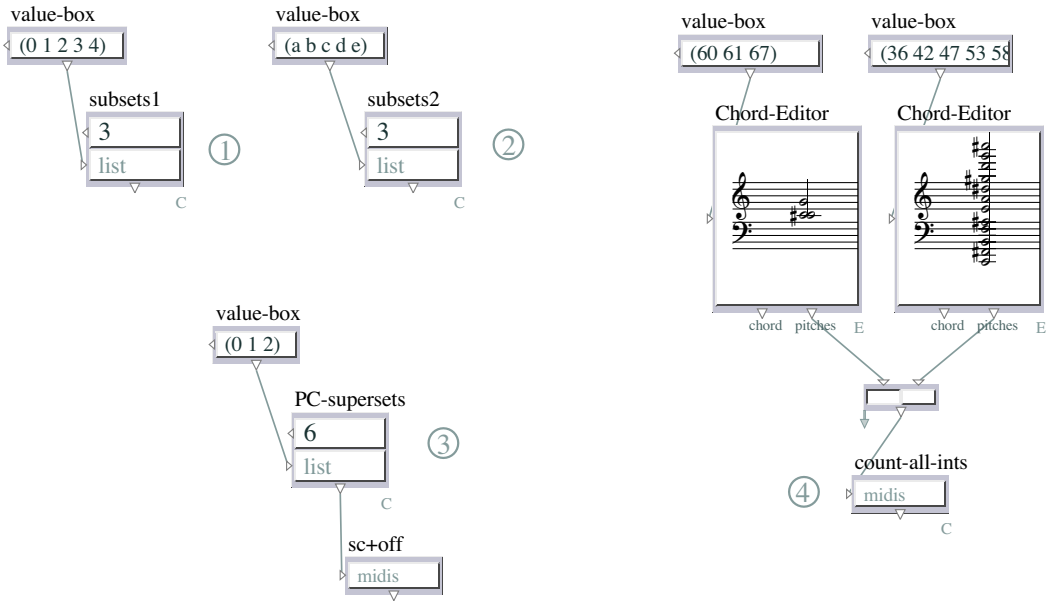

Figure 2.78: 03-PMC-examples

2.4.3.5 Transpose-Chords-V2

This patch is equivalent to the patch given in Editors/Score-editor/transpose-chords, except we use here the code-box to define chord sequences.
The pitches are calculated by combining two chords (’static’ and ’transp’). The static chord is kept untransposed while the transposed one is transposed with intervals ranging from 0 to 12. The user can choose (by selecting one of the options of the switch box called ’a tempo/acc’) whether the resulting chord sequence will have static delta-time values or whether the sequence forms an accelerando gesture. The non-mensural result is shown in the ’Score-Editor’ box.

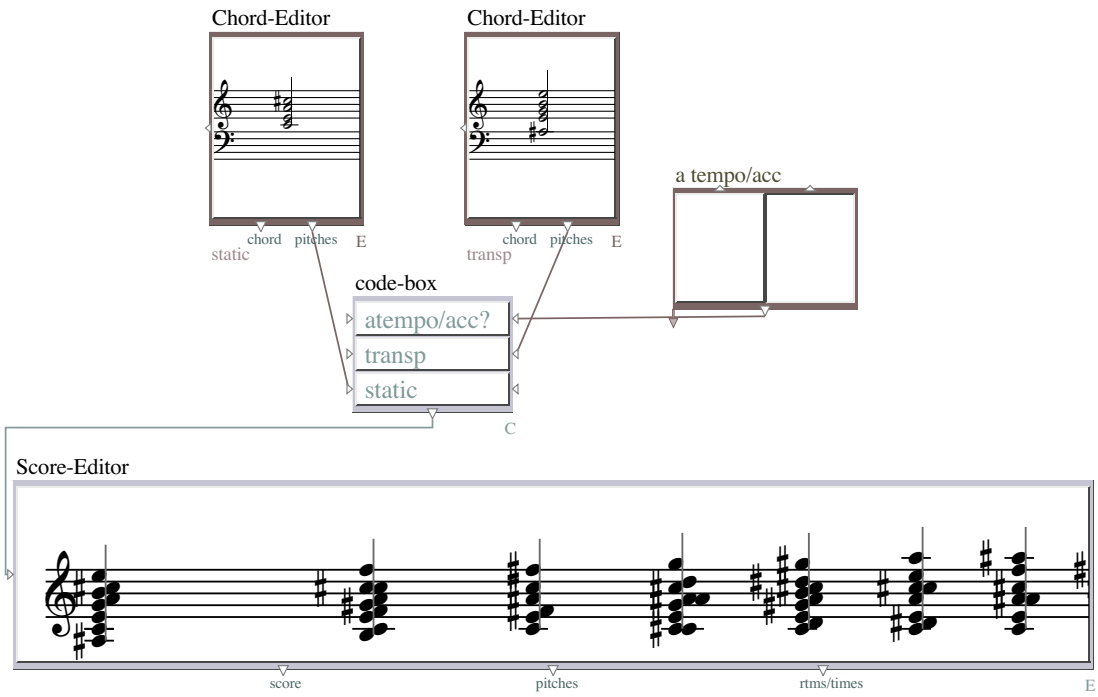

Figure 2.79: 03a-transpose-chords-V2

2.4.3.6 Function-Argument

This patch gives a simple demonstration how to pass the Code-box a function as an argument. Here, we construct a number of chords and sort them according to their bass note.
The building blocks of the chord are given in (1). The note-heads define an intervall class (relative to middle-C) so that c=0, and f#=6, for example. The default intervals basically give the ingredients of a 'Webern Triad'.
The Code-box (2) generates a number of chords using the interval classes. The lowest note in the chord will be inside the provided bass range. Also, the maximum number of chords and the maximum number of notes in each chord can be defined.
The sorting function is given in the menu-box (3).
As an exercise, the minimum number of chords and notes could be added.

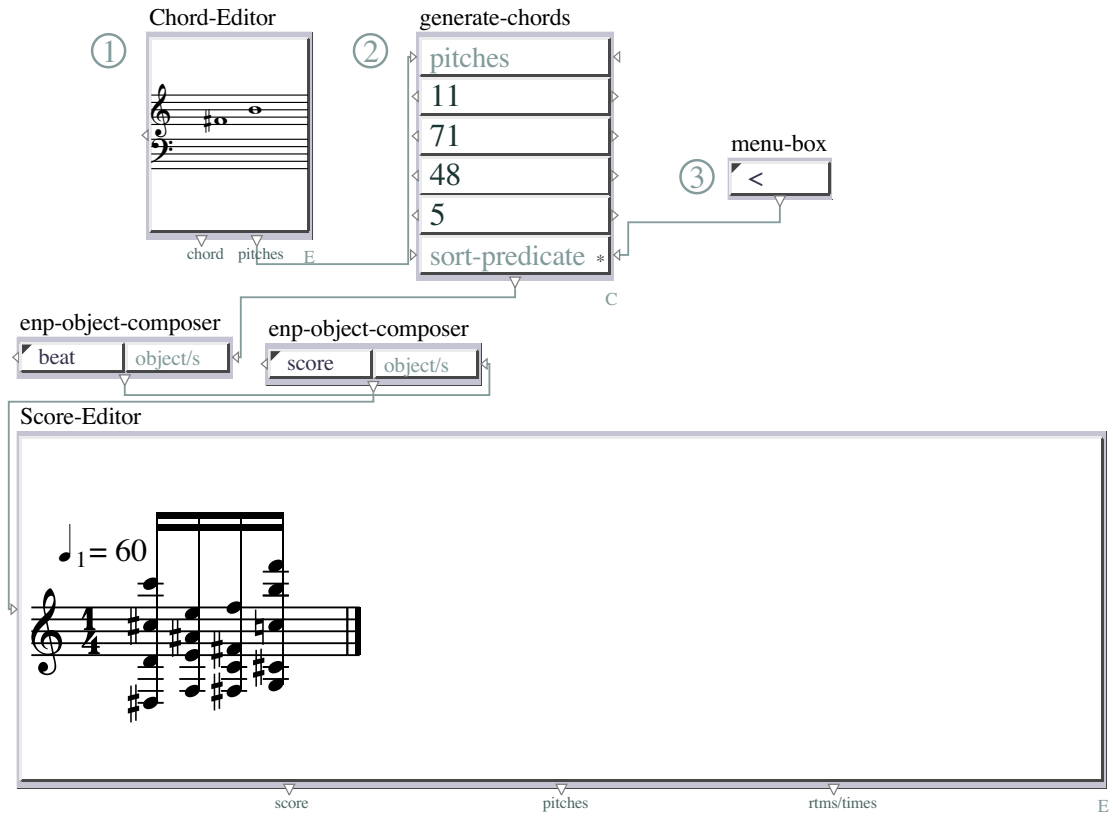

Figure 2.80: 10-function-argument

2.4.3.7 Series-Filter

In this patch we use the Code-box to generate an overtone series (1) where all the occurences of members of a given pitch-class set (2) are highlighted. Furthermore, notes whose pitch does not belong to the given set are muted (velocity is set to 0).

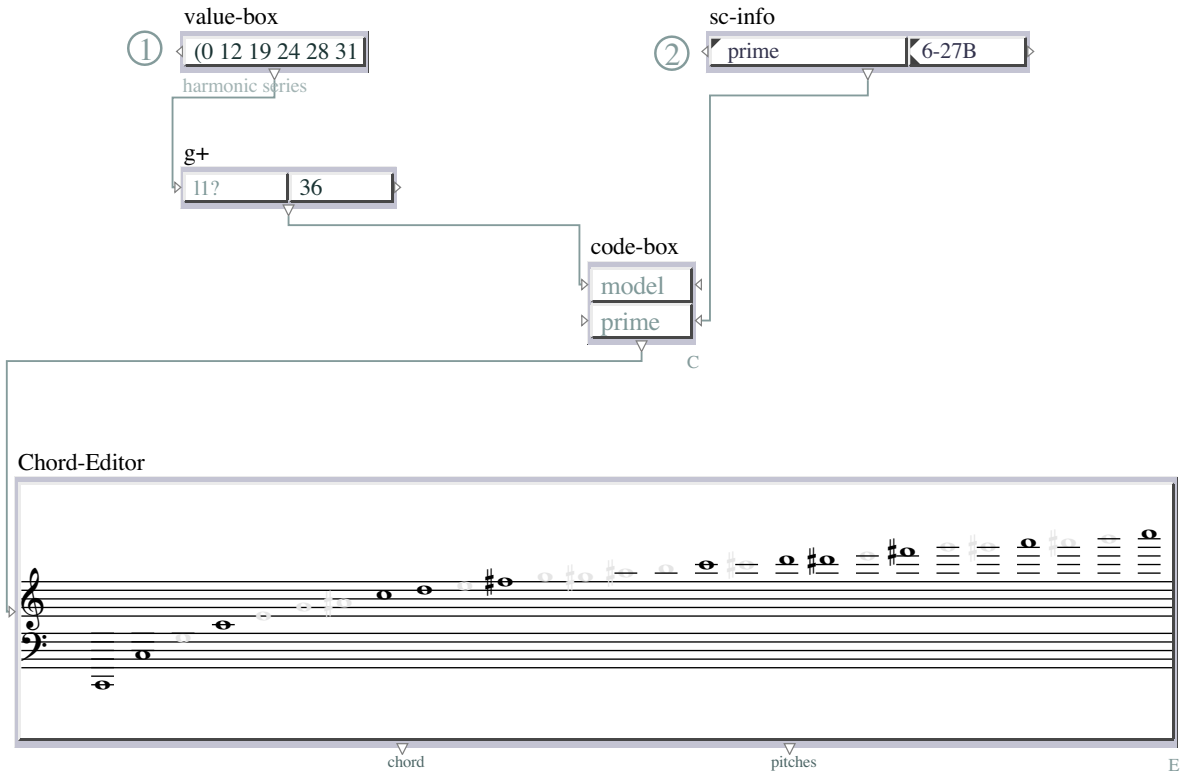

Figure 2.81: 20-series-filter

2.4.3.8 Multi-Eval

This patch demonstrates a free variable type used by the chord-box, called 'multi-eval' variable. This variable should have a name that starts with the character '!'. A multi-eval variable will be re-evaluated each time it is encountered in the expression. This scheme can be used to dynamically extract new values from PWGL boxes.
In (1), the second input, '!rnd', is connected to a 'g-random' box and this box will be evaluated 'count' times. This will result in a list 50 random values.
In (2), the '!ys' input evaluates a more complex patch consisting of several PWGL boxes. Each evaluation will generate a list of y-values that in turn will be used inside the code-box to generate 'count' bpfs.

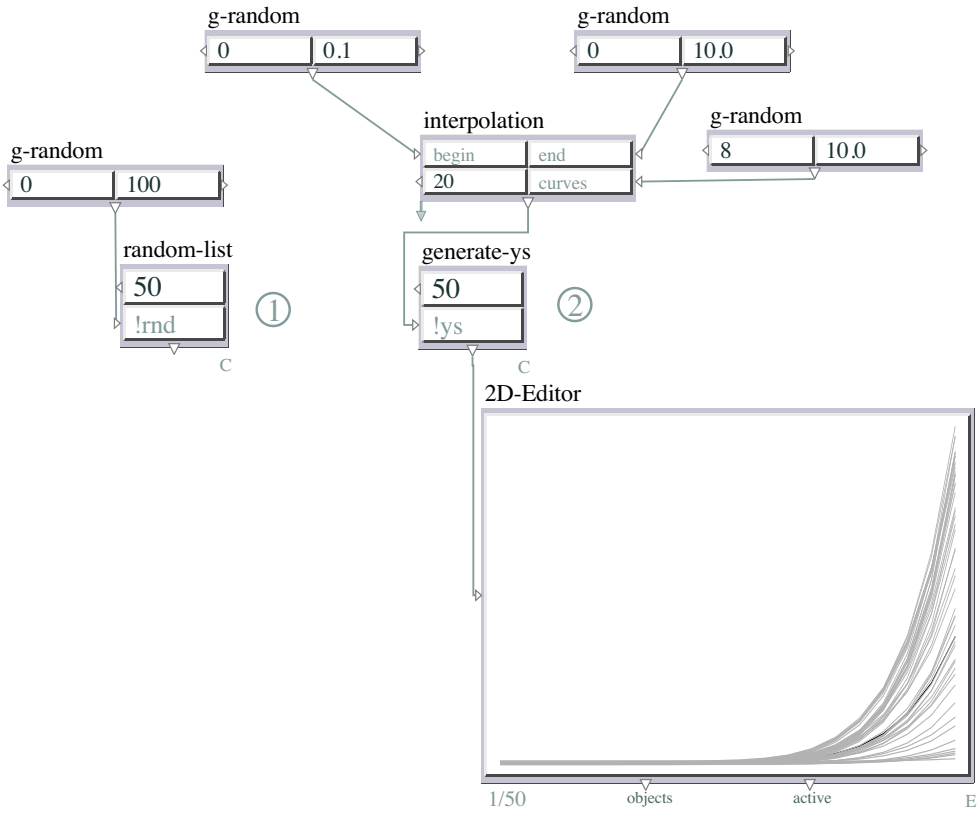

Figure 2.82: 30-multi-eval

2.4.4 Frame-Box

The frame-box is a special documentation/ presentation box. It does not have any inputs or outputs. Instead, Its purpose is to mark a subsection or several subsections in the patch. The frame-box is displayed as a rectangle that has an identity (the box string on the upper left corner) and documentation (the user string at the bottom of the box). When a frame-box is selected only the boxes enclosed by it remain in focus and the others are dimmed. By typing 'n' or 'N' it is possible to cycle through the frame-boxes in the order specified by its box string. The box string can be changed by typing 1-9. The documentation string can have line-breaks and it is automatically wrapped to the width of the frame-box.
The frame-boxes can be 'locked' to their current position by typing 'l' for lock.
This patch illustrates the use of the frame-box. Here, a small presentation is prepared with the help of the frame-box. It consists of three steps (1-3). You can test the patch by selecting the first frame-box marked as (1) and then typing 'n' several times. You can return to the 'normal' display mode by deselecting the active frame-box.

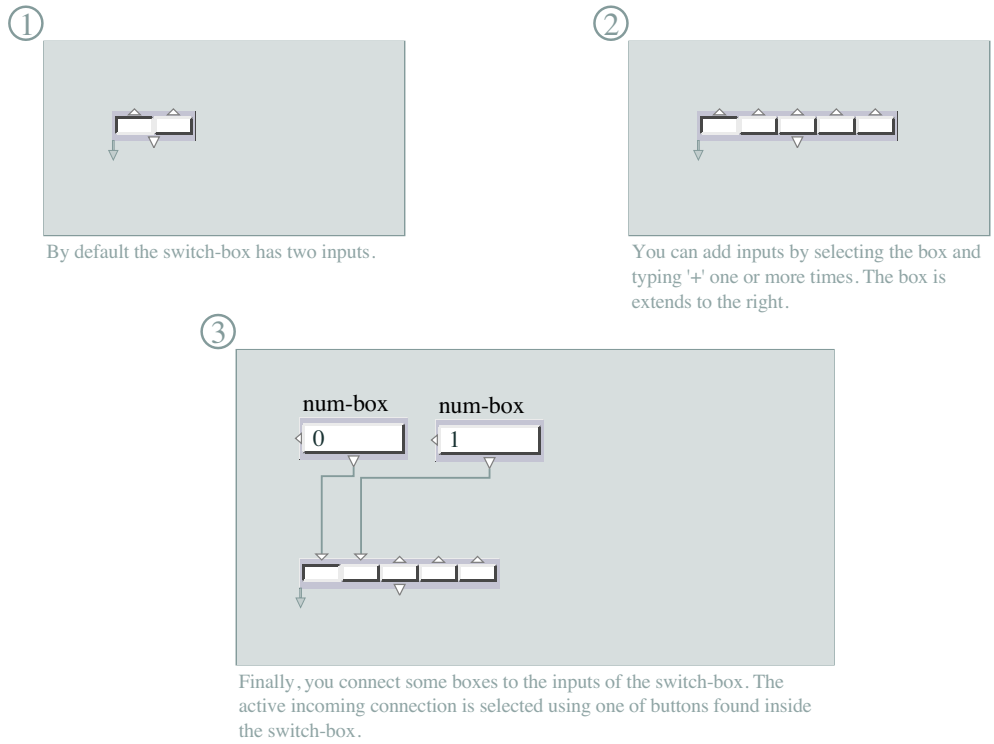

Figure 2.83: 40-frame-box

2.5 Constraints

2.5.1 Introduction

When using a procedural programming language, such as C or Pascal, or a functional language, like Lisp, typically the user has to solve a problem in a stepwise manner. In many cases this approach is an adequate one, but for many types of problem it may lead to programs that are difficult to design or understand.
Descriptive languages, such as Prolog, offer an alternative way to look at this problem: instead of trying to solve a problem step-by-step, the user describes a possible result with the help of a set of rules. It is then up to the language to find solutions that are coherent with the descriptions. This approach is probably more natural for individuals with a musical background. A typical music-theoretical writing offers a discussion on some properties of some pieces of music, not a step-by-step description of how those pieces were made.
PWGLConstraints (Laurson 1996, Laurson and Kuuskankare 2005) can be thought of as a descriptive language. PWGLConstraints is written in Common Lisp and CLOS. When using it we do not formulate stepwise algorithms, but define a search-space and

produce systematically potential results from it. Typically we are not interested in all possible results, but filter (or, rather, constrain) these with the help of rules describing an acceptable solution.

2.5.1.1 Main Components

The two main components of our constraint-based system are:

(1) PMC

(2) Score-PMC

There are 2 articles related to the constraint syntax:

2.5.2 Overview

2.5.2.1 Search-Space

Normally a search is defined by:

(1) a search-space and

(2) a set of rules.

This page gives some search-space examples. The results can be affected (even without rules) by choosing an appropriate search-space (see especially 3, 4 and 5). See also the next patch called 'search-space' in this tutorial

2.5.2.1.1 Search-Space Examples

(1) a search-space with 3 variables each with a domain of 3 items:

```
V1   V2   V3
60   60   60
62   62   62
64   64   64
```

(2) 4*8 search-space:

```
V1   V2   V3   V4
60   60   60   60
62   62   62   62
64   64   64   64
65   65   65   65
67   67   67   67
69   69   69   69
71   71   71   71
72   72   72   72
```

(3) preference ordering (the results have a tendency to ascend):

```
V1   V2   V3   V4
60   64   67   72
62   65   69   71
64   62   65   74
65   67   71   69
67   60   64   75
69   69   72   67
71   71   62   77
```

(4) moulding a search-space (the results are forced to ascend):

```
V1   V2   V3   V4
60
62   62
64   64   64
     65   65   65
     67   67   67
          69   69
          71   71
               72
               74
```

(5) constraining individual notes (all solutions will have 72 as a third element:

```
V1   V2   V3   V4
74   67   72   62
67   64        65
65   72        69
75   62        77
64   71        67
60   74        60
71   69        71
62   65        75
```

(6) 5*11 search-space with random ordering:

```
V1   V2   V3   V4   V5
74   67   74   62   74
67   64   69   65   77
65   72   67   69   60
75   62   77   77   62
64   71   60   67   65
60   74   75   60   75
71   69   72   71   67
62   65   62   75   69
69   75   64   72   64
72   60   65   74   72
77   77   71   64   71
```

2.5.2.2 Search-Space

This patch demonstrates the effect of different search-spaces when used in conjunction with a Multi-PMC box. Note that we do not use here any rules. In all cases we calculate all solutions (i.e. cartesian product, see the Multi-PMC box where the second input in row three is ':all').
In (1) we have a simple symmetric search-space that produces (3*3*3)=27 solutions, and in (2) we have a larger 4*8 example with (8*8*8*8)=4096 solutions.
In (3) we have search-space that has a tendency for the first solutions to produce ascending results. In (4) this tendency is much stronger.
In (5) the search-space contains at the third position a one-element list (72). This has an interesting effect as all solutions will have 72 as the third element.
Finally, in (6) all search-spaces can be randomly reordered by setting the first input of the third row to 'T'. This is often desirable-especially when using later the constraints system with rules-as randomness has a tendency of producing solutions that differ from each other.

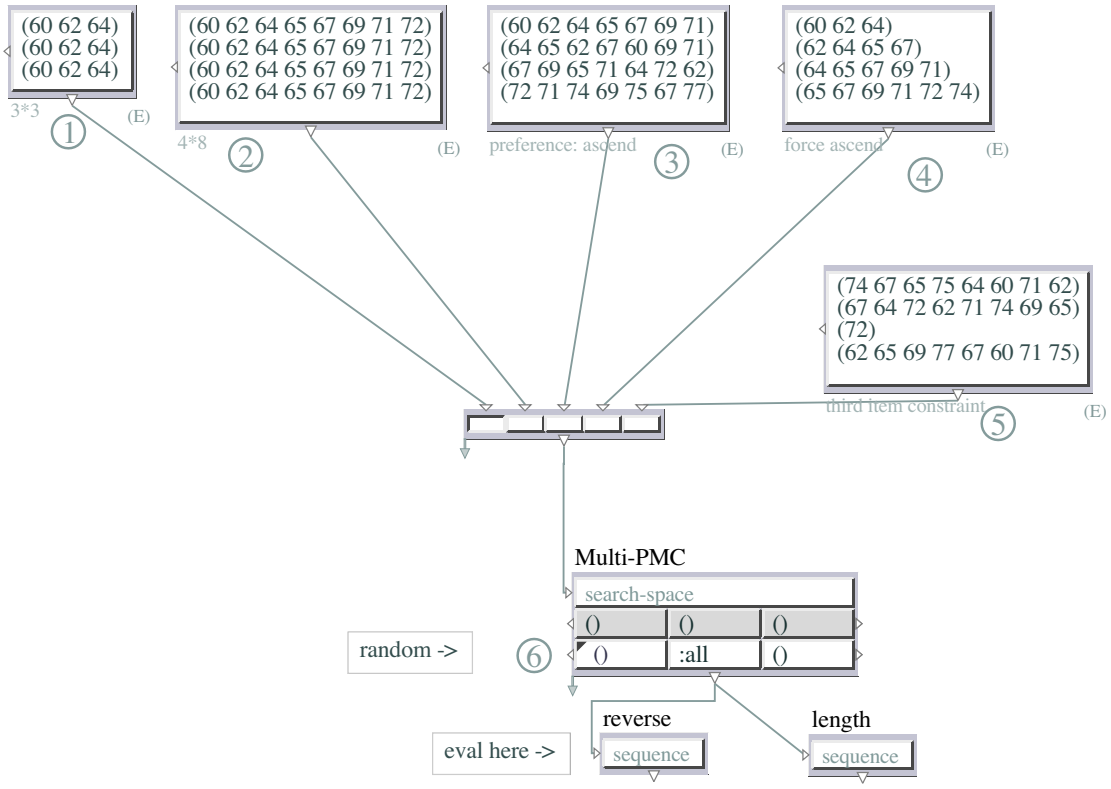

Figure 2.84: 02b-search-space

2.5.2.3 PM-Syntax

2.5.2.3.1 PMC Rule Structure

A PMC rule consists of 3 main parts:

(1) a pattern-matching part (PM-part)

(2) a Lisp-code part

(3) a documentation string.

A rule uses the PM-part to extract relevant information from a potential solution. This information is given to a Lisp test function that either accepts or rejects the current choice made by the search-engine.
Here are the main components of the PMC syntax:

2.5.2.3.2 PM-Part

?1 = variable * = one or two wild cards i1 = index-variable ? = anonymous-variable

2.5.2.3.3 Lisp-Code Part

(?if <test>) = begins a Lisp expression l = partial solution rl = reversed partial solution len = length of the partial solution

2.5.2.3.4 Pattern Matching Examples

```
input:   (1  2  3  4  5)
pattern: (*          ?1)
match:    * = (1  2  3  4), ?1 = 5

input:   (1  2  3  4)
pattern: (*    ?1 ?2)
match:    * = (1  2), ?1 = 3, ?2 = 4

input:   (1  2  3  4  5  6  7)
pattern: (?1 *          ?2 ?3)
match:    * = (2 3 4 5), ?1 = 1, ?2 = 6, ?3 = 7

input:   (1  2  3  4  5)
pattern: (i1 i2      i5)
match:   i1 = 1, i2 = 2, i5 = 5


input:   (1  2  3  4  5  6  7  8  9)
pattern: (  i2 i3           i7    i9)
match:   i2 = 2, i3 = 3, i7 = 7, i9 = 9
```

2.5.2.4 PM-Syntax

This patch shows visually how some typical PM examples match a sequence of notes. There are six examples that are defined as scripting rules in a text-box (1). Note especially the first line of each rule (i.e. the PM-part).
This patch and several forthcoming patches in this section contain rules with the Lisp-expression (?mark ?1 ?2 ...) which is used to temporarily store the indicated objects (?1 and ?2) so that they can later be displayed in the score. This information is not saved or copied along with the score or the patch but needs to be generated again if needed.
First evaluate the enp-script box (2). This creates all possible matchings for each rule. To see the matchings double-click the 'enp-script' box. This opens a diagnostics dialog. The upper part of the dialog contains two columns. To the left we have names of all current scripting rules that have one or several matches.
When a rule is selected a list of all matching positions can be found in the right part of the dialog. When one of these positions is clicked, the exact position is shown in the input score using various drawing devices (such as circles, connected shapes, bezier functions, etc.).

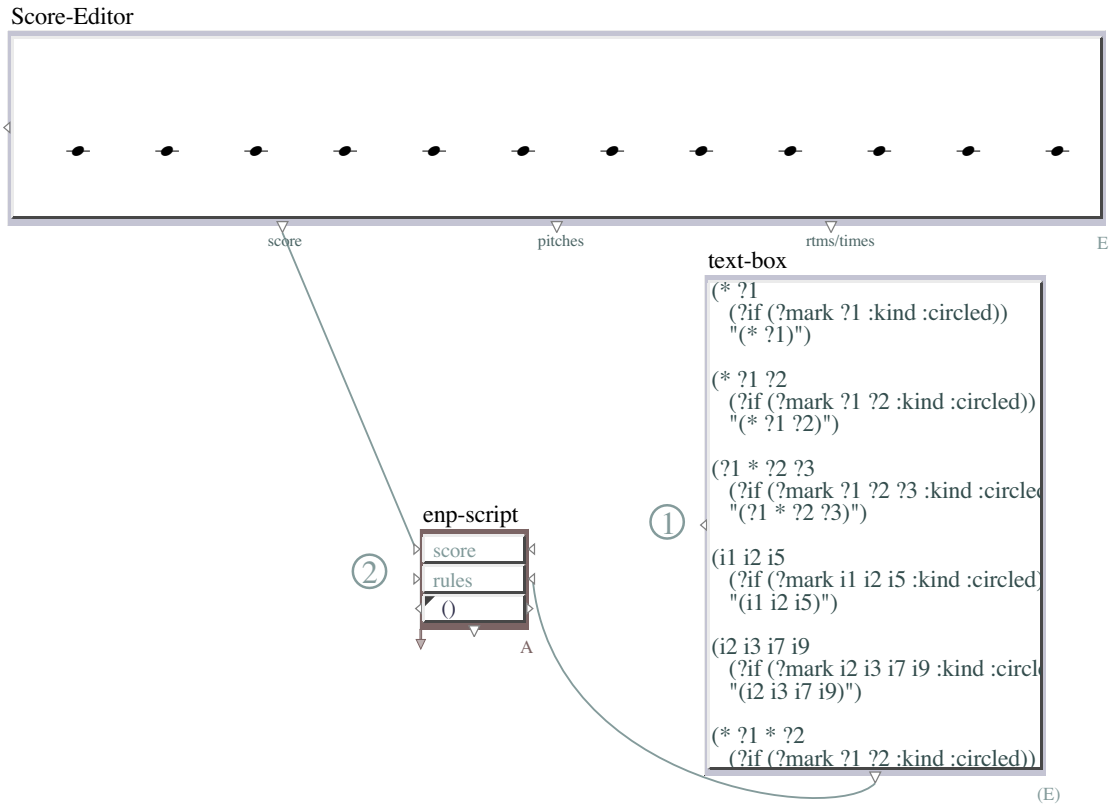

Figure 2.85: 03b-PM-syntax

2.5.2.5 PMC-Rule-Examples

Ordinary PMC rules always return a truth value (i.e. in Lisp terminology either 'T' or '()'):

```
(* ?1 ?2                    ;; PM-part
  (?if (/= ?1 ?2))          ;; Lisp-code part
  "No␣equal␣adjacent␣values")

(* ?1 ?2
   (?if (member (- ?2 ?1) '(5 6)))
   "Interval␣rule")

(* ?1
  (?if (not (member ?1 (rest rl))))
  "No␣duplicates")

(* ?1
   (?if (not (member (mod ?1 12) (rest rl) :key #'mod12)))
   "No␣pitch␣class␣duplicates")

(*
   (?if (apply #'< l))
  "Result␣in␣ascending␣order")

(i1 i2 i4 i8
  (?if (eq-SC? '(4-1) i1 i2 i4 i8))
  "index␣rule")
```

The 'PMC' section contains example patches that demonstrate how PMC can be used to solve some basic combinatorial problems.

2.5.2.6 PMC-Rule-Examples

In this patch we demonstrate how the 'Multi-PMC' solver box can be used in conjunction with six rules (1).
You can choose the current rule with the switch-box (2). To run the patch, evaluate the 'Multi-PMC' box (3). For each rule case the search space is the same (see the first input): (8* ((0_11)) or 8 times the list (0 1 2 3 4 5 6 7 8 9 10 11).
Note also that we only ask for one solution and that the search-space is in random order (the default behavior; you can change this by extending the 'Multi-PMC' box' and setting the 'rnd?' argument to ()).

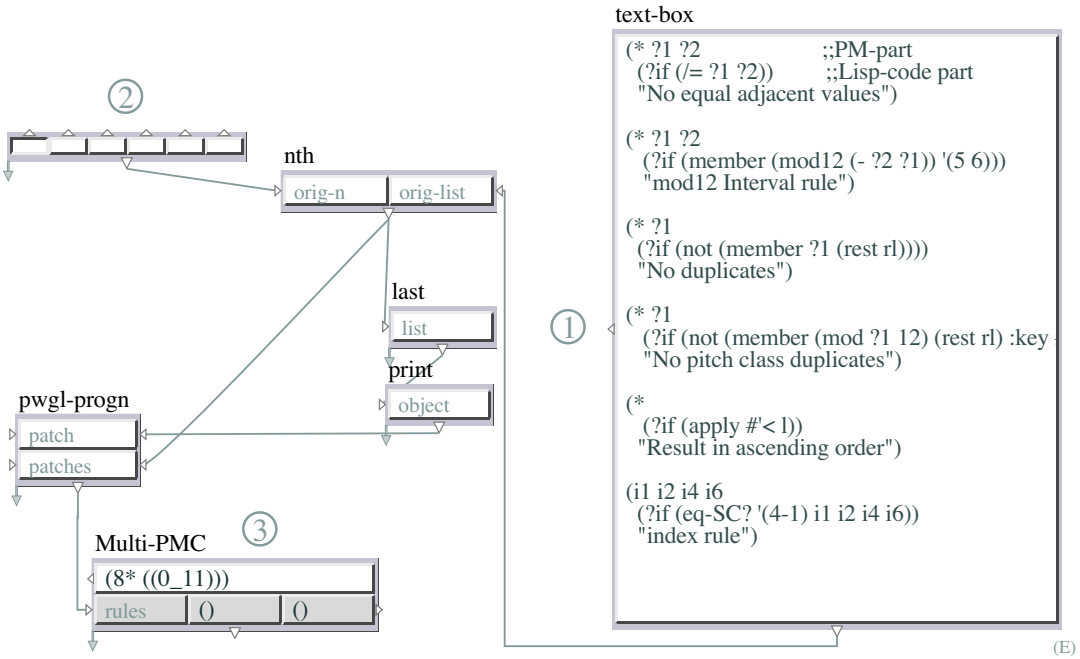

Figure 2.86: 04b-PMC-rule-examples

2.5.2.7 Heuristic-Rules

The user can also define preferences by heuristic rules. These are similar to the ordinary rules except the Lisp-code part of a heuristic rule returns a numerical value instead of a truth value (heuristic rules never reject candidates). This scheme allows the search to sort candidates that are accepted by the ordinary rules. The search tends to favour candidates with high numerical values. For instance a heuristic rule that prefers large intervals can be written as follows:

```
(* ?1 ?2
  (?if (abs (- ?2 ?1)))
  "prefer␣large␣intervals")
```

Or we can also prefer small intervals:

```
(* ?1 ?2
  (?if (- (abs (- ?2 ?1))))
  "prefer␣small␣intervals")
```

The 'Heuristic' section contains example patches that demonstrate how the user can shape the search result with the help of heuristic rules.

2.5.2.8 Heuristic-Rule-Examples

Here we demonstrate how the 'Multi-PMC' solver box can be used in conjunction with two heuristic rules (1).
You can choose the current rule with the switch-box (2). To run the patch, evaluate the 'Multi-PMC' box (3). For each rule case the search space is the same (see the first input): (8* ((0_11)) or 8 times the list (0 1 2 3 4 5 6 7 8 9 10 11).

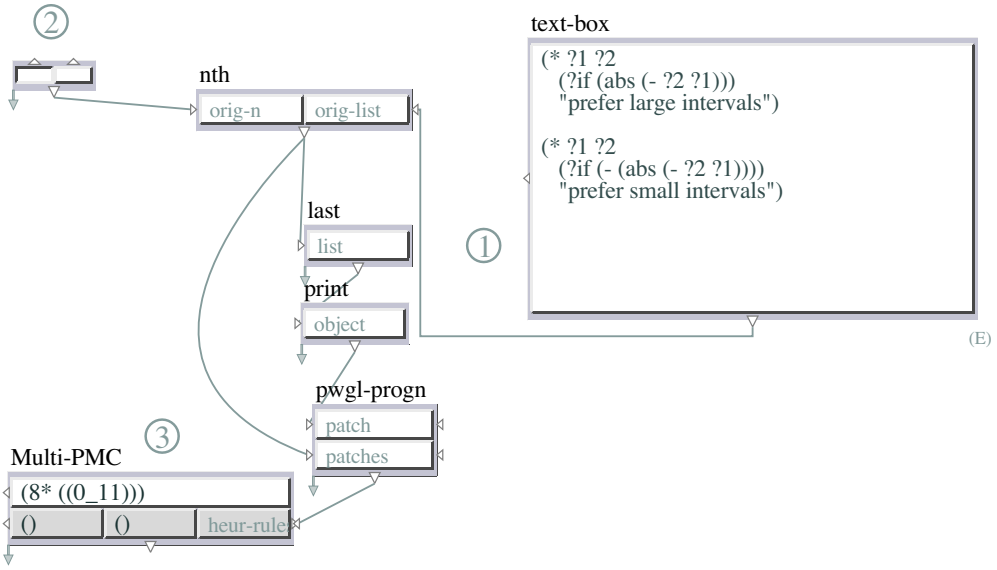

Figure 2.87: 05b-Heuristic-rule-examples

2.5.2.9 Score-PMC-Syntax

2.5.2.9.1 Score-PMC Rule Structure

The main difference between a PMC rule and a Score-PMC rule is that in the latter case variables refer to music notation related objects, such as note, chord, beat, measure, and so on. Next we discuss three main additions to the rule syntax:

(1) Accessors. The type of the object (called 'accessor') is defined in the PM-part of a rule after the variables, wild-cards and index-variables.

(2) Selectors. The second addition to the PM-part of a rule are selectors that allow to restrict the scope of a rule. Here the user can specify in more detail when a rule is run.

(3) M-method. Finally, we go inside the rule to the Lisp-code part, and discuss the third main addition to the rule system: the m-method

2.5.2.10 Accessors

The type of the score object is defined in the PM-part of a rule using accessors. A special case is the note that requires no type specification. The note and the :score-sort (see below) accessors are special as they refer to simple objects (i.e. notes), whereas all other cases refer to compound objects containing one or several notes.

2.5.2.10.1 Accessors

Currently supported accessors are:

(1) :chord

(2) :beat

(3) :measure

(4) :harmony

(5) :score-sort

2.5.2.10.2 Examples

Thus in a pattern-matching part (PM-part) without any accessor specification:

```
(* ?1 ?2 ...)
```

the variables ?1 and ?2 refer to two adjacent melodic note objects.
Compare this to a rule with the accessor ':harmony':

```
(* ?1 ?2 :harmony ...)
```

the variables ?1 and ?2 refer to two adjacent harmonic formations.
Other possible PM-part examples are:

```
(* ?1 ?2 :chord ...)
```

the variables ?1 and ?2 refer to two adjacent chords.

```
(* ?1 ?2 :beat ...)
```

the variables ?1 and ?2 refer to two adjacent beats.

```
(* ?1 ?2 :measure ...)
```

the variables ?1 and ?2 refer to two adjacent measures.

```
(* ?1 ?2 :score-sort ...)
```

the variables ?1 and ?2 refer to two adjacent notes in the score-sort ordering.

2.5.2.10.3 Accessor Test

After the accessor an optional keyword ':accessor-test' allows to define the criteria how objects of the current accessor type are chosen. This results in a more dynamic behavior of the PM-part and it can react to the current musical context in a more interesting way than using the ordinary PM-part only.
Now the notational objects that are bound in the PM-part are for instance not necessarily adjacent in the score even when the PM-part suggest so. Thus, in the following rule we are interested in all two long note-pairs:

```
(* ?1 ?2  :accessor-test #'(lambda (note) (> (durt note) 1.0)) ...)
```

?1 and ?2 can be any note-pair in a melodic line as long as they share the criteria: both notes should be longer than 1s. Furthermore there should be no long notes between them (thus shorter notes can be found between ?1 and ?2).
In another :accessor-test' example we look for all long harmonic pairs (the duration of the both harmonic formations must exceed 2s):

```
(* ?1 ?2  :harmony :accessor-test #'(lambda (harm) (> (durt harm) 2.0)) ..
```

2.5.2.11 Accessors1

The first accessor patch shows visually how PM examples with accessors match a two-part score. There are five examples that are defined as scripting rules in a text-box (1). Note especially the first line of each rule (i.e. the PM-part).
The five rules contain the following accessors: (1) none (i.e. melodic accessors) (2) :beat (3) :measure (4) :harmony (5) :score-sort
Note that we do not use here the :chord accessor as it would be identical to the melodic one (i.e. all chords are single note chords).
For all rules except the first one and the last one we use a selector test '(m ?2 :complete? T)' - see the next section in the tutorial - to guarantee that all notes in the compound structure are present.
First evaluate the enp-script box (2). This creates all possible matchings for each rule. To see the matchings double-click the 'enp-script' box. This opens the diagnostics dialog.

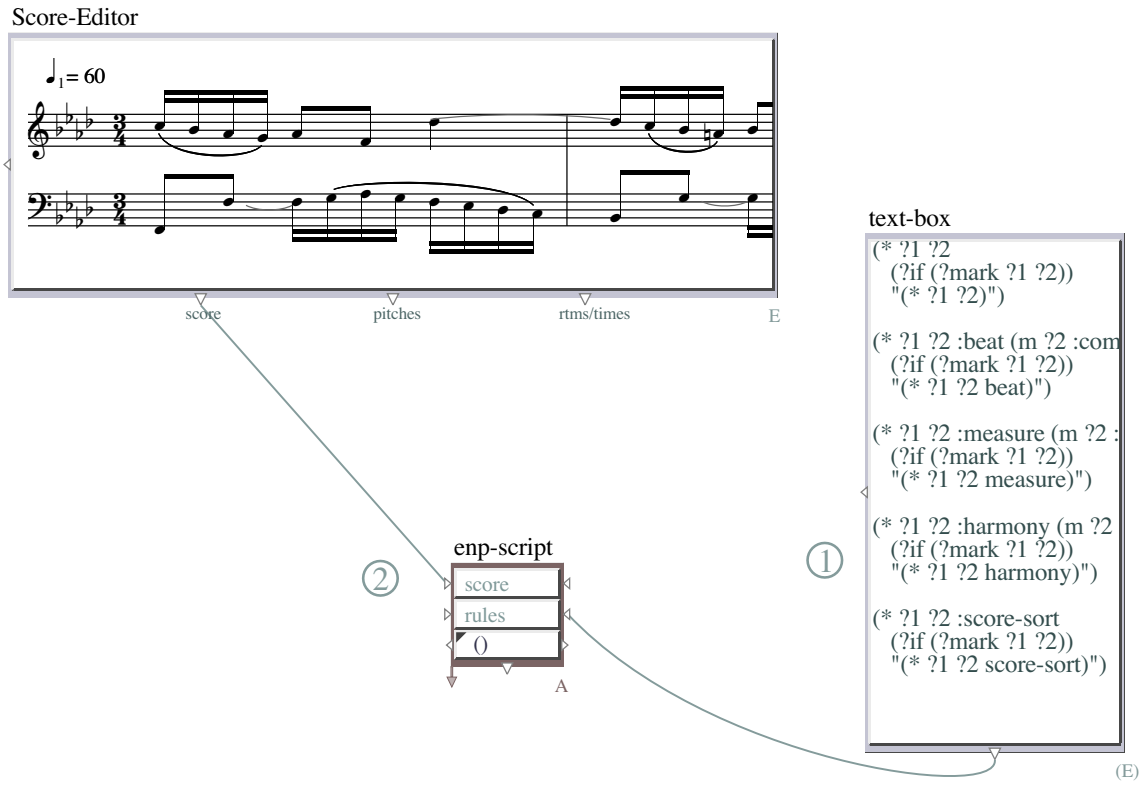

Figure 2.88: 06-Accessors1

2.5.2.12 Accessors2

The second accessor patch shows visually how PM examples with accessors match a more complex two-part score. The main difference with this score and the previous one is that now both parts contain multi-note chords. There are five examples that are defined as scripting rules in a text-box (1). Note especially the first line of each rule (i.e. the PM-part).
The five rules contain the following accessors: (1) :chord (2) :beat (3) :measure (4) :harmony (5) :score-sort
Note that we do not use here the melodic accessor as both parts contain multi-note chords.
For all rules except the last one we use a selector test '(m ?2 :complete? T)' - see the next section in the tutorial - to guarantee that all notes in a compound structure are present.
First evaluate the enp-script box (2). This creates all possible matchings for each rule. To see the matchings double-click the 'enp-script' box. This opens the diagnostics dialog.

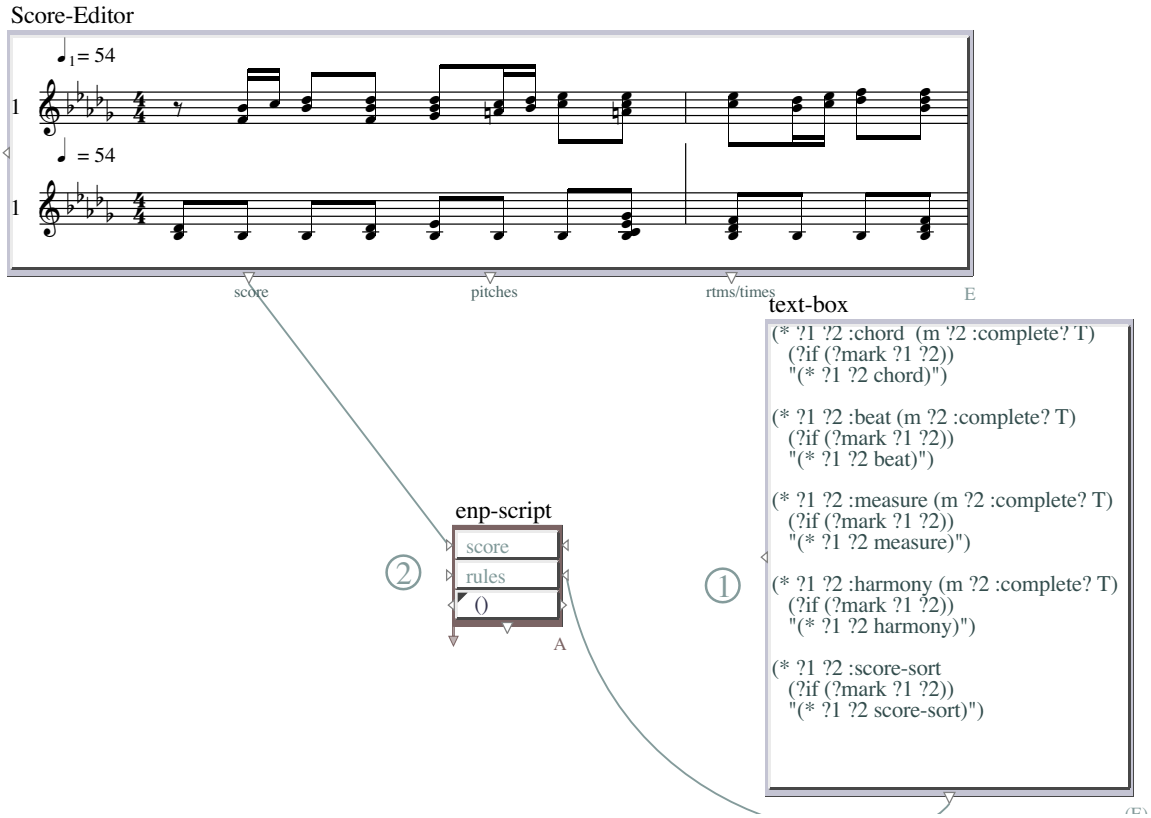

Figure 2.89: 06-Accessors2

2.5.2.13 Accessors3

The third accessor patch shows visually how PM examples with accessors match using the :accessor-test keyword. Note again especially the first line of each rule (i.e. the PM-part).
Here we have two rules (1): first, we extract two melodic note pairs that are equal or longer than an 1/8th note; second, we extract two melodic note pairs that are shorter than an 1/8th note.
First evaluate the enp-script box (2). This creates all possible matchings for each rule. To see the matchings double-click the 'enp-script' box. This opens the diagnostics dialog.

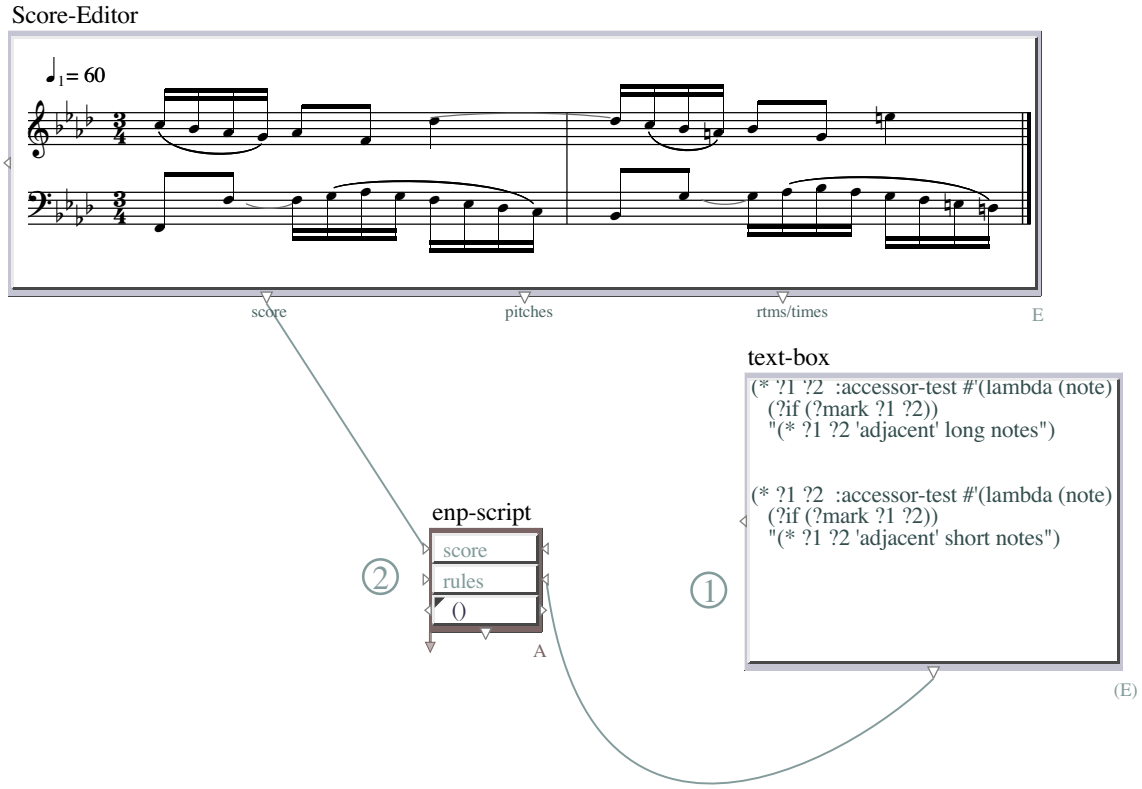

Figure 2.90: 06-Accessors3

2.5.2.14 Selectors

The PM-part may also contain one or more 'selectors' which allows further to restrict the scope of the rule. A selector is a keyword/value pair, or it can also be a Lisp expression that returns a truth value. In the latter case the rule is run only if the Lisp expression returns T.

2.5.2.14.1 Selector Keywords

Currently supported selector keywords are:

(1) :notes, number or a list of numbers

(2) :chords, number or a list of numbers

(3) :beats, number or a list of numbers

(4) :cont-beatnum, number or a list of numbers

(5) :measures, number or a list of numbers

(6) :parts

(7) number or string: <pnum>, <pname>,

(8) simple list: (<pnum1> <pnum2> ...), (<pname1> <pname2> ...), or

(9) a list of lists of part-voice specs: ((<pname1> <vname1>) (<pnum> <vname>), ...))

2.5.2.14.2 Examples

For instance in the following melodic rule (note that the rule does not have an accessor):

```
(* ?1 ?2 :parts '(1 3) ...)
```

the rule is applied only for notes that belong to parts number 1 or 3.
In the melodic rule:

```
(* ?1  :measures '(1 4) :parts '(1 2 5) ...)
```

the rule is applied only for notes belonging to measures 1 or 4 and belonging to parts 1, 2 or 5.
In the melodic rule:

```
(* ?1  :parts '(("flute" 2) 2 (3 1) (3 3)) ...)
```

the rule is applied only for notes belonging to the second voice of a part called "flute", i.e. ("flute" 2); second part, i.e. 2; and first and third voices of part 3, i.e. (3 1) and (3 3).

2.5.2.15 Selectors

This patch shows visually how PM examples with selectors match in a four-part score. Note especially the first line of each rule (i.e. the PM-part).
The five rules contain the following selectors in various combinations (1): :parts :measures :beats :notes
First evaluate the enp-script box (2). This creates all possible matchings for each rule. To see the matchings double-click the 'enp-script' box. This opens the diagnostics dialog.

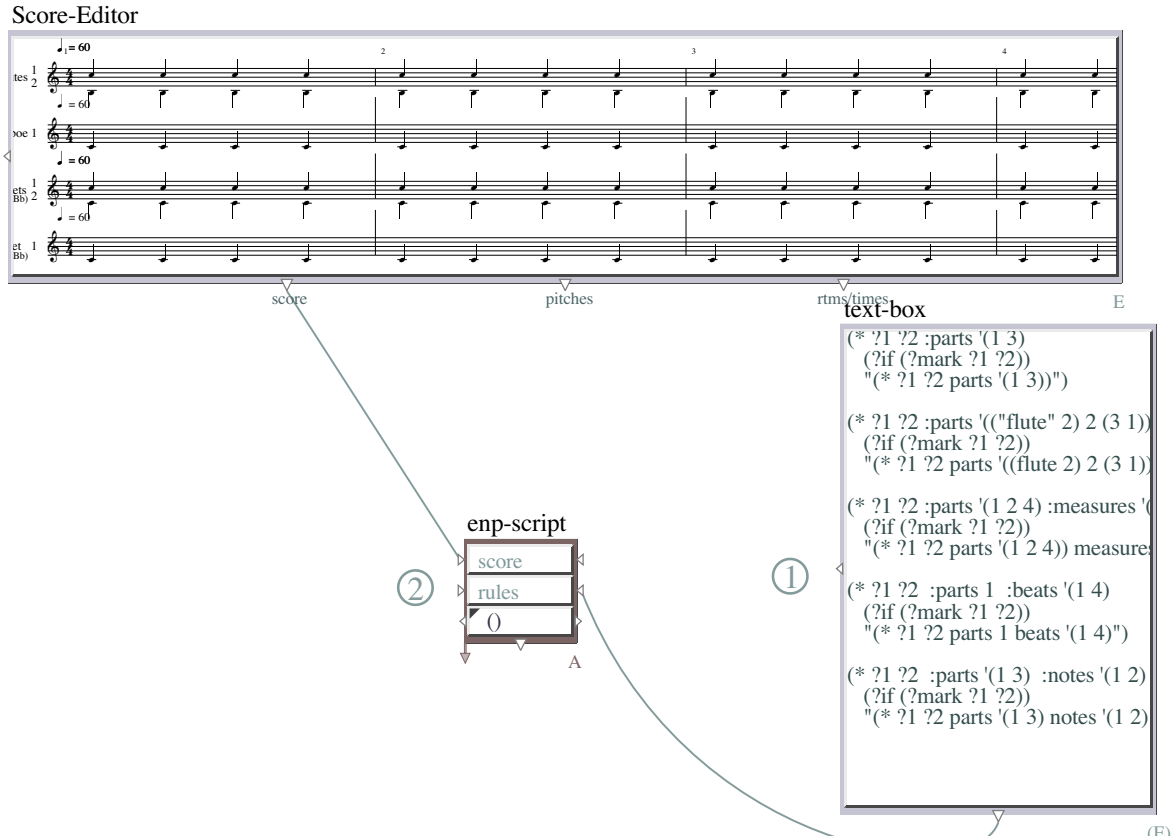

Figure 2.91: 06a-Selectors

2.5.2.16 M-Method

Once the PM-part of a rule has accessed the desired objects (i.e. notes, chords, beats, measures, harmonic formations), these variables can be used within the Lisp-code part. To access information from the variables typically the 'm-method' ('m' for 'multi-accessor') is used. Normally 'm' returns either single midi-values from non-compound objects (notes) or a list of midi-values from compound objects (chords, beats, measures, harmonic formations).
For instance in a rule:

```
(* ?1 ?2
  (?if (/= (m ?1) (m ?2)))
  "no␣adjacent␣melodic␣pitch␣dups")
```

the variables '?1' and '?2' refer to notes. The m-method will by default return the respective midi-values of the variables.
In a rule:

```
(* ?1 ?2 :harmony
  (?if (not (equal (m ?1) (m ?2))))
```

```
  "no␣adjacent␣harmonic␣pitch␣dups")
```

the m-method is used to extract two midi-value lists from two adjacent harmonic formations.

2.5.2.16.1 M-Method Keywords

The m-method accepts a list of keyword arguments that allow to specify more accurately the data that is returned. Here is an example list of supported keywords with their possible key-values:

(1) :data-access

(), :min, :max, :int, :harm-int, #'(lambda (x) <code>)

(2) :complete?

(), T

(3) :parts

(),

number or string: <pnum>, <pname>,

simple list: (<pnum1> <pnum2> ...), (<pname1> <pname2> ...),

a list of lists of part-voice specs: ((<pname1> <vname1>) (<pnum> <vname>), ...)

(4) :object

(), T, :accessor

(5) :vl-matrix

(), <count>

(6) :attack

(), T

(7) :l, :rl

(), T, <count>

(8) :l-filter

(), #'(lambda (x) <code>), #'(lambda (x res) <code>)

2.5.2.16.2 Examples

For instance a harmonic rule that disallows any interval duplicates can be written as follows (note that we use here the :data-access :harm-int pair in order to get the harmonic intervals from ?1):

```
(* ?1 :harmony
  (?if (setp (m ?1 :data-access :harm-int)))
   "no␣harm␣int␣repetitions")
```

2.5.2.17 Utility-Functions

This page lists some important Score-PMC utility functions used in this tutorial.

(1) Access the index number of ENP objects (the indexing starts from 1)

(2) notenum

(3) chordnum

(4) beatnum

(5) cont-beatnum

(6) measurenum

(7) voicenum

(8) partnum

(9) Metric rhythm

(10) downbeat?

(11) rtm-pattern

(12) Check whether an object is the first or last item of its kind in a part

(13) first?

(14) last?

(15) Timing information in seconds

(16) startt

(17) durt

(18) endt

(19) Timing information metric (as written)

(20) face-value

(21) Expression access

(22) e

(23) Misc

(24) grace-note-p

(25) matrix-access

(26) match-ART-rtms?

(27) PMC-imitation

2.5.2.18 Score-PMC-Rule-Examples

```
(* ?1 ?2                              ;;PM-part
  (?if (/= (m ?1) (m ?2))) ;;Lisp-code part
  "no␣adjacent␣melodic␣pitch␣dups")

(* ?1 ?2
  (?if (member (- (m ?2) (m ?1)) '(1 -1 2 -2)))
  "melodic␣interval")

(* ?1 :harmony
  (?if (setp (m ?1)))
  "no␣harm␣pitch␣repetitions")

(* ?1 :harmony
  (?if (setp (m ?1 :data-access :harm-int)))
   "no␣harm␣int␣repetitions")

(* ?1 ?2 :harmony
  (?if (not (equal (m ?1) (m ?2))))
  "no␣adjacent␣harmonic␣pitch␣dups")

(* ?1 :chord :parts '(1 3)
    (?if (let ((ints (m ?1 :data-access :harm-int)))
           (if ints
             (and (not (member  1 ints)) (apply #'>= ints))
              t)))
       "no␣min␣seconds␣and␣ascending␣chord␣ints␣rule,␣parts␣1,3")
```

2.5.3 Heuristic

2.5.3.1 Profile-PMC

This patch demonstrates how to use Multi-PMC in conjunction with ordinary rules and heuristic rules.
The heuristic rule is generated by the box 'mk-pmc-profile-rule' that directs the search so that the overall melodic profile tries to match the given bpf function as close as possible. There are two options in the patch: (1) If the output of the left-most 2D-editor is connected to the 'bpf' input of the 'mk-pmc-profile-rule' box, then the result is deterministic. (2) If the 'bpf' input is connected to the right hand 2D-editor containing two bpfs–defining a range of pitches–the result will vary for each search depending on the ordering of the search domains.
The search results in a list of pitches, which is given to a Score-editor. The Score-editor converts this result to a non-mensural part. The result can be played by selecting the Score-editor and typing the 'space' key.

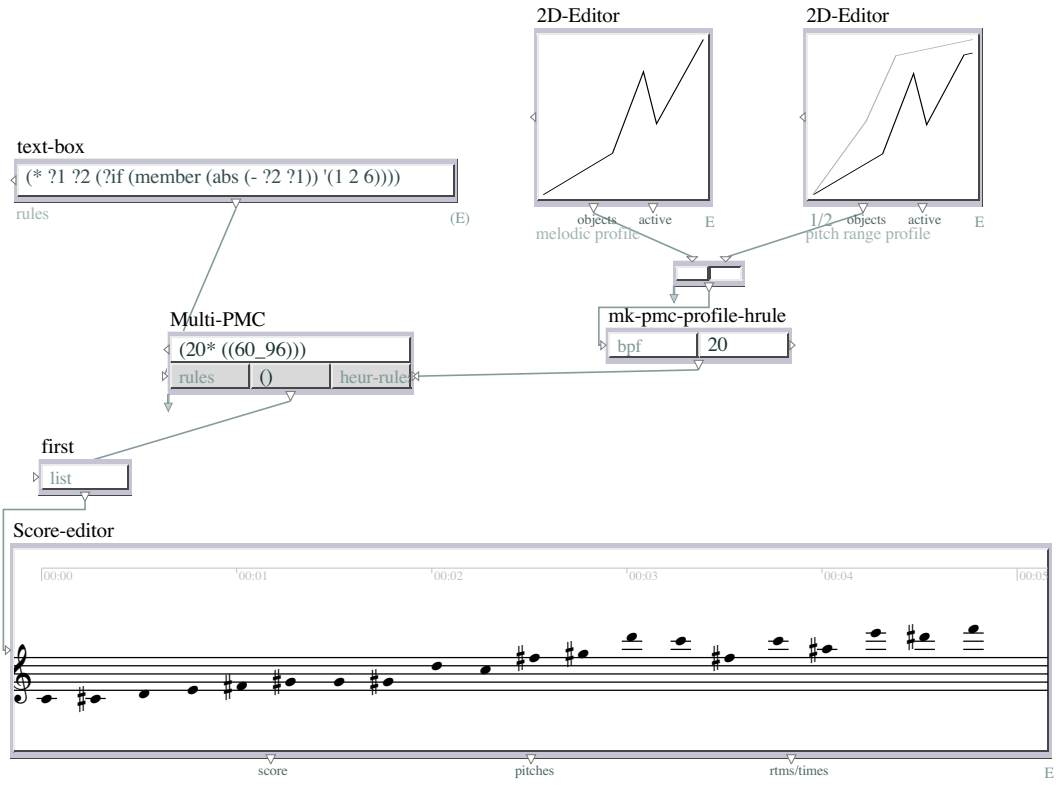

Figure 2.92: 01-profile-pmc

2.5.3.2 Heuristics-W-Menu-Box

This tutorial shows how various heuristic rules affect a result in a melodic line. A 'menu-box' contains several labeled heuristic rules (1). The actual definitions of the rules can be found by double-clicking the menu input. This operation opens a text-editor window that will show the rules and their labels in textual form.
After choosing one heuristic rule (1) the user has 2 rule options, (2) and (3), that will be used in conjunction with the chosen heuristic rule. In (2) no ordinary (or 'strict') rules are used; in (3) one ordinary Score-PMC rule is used that controls the set-class identity of each adjacent 3-note group.
To see results evaluate the 'Multi-Score-PMC' box (4).

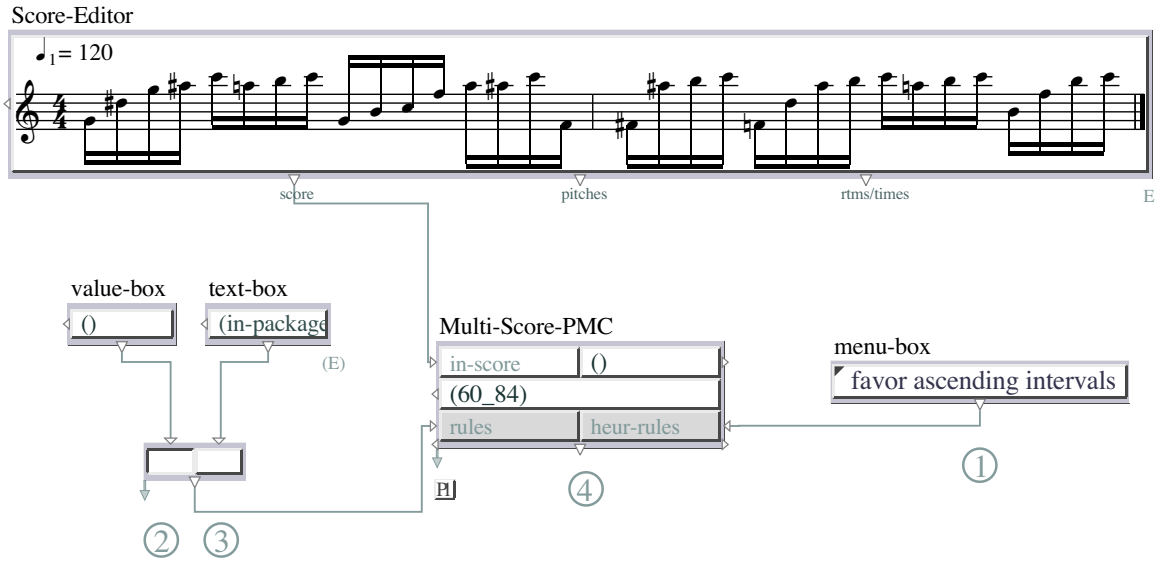

Figure 2.93: 03-heuristics-w-menu-box

2.5.3.3 Heuristics-W-Score-Bpfs

This patch demonstrates how special ENP expressions, called score-bpfs, can be used to control search results. The 2-measure score contains a score-bpf expression, that in turn contains 4 bpfs (break-point functions) that have different colors.
The user can choose a bpf color (1). This information is bound before the search to the keyword ':bpf-index'. This keyword is in turn used internally by the heuristic rule which is defined in the 'text-box' (3). Like in the previous example the heuristic rule is used in conjunction with an ordinary Score-PMC rule (4).
The search can be evaluated in (5). The result will follow the chosen bpf as close as possible.

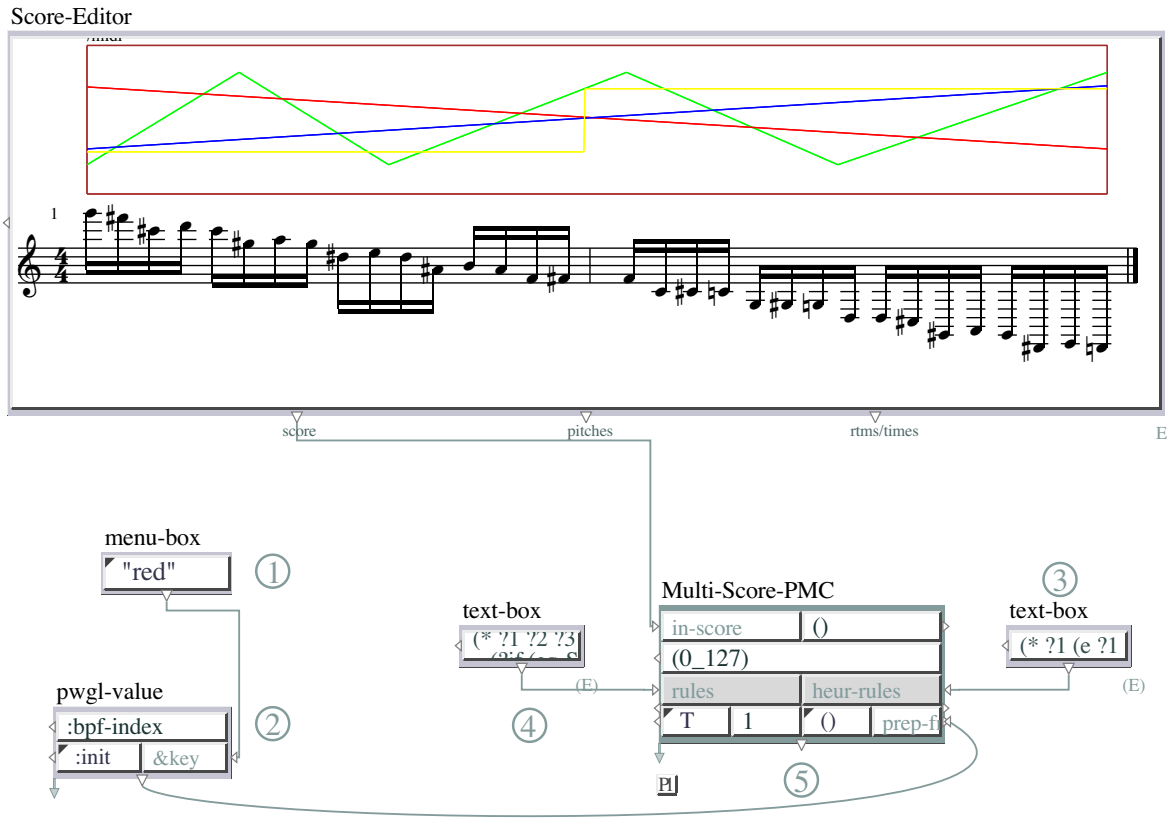

Figure 2.94: 04-heuristics-w-score-bpfs

2.5.4 PMC

2.5.4.1 Cartesian-All-Perm

This patch demonstrates some of the main concepts behind the 'Multi-PMC' box. The most important arguments are: (1) search-space (2) rules
The patch gives 2 basic combinatorial problems:
(1) When a search is run without rules (i.e. the rule input is 'nil' or '()') and the user requests for all solutions, the box returns the cartesian product of the search space (i.e. all possible paths).
(2) When a search is run with rules (i.e. the rule input receives one or several rules in textual form using the PMC syntax) and the user requests for all solutions, the box returns all possible paths that fulfill the given rules. In this case the solution consists of all permutations of the list: (0 1 2).

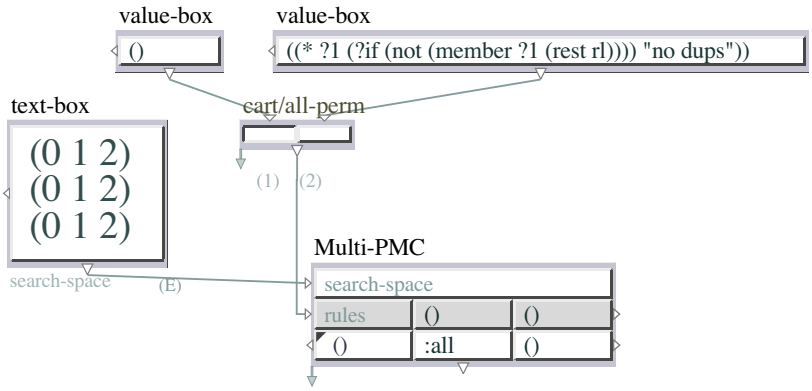

Figure 2.95: 01-cartesian-all-perm

2.5.4.2 12-Note-Chord

The patch gives a typical search problem where the task is to find all possible 12-note chords where:
(1) no octaves nor unisons are allowed. (2) adjacent chord intervals can be either 5 or 6 (perfect fourth or tritone).
The result (4 possible chords) is shown in the 'Score-Editor' box. The 'construct-chords' abstraction assigns the notes with a pitch-value below 60 to the bass clef.

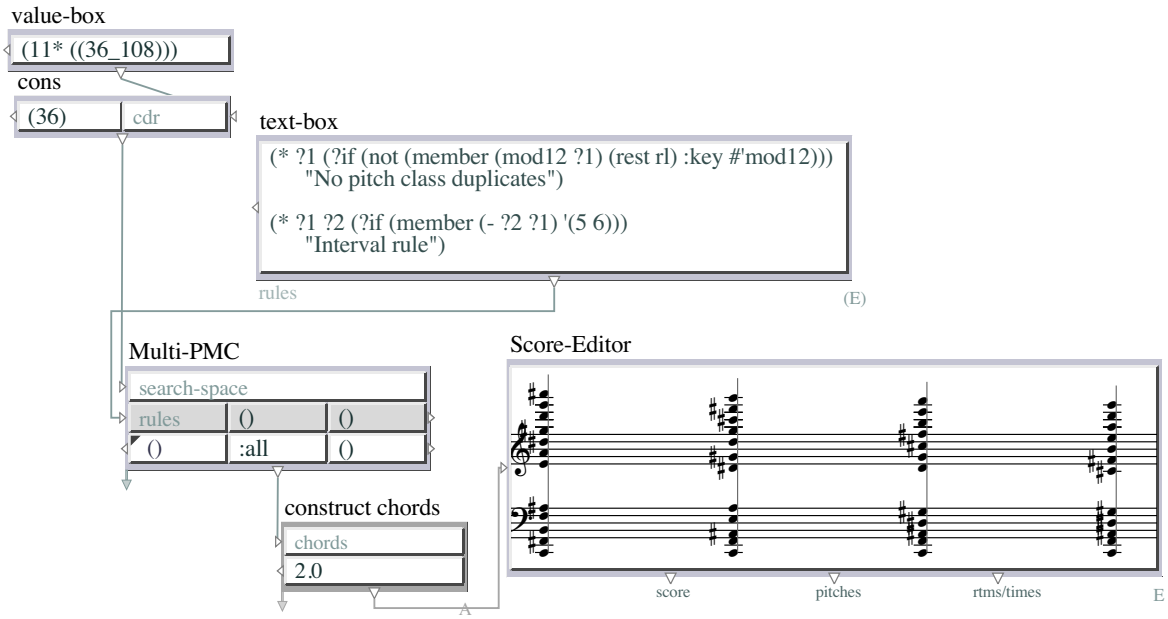

Figure 2.96: 02-12-note-chord

2.5.4.3 PMC-PCS-Ex

This patch demonstrates how to use Multi-PMC in conjunction with a text-box that contains 5 rules.
The search finds all solutions containing 8 pitch-classes (pcs) that form the set-class (sc) 5-34 (i.e. the solutions will have pc duplicates). The first 4 pcs should be 4-21 and the last 4 4-27a.
The search result is analysed according to total sc contents (left) and according to subset contents (right).

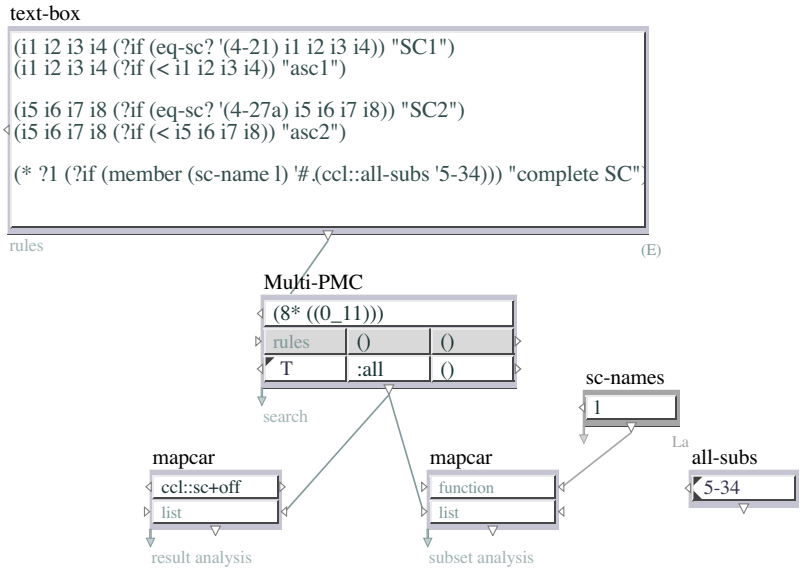

Figure 2.97: 03-PMC-PCS-ex

2.5.4.4 All-Interval-Series

This patch calculates all possible 'all interval series' instances using Multi-PMC. The lowest box 'time-box' is used to time the patch.

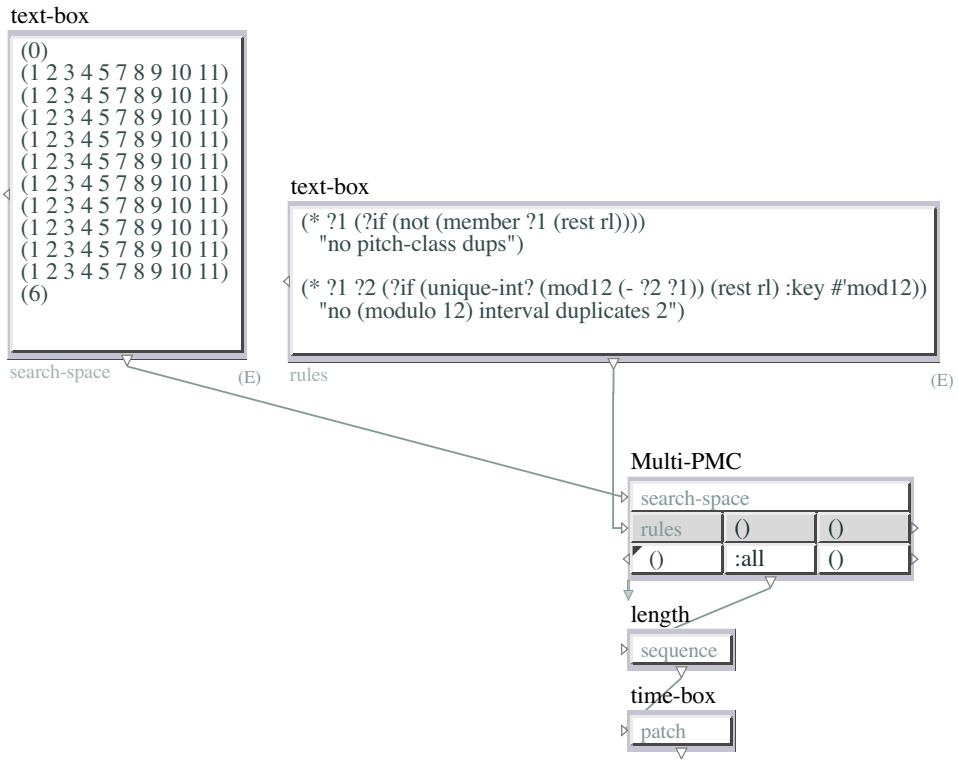

Figure 2.98: 04-all-interval-series

2.5.4.5 All-Interval-Series-2-Wildcard

This patch is a variant of the previous patch that calculates all possible 'all interval series' instances using Multi-PMC.
Run the patch normally at (1).
In this example we formalize the two rules differently as both rules use 2 wildcards in their PM-part. The general behavior of the 2-wildcard PM is explained in the the 'Overview' section.
At (2), the 'enp-script' box can be used to visualize the matches caused by the 2-wildcard PMs.
First evaluate the 'enp-script' box (2). This creates all possible matchings for each rule. To see the matchings double-click the 'enp-script' box. This opens a diagnostics dialog. For more details see the 'Overview' section.
The second rule is special as we use the extra keyword ':pm-overlap' after the normal PM-part. This allows ?2 and ?3 to overlap (i.e. they can point to the same variable) while the pair ?1 and ?2 is looped through the the search variables before the pair ?3 and ?4.

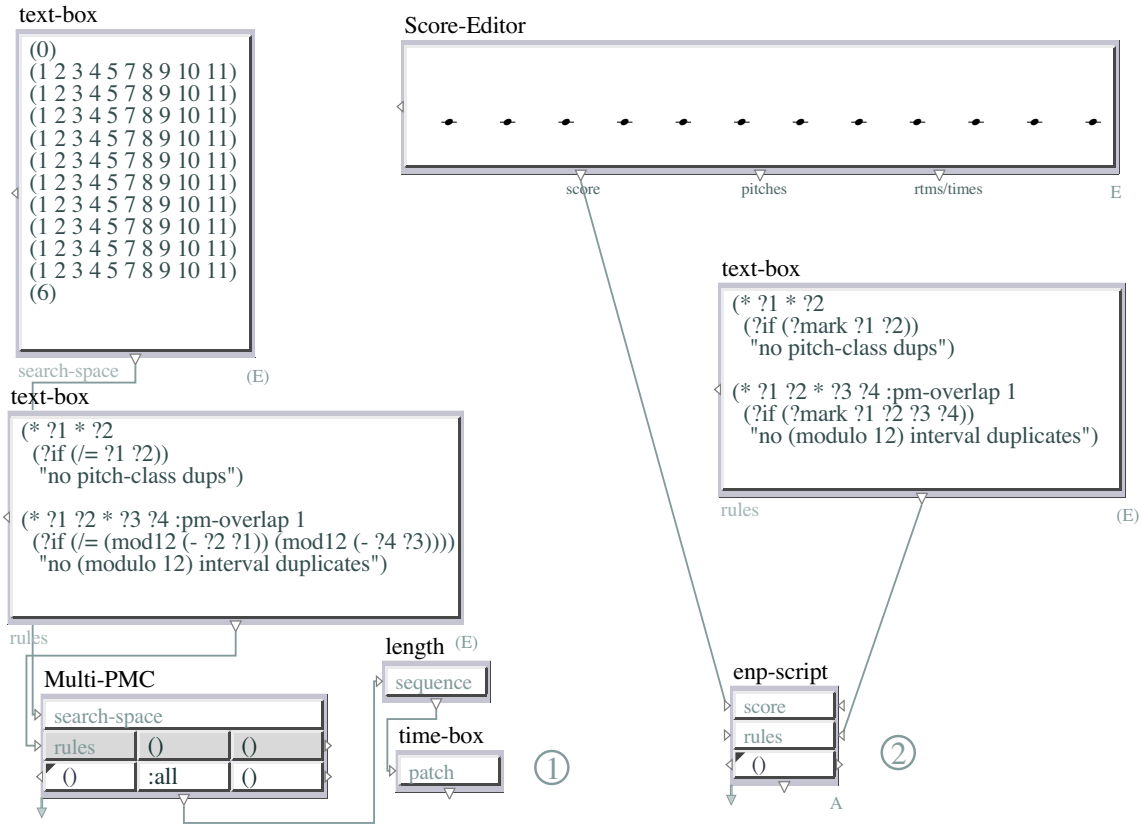

Figure 2.99: 04b-all-interval-series-2-wildcard

2.5.4.6 PMC-Beats

Here a PMC search is used to generate beats. In (1) a search space is created consisting of attacks, rests and ties. The search uses 2 rules: the beat should not begin with a tie and the result should not contain only rests (2). After the search (3) the result is displayed in (4) in textual form and in a score (5).

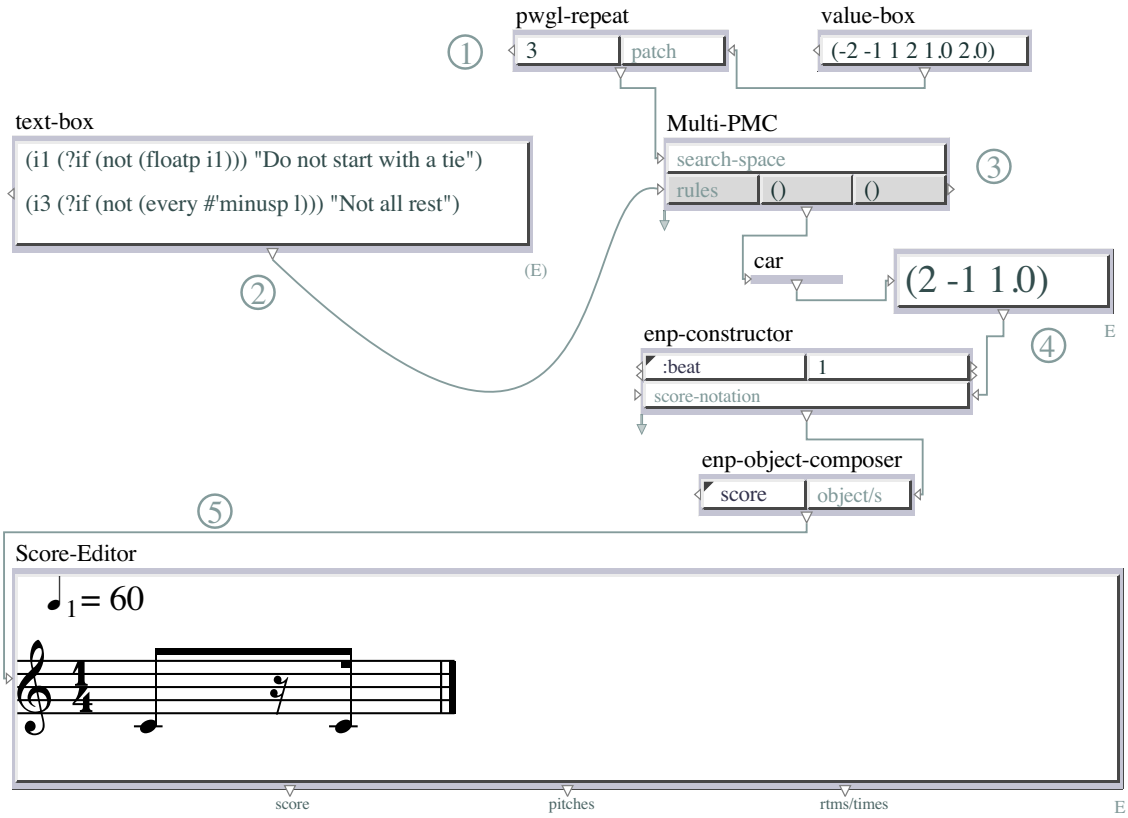

Figure 2.100: 05-PMC-beats

2.5.4.7 Subsets

This is a classical combinatorial problem where our stating point is a 12-tone 'mother-chord' (1). We want to search all 4-note subsets of the mother-chord. This can be formulated as a PMC search so that we first define a search-space which is 4 times a list of indexes ranging from 0 to 11 (2).
Next, in the rules (3) we first always guarantee that the resulting indexes list is in ascending order (see the first rule 'ascending order'). When we apply this rule alone we get 495 subsets.
We can further constrain the search by using the 'rule-filter' box (4) to turn on or off the two other rule options: 'sc identity' and symmetric chord'.
The results can be analyzed with the 'sc-name' box (5). The final result can be inspected in a 'chord-editor' (6) which displays the chords as a matrix.

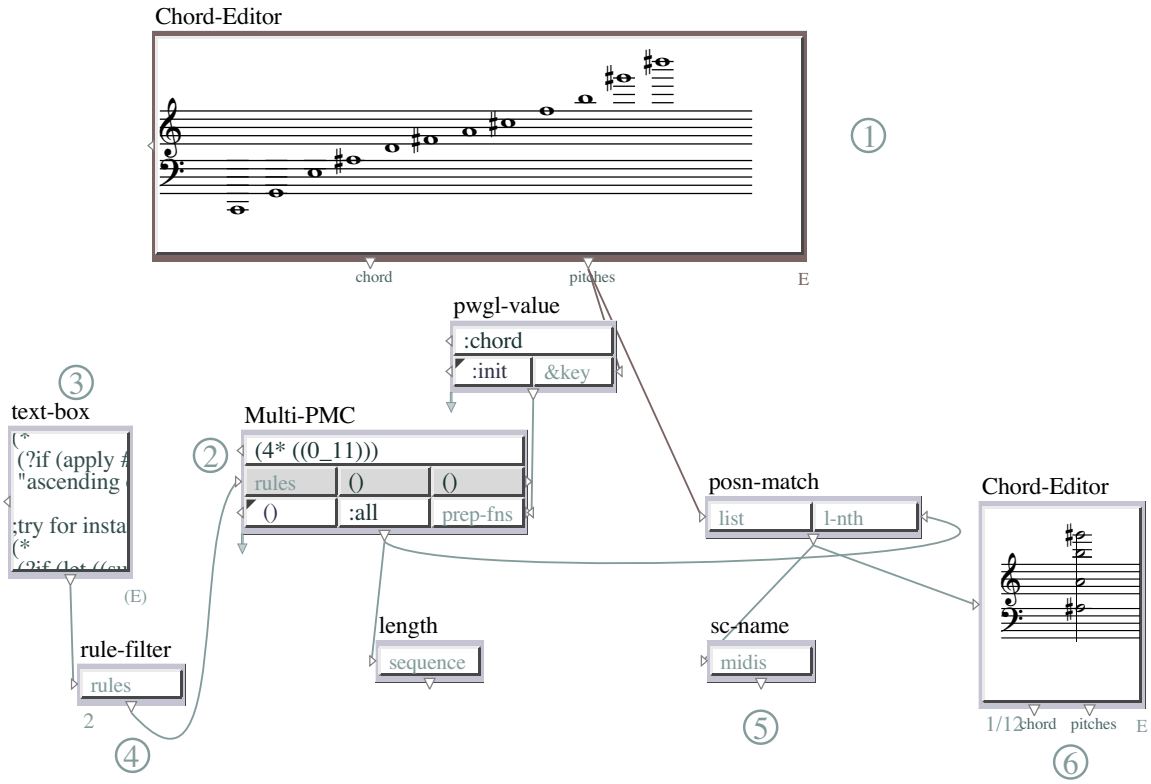

Figure 2.101: 06-subsets

2.5.4.8 Fantasiesonnightfantasies

We meditate here on the pitch-dimension of 'Night Fantasies' by Elliott Carter. First we calculate all the all-interval chord-classes found in the piece following David Schiff (The Music of Elliott Carter, pp. 316-318). Then we tighten the constraints and filter out chords with more specific properties.
Select one of the four 'filters' with MasterSW. Then evaluate the corresponding Chord-Editor (1-4).
1. 176 all-interval-chord-classes are filtered from 3856 in total. Because each interval is paired symmetrically with its inversional equivalent we have 88 originals and 88 inversions. 88 - isn't that the number of the pianokeys?
The chords are sorted in relation to the average frequency and three relatively most bright and dark chords are chosen.
2. A few pitches may be sustained as a kind of pedal in 'Night Fantasies' . Here we constraint a major second between indexes 11 and 12.
3. A ref-chord (31,44,74,85) has to be found in the result. Please notice that we use now the complete tessitura of the piano as a search-space (21-108).
4. Indexes 2,4,7,11 should form setclass 4-z15a.

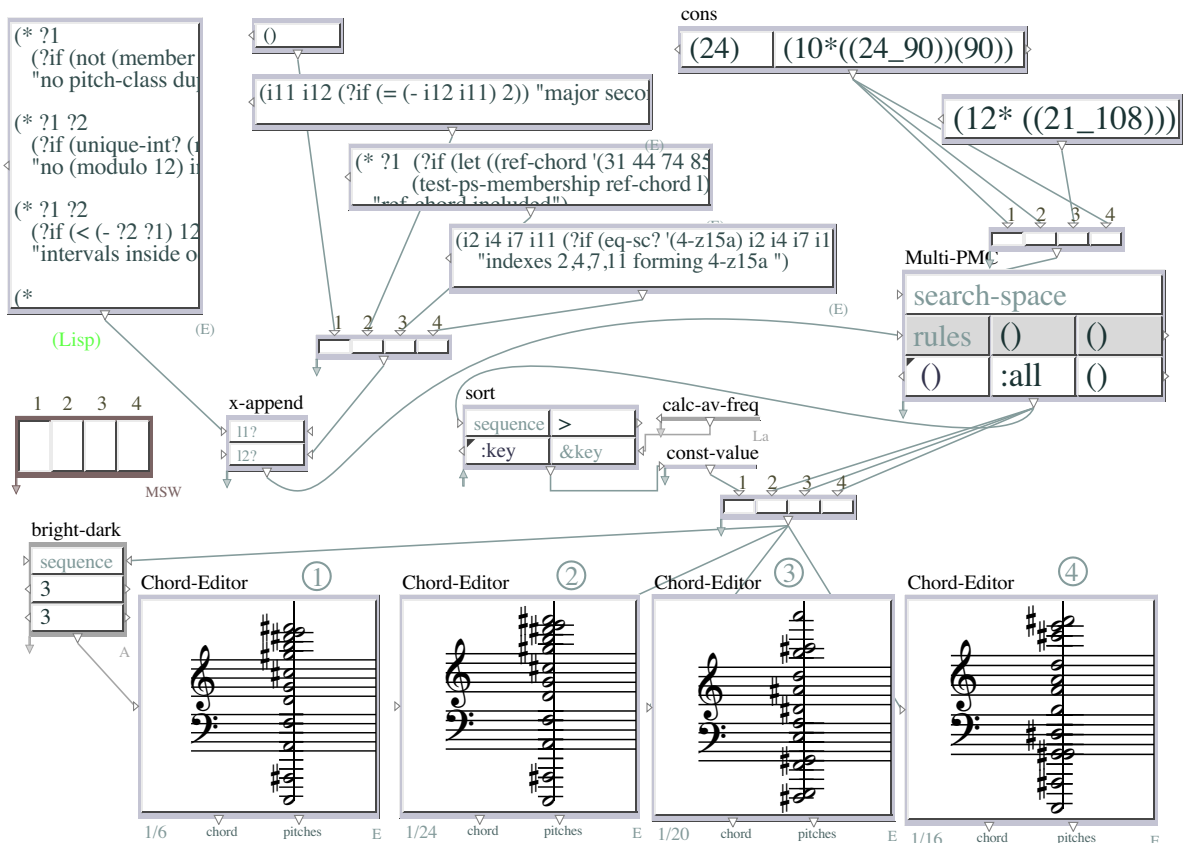

Figure 2.102: 07-FantasiesOnNightFantasies

2.5.4.9 Fund-Suspension-Chain

In this patch we examine the interaction between components in a procedure where a set-class (sc) structure is projected to real intervals (chord). The result is in turn interpreted as overtones of a fundamental. Moreover, this resulting sequence of chords is modified by making a chain of suspensions in the bass. Please notice that outcomes can vary a lot because of a random openness in many turns of a patch.
We start by generating a symmetrical dodecaphonic 'mother-chord' (1). The chord is transposed randomly within one octave.
We throw the dice for a subset sc with which the 'mother-chord' will be filtered (2). We give a cardinality and a randomly picked subset sc is used in the 'Gen-chords' abstraction (3). Here we search for all possible subset chords from the mother-chord that constitute the given subset sc.
The resulting chords are compared with a list of midis which is an equal tempered harmonic overtone approximation (4). The output is a list of fundamentals of the chords. The melodic contour of the fundamentals is seen in a 2D-Editor (5).
We combine the fundamentals with the corresponding chords (or overtones). The first overtone, octave, is added to every list (6). By sustaining the lower two notes of

the chords (fundamental+octave) we create suspension sequences between subsequent chords (7).
Our tension-release schema is not constant because we can have the same fundamentals following one another (in this case only the brightness of the chords is changed). In the end we make a little swing to the choral by randomizing durations between low and high limits (8).
Evaluate the patch at (9).

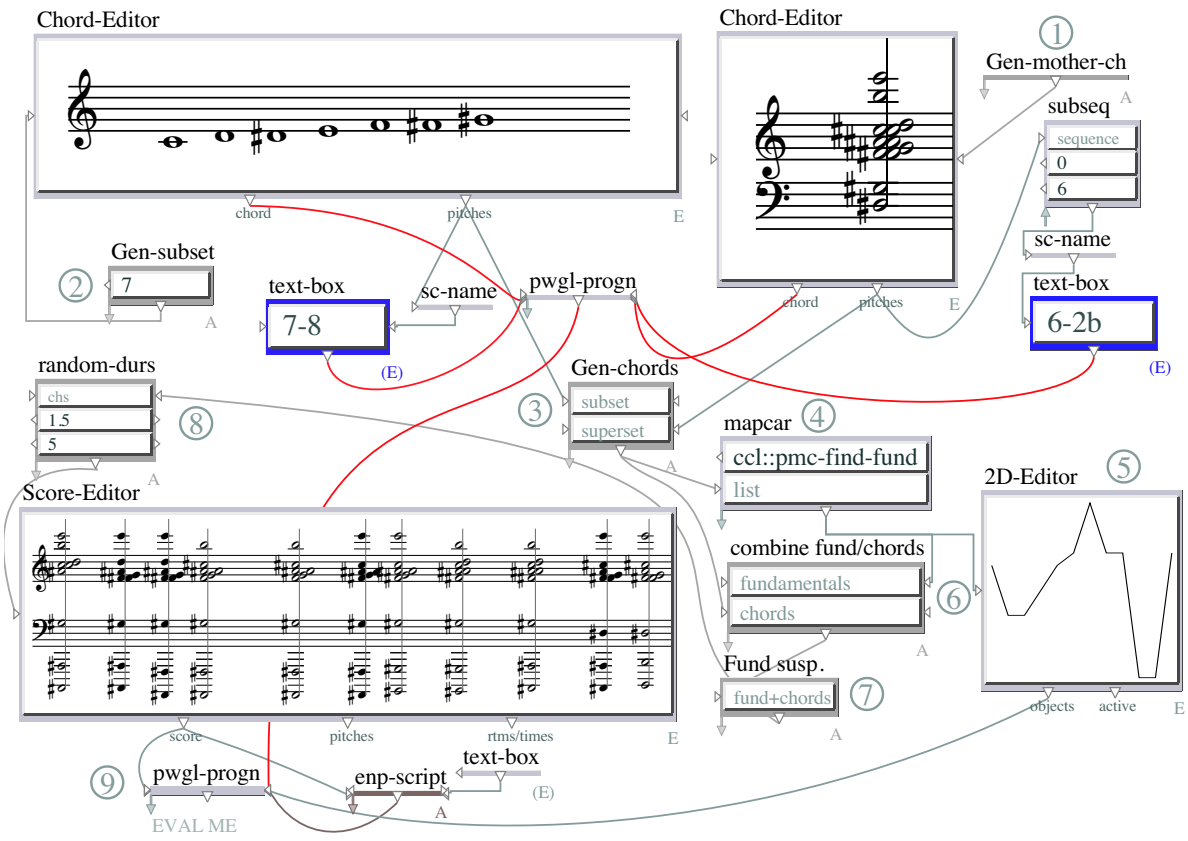

Figure 2.103: 08-fund-suspension-chain

2.5.5 Score-PMC

2.5.5.1 PMC Vs-Score-PMC

This patch demonstrates some of the differences and similarities between (1) PMC and (2) Score-PMC.
Score-PMC operates with an input-score, 'in-score', where the notes act as search-variables. The second input, 'res-score', can optionally be connected to another 'Score-Editor' box in order to display the result of the calculation. In this simple patch example this input is '()', and the result will be updated directly in the input-score.

The search-space is defined with the 'search-space' input, which is in this case a simple list of midi values ranging from 60 to 72.
The search aims to find pitch-values in the score that are valid according to rules. PMC rules are quite similar than the melodic rules in Score-PMC. The main difference is that in Score-PMC the m-method ('m') is used access information from the notes (in this example the rule deals with midis).

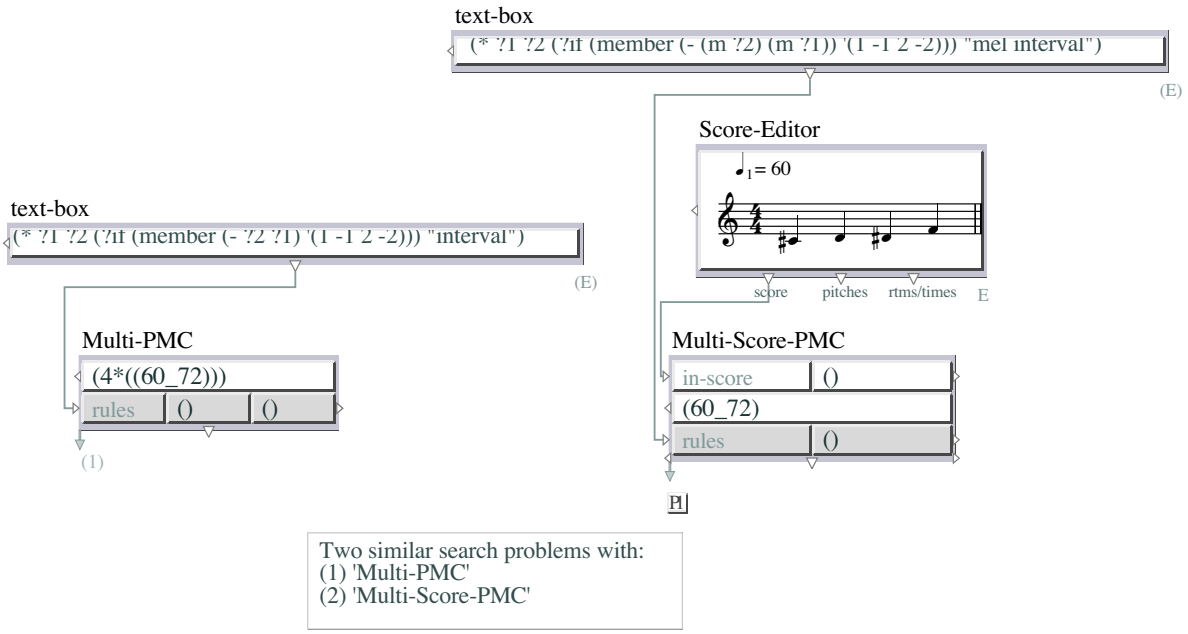

Figure 2.104: 01-PMC-vs-Score-PMC

2.5.5.2 3-Voice

This small 4*3 matrix score deals with some melodic, harmonic and voice-leading rules. We have two 'Multi-Score-PMC' boxes here: the one to the left finds one solution, and the one to the right looks for all solutions. In the latter case our problem has 52 solutions which can be accessed individually by selecting the 'Multi-Score-PMC' box and using the up/down arrow keys. The numerical information below the box displays a small label with two numbers (e.g. 1/52): the first number points to the current solution, and the second number gives the total number of solutions.

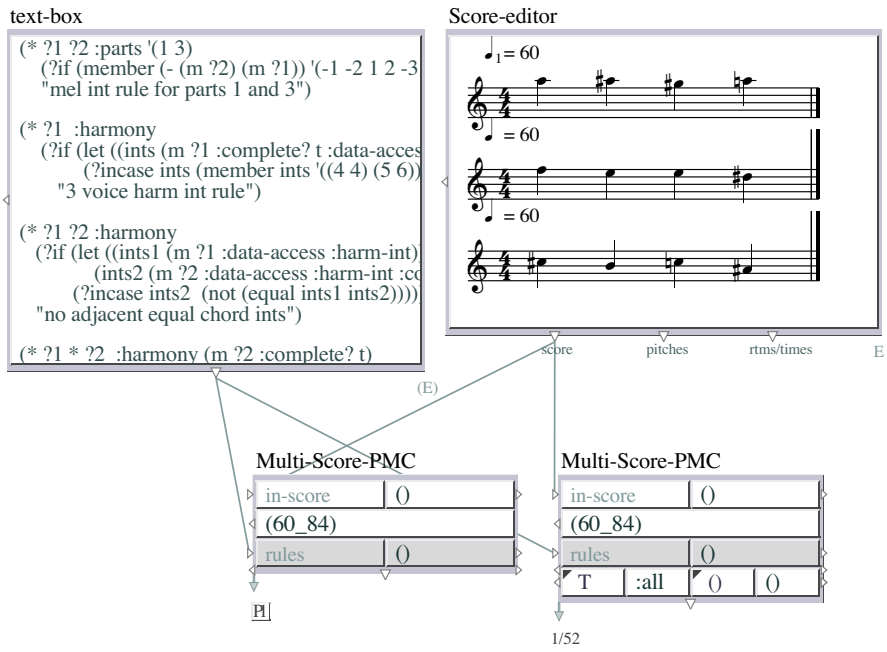

Figure 2.105: 02-3-voice

2.5.5.3 6-Voice

Our next example is more complex and the rhythmic structure is more lively containing overlapping notes. Again several basic melodic, harmonic, and voice-leading rules are applied.
The patch contains also an 'enp-script' box that performs a harmonic analysis.

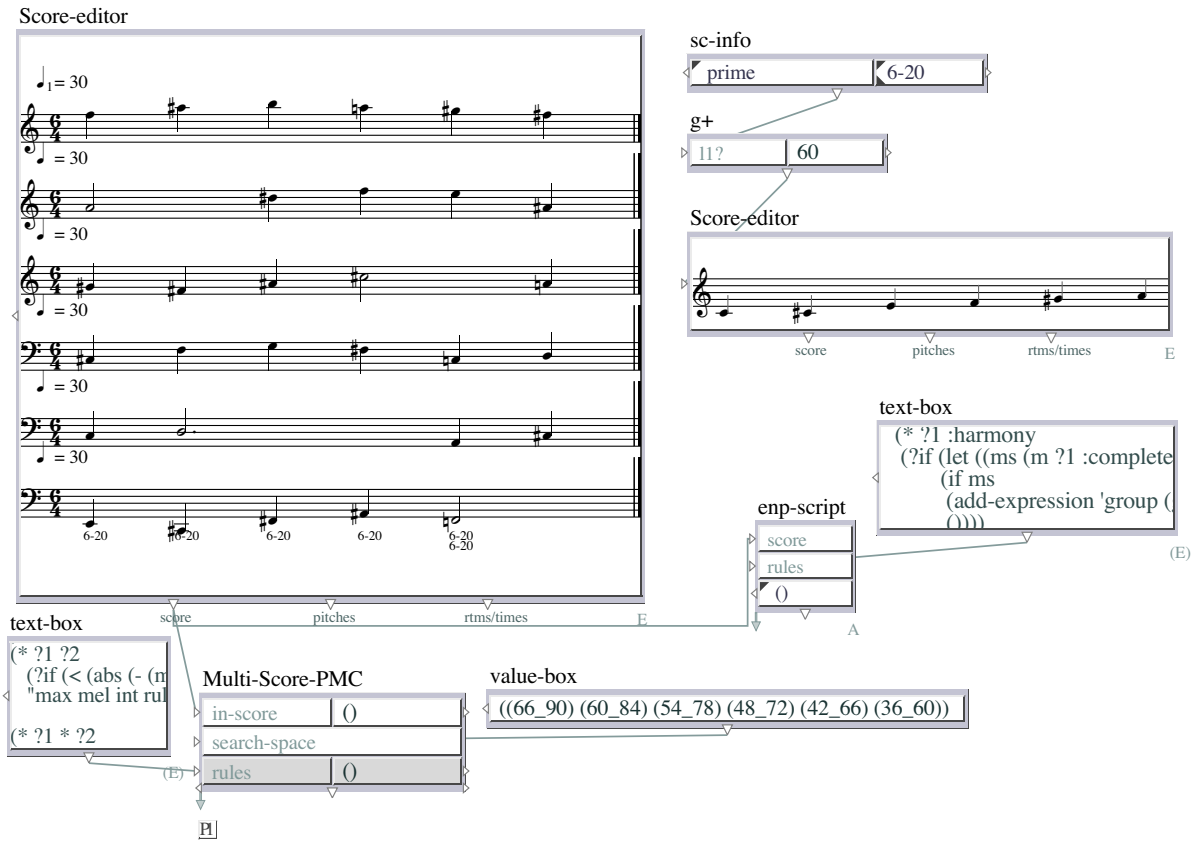

Figure 2.106: 03-6-voice

2.5.5.4 Chord

This patch demonstrates rules that deal with chords of varying density. The rules control harmony and melodic formations of the lowest and highest pitches of the chords.

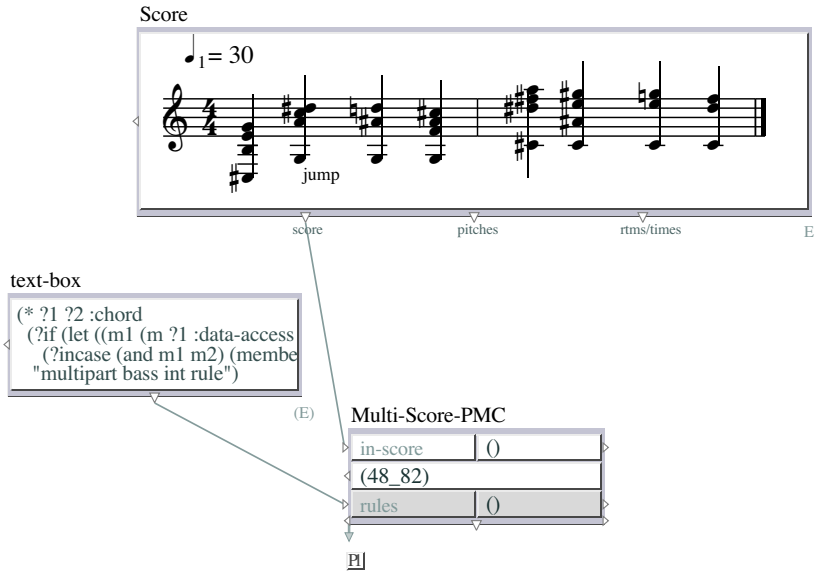

Figure 2.107: 04-chord

2.5.5.5 Grace

This patch deals with grace notes. The first rule, 'mel int', controls the melodic intervals between notes regardless of whether the notes are ordinary ones or grace notes.
In the second rule, 'grace int', grace notes have a different set of intervals than the ordinary ones.

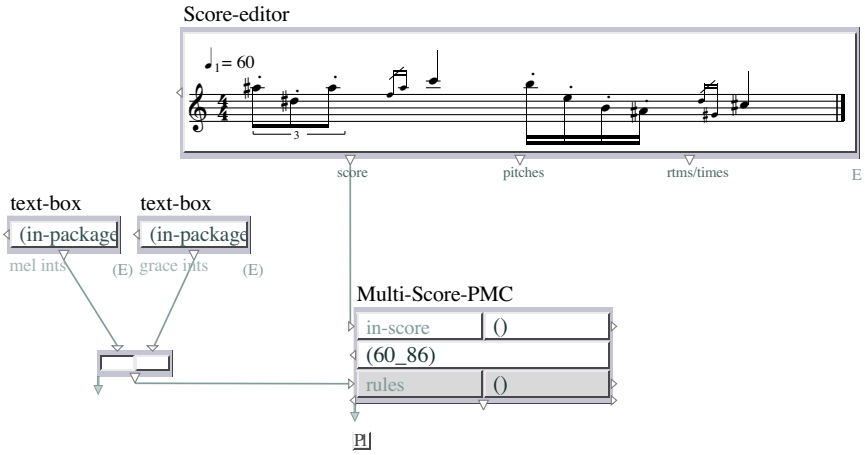

Figure 2.108: 05-grace

2.5.5.6 HSG

This patch has been developed in collaboration with Paavo Heininen and it is related to the discussion of chapter 5.4 in the following text:
M. Laurson, PATCHWORK: A Visual Programming Language and some Musical Applications. Studia musica no.6, doctoral dissertation, Sibelius Academy, Helsinki, 1996.
The aim is to calculate the pitch information for the 3 topmost parts (sop, mid-voice, bass). The melodic lowest part has been pre-composed by Paavo Heininen and acts as a kind of cantus firmus, i.e. the pitches in part 4 are always fixed and should be 'constraint' by the user before the search. These constraint notes are dimmed (i.e. they are drawn in light-gray color, see part 4 in the score).

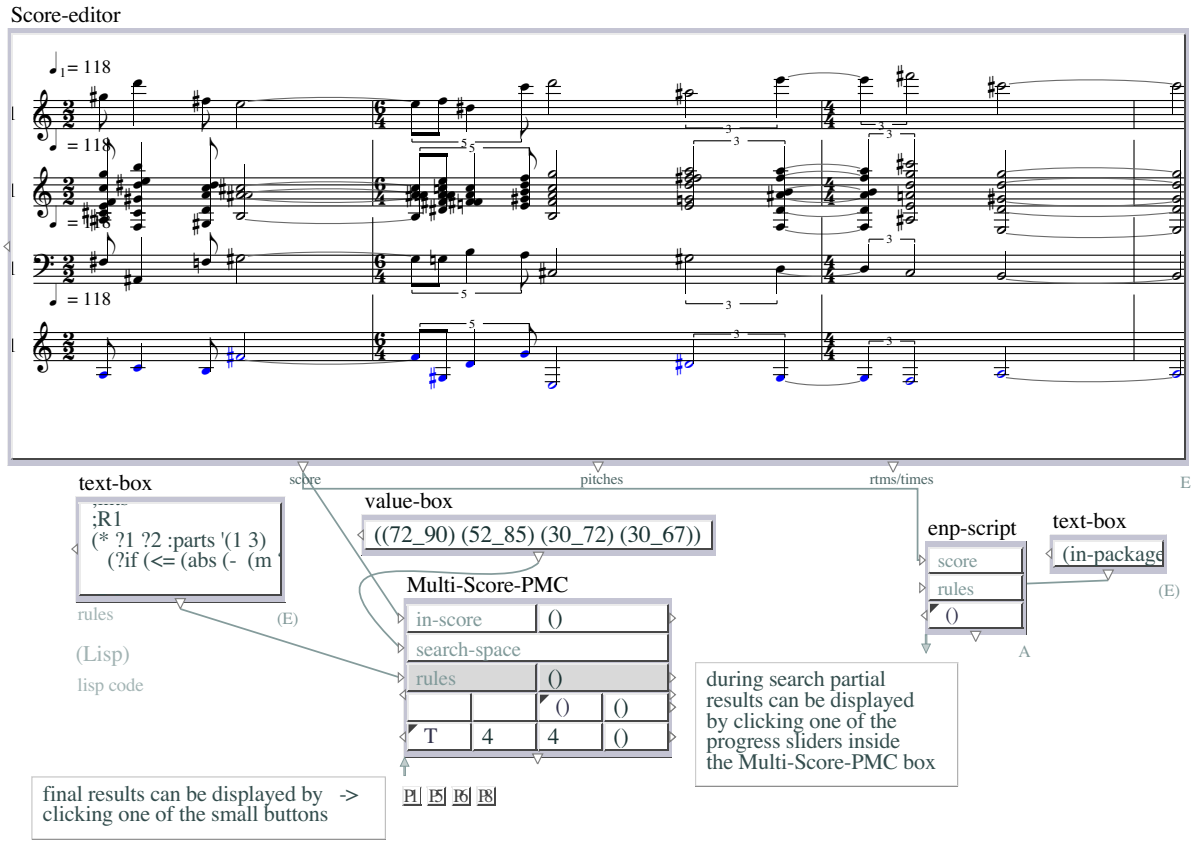

Figure 2.109: 06-HSG

2.5.5.7 6-Z47b-Blues

In this example a score-pmc search is applied to a non-mensural score. The one-part score consists of chords of varying density. Thus several rules use the ':chord' accessor. These rules allow for instance to control separately the voice-leading and interval content of the melodic contours that are formed out of the upper and lower notes of the chords.
The heuristic rule comes from an abstraction that uses internally the 'mk-score-pmc-profile-hrules' function (see also the 'Heuristic' section).
Finally, this patch uses also scripting to assign playback information.

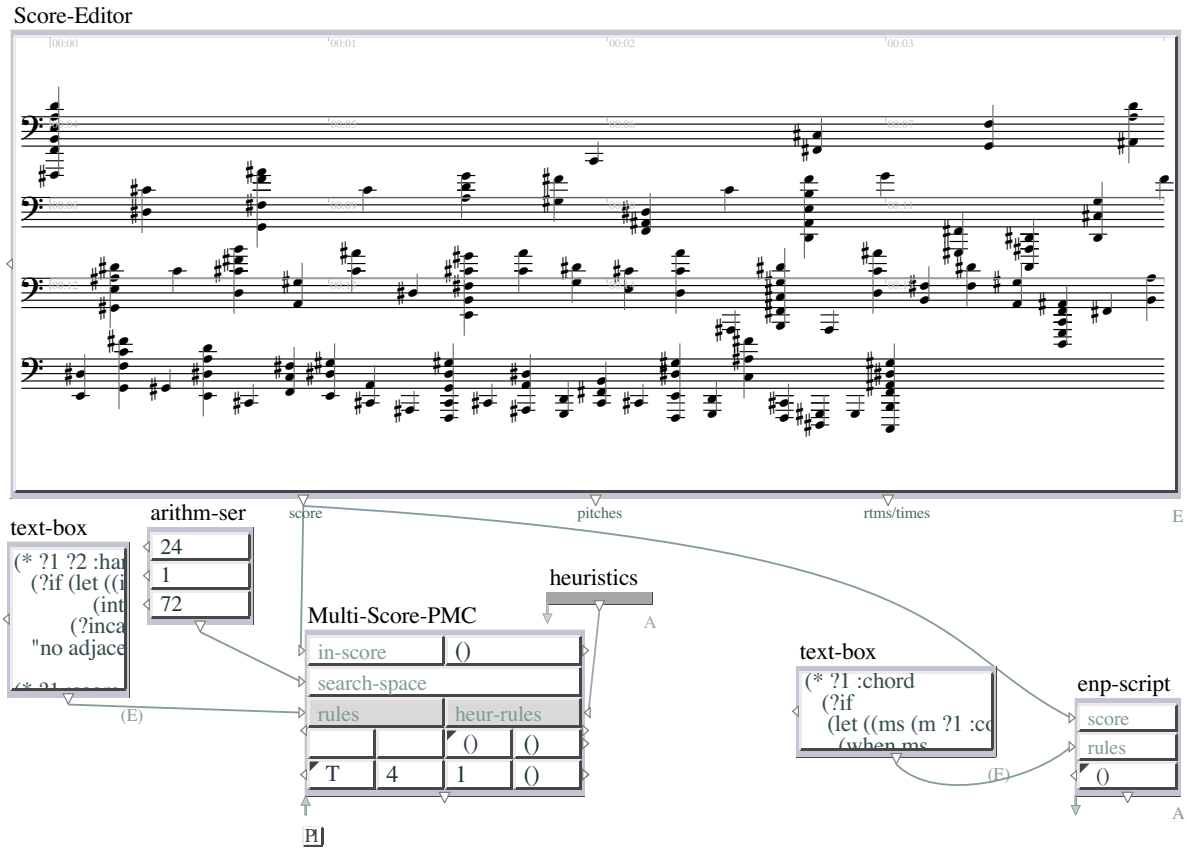

Figure 2.110: 07-6-Z47B-blues

2.5.5.8 Grace-Duetto

This patch is a one-part score where grace-note gestures of varying length alternate with long notes. There are here three basic rules sets: (1) rules that apply only to long notes; (2) rules that apply only to grace-notes; (3) rules that apply to all notes.
Besides typical rules that deal with pitch ranges, forbidden pitch-class repetitions and allowed pitch-class sets, this example contains several rules that control the pitch contour of the grace-note groups. These contours are marked in the score with labels like '7/5', '6/2', '6/3', etc. The groups that are marked with labels that begin with 7 form contours that fan out, while the ones beginning with 6 fan in.
The patch contains also a 'enp-script' box that is used to assign notes below middle-C to the bass clef, to set playback information (channel and velocity) and to add color-coding of the note-heads.

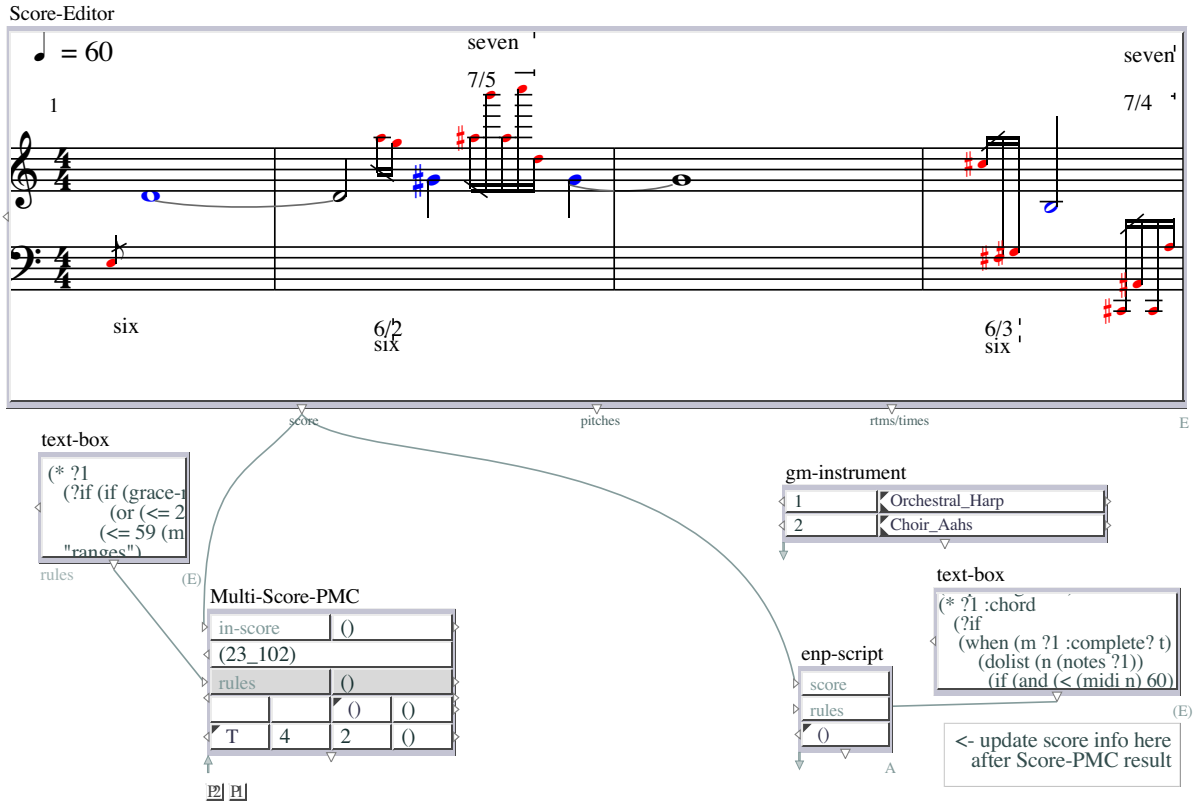

Figure 2.111: 08-grace-duetto

2.5.5.9 First-Species-Counterpoint

In this patch, we sketch a small rule set to generate simple 'first species' counterpoint. In the middle of this patch (literally speaking) is one big merger box. The merger box allows us to experiment with different combinations of the rules. It divides the whole rule set into predefined 5 groups. By default (all the switch buttons are off) the result of the calculation is random. By turning on the switch buttons one by one it is possible to observe how the constraints engine gradually refines the end result. The predefined rule groups and their approximate roles are as follows:
(1) defines some simple melodic rules; (2) defines the allowed harmonic intervals between the voices; (3) implements the 'horror vacui' rule where a leap is balanced with a smaller contrary movement; (4) prohibits both strict and hidden parallel movement; (5) contains rules for the cadence
In (6) the user can add her/his own rules.

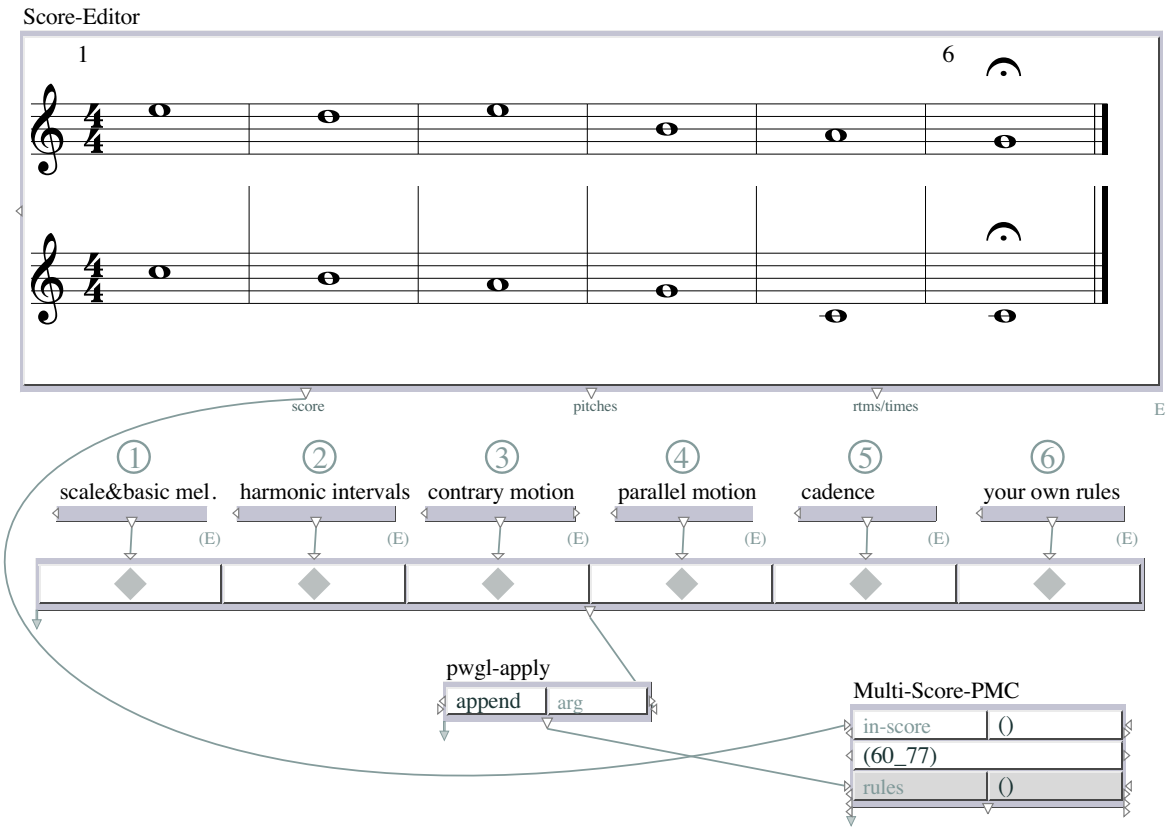

Figure 2.112: 09-first-species-counterpoint

2.5.5.10 Alberti-Bass

This tutorial demonstrates how to simulate Alberti bass accompaniment for a given melody. The right-hand part has been constrained in advance (the notes are drawn in light-grey color), and thus will be fixed during search. The same applies also for the first beat of the left-hand part. The patch gives the harmonic degree markings as Roman numerals.
The rules demonstrate some advanced topics in the PMC syntax: :or rules and accessing the plist information (here the Roman numerals are stored in the plists of the chord objects of the right-hand part).

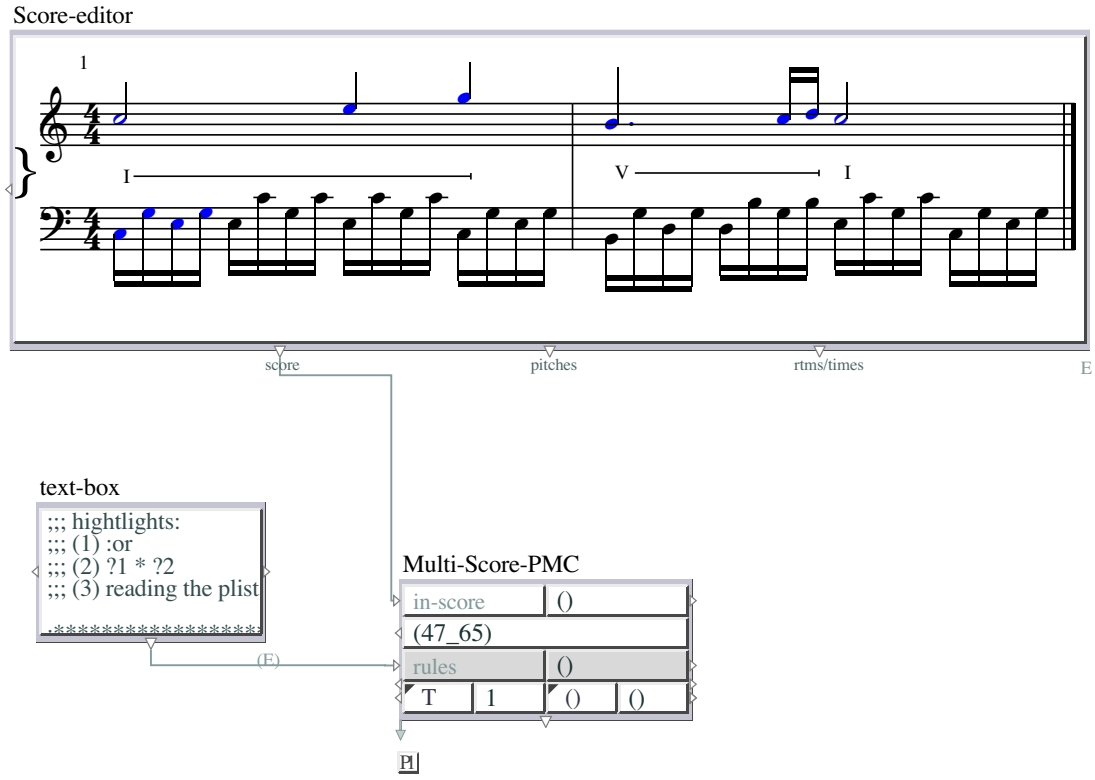

Figure 2.113: 10-alberti-bass

2.5.6 RTM

2.5.6.1 Introduction

Until now all Score-PMC searches have been pitch-oriented. In this section we present a way how to expand this scheme so that Score-PMC can be used to solve problems that are related to other score parameters, such as rhythm, dynamics, expressions, enharmonics, and so on. Several parameters can also be searched simultaneously in one search process. In this section we concentrate on the generation of rhythms.
The most important new box that is introduced here is called 'score-pmc-search-space' that returns a list of pmc-domain objects that contain internally an association list of property/value pairs. This box is used in conjunction with the 'multi-score-pmc' box, and its output is normally connected to the 'search-space' input.
The user can choose different search properties, thus allowing to specify searches with multiple parameters. These properties can be accessed during search in rules using the m-method with the keyword ':data-access'. After the search has been completed, the result score of the 'multi-score-pmc' box will be updated automatically. For more details see the box documentation of 'score-pmc-search-space'.

In the following examples we use for the search-space the property/value ':rtm' and 'T'. Thus this means that any note in the 'in-score' can become either an attack, rest or tie ('ART'). In these examples we connect a separate 'score-editor' box to 'res-score' in oder to be able to build the result score.
In rhythmical rules we use in the m-method the pair ':data-access :rtm' to access the RTM values, which can be either :attack, :rest, or :tie. Thus in the following rule we force all downbeats to be attacks:

```
(* ?1 :rtm (downbeat? ?1)
   (?if (eq :attack (m ?1 :data-access :rtm)))
  "downbeat␣attacks")
```

2.5.6.2 Rnd-Mod-RTM

This patch shows how to create from a pulse of 1/16 notes: (1) random rhythms (2) modulo rhythms.
We use here a 'score-pmc-search-space' box to construct a search-space for the 'Multi-Score-PMC' box. In this case the search-space used internally is the following list: (:attack :rest :tie).
In option (1) this means that any note in the pulse stream given in 'in-score' can become either an attack, rest or tie.
In option (2) we use a rule that operates with modulo arithmetic. We take the melodic index 'notenum', and check if the index modulo 'mod' is equal to 'pos'. If this is the case then we have an attack, otherwise a rest.
The 'mod' and 'pos' values can be edited in main patch.

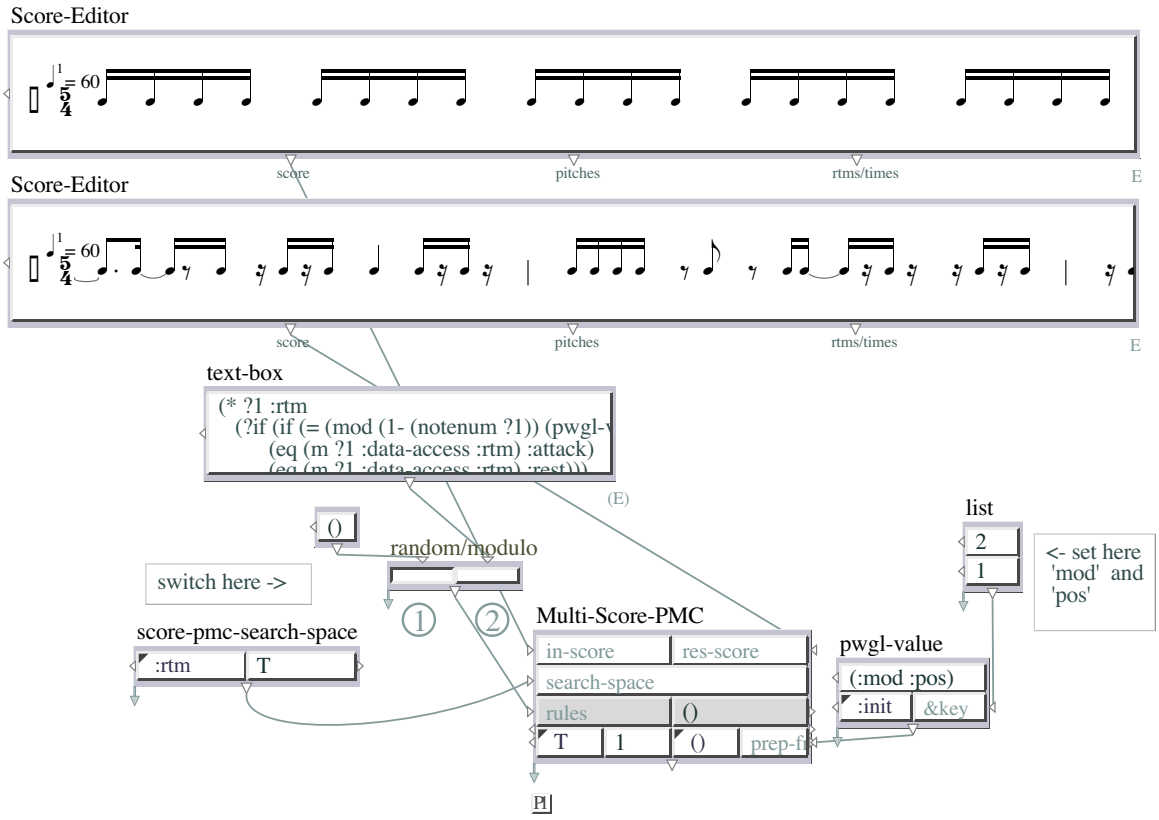

Figure 2.114: 01-rnd-mod-rtm

2.5.6.3 2-Part-RTM-Textures

This patch has a more complex 2-part input-score (to see it open the score that is connected to the 'in-score' input). The example has 5 rules that use a lot of rule selectors. These allow to utilize rules locally resulting in 3 different texture types:
(1) measures 1-2, control of onbeat and offbeat rhythms in both parts (2) measure 3, alternating, or 'hoquetus' like, rhythm in both parts (3) measure 4, both parts are synchronized
There is also a rule that allows to reduce a very dense pulse (this example contains beats that are divided upto 12 units), to more simple beat structures (see the last 'simplify rtms' rule). This rule utilizes the 'match-ART-rtms' function that is optimized for RTM search problems.

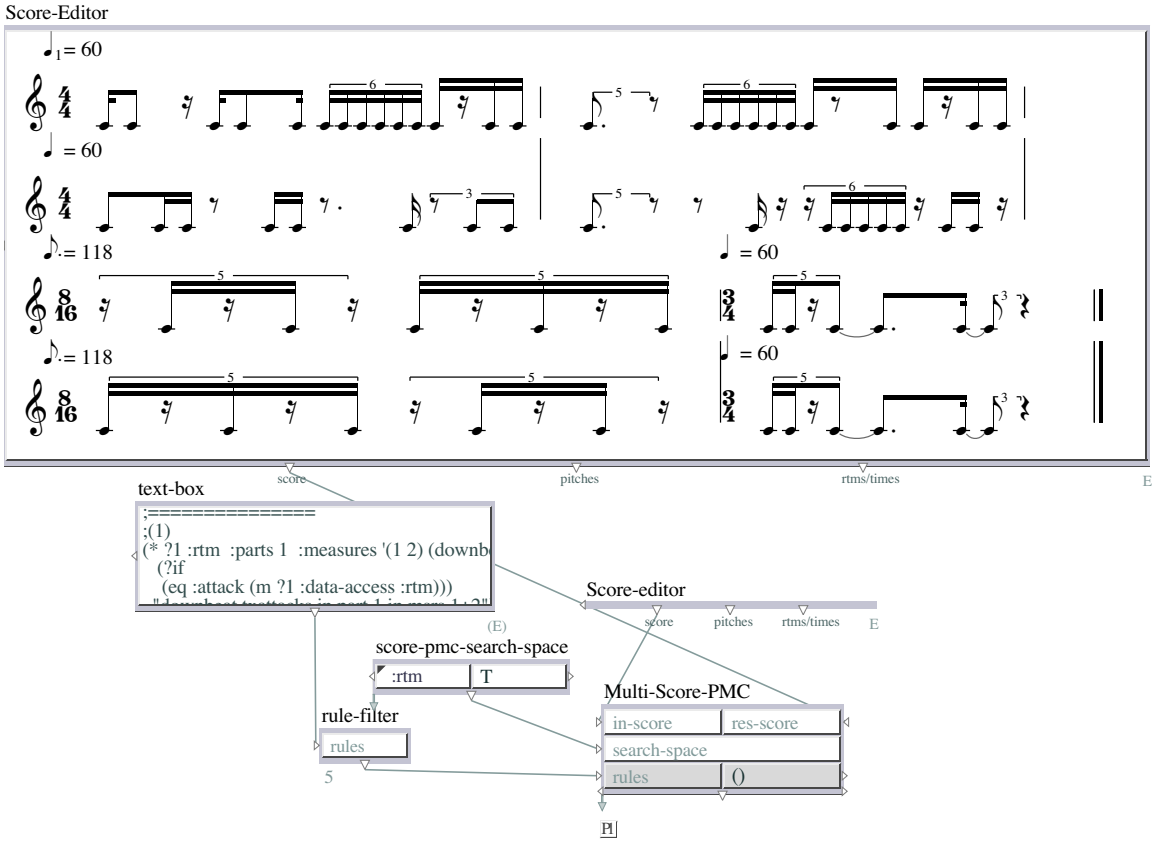

Figure 2.115: 02-2-part-rtm-textures

2.5.6.4 Reduce-RTM

In order to generate various tuplets out of a pulse stream, one can calculate the needed beat division with the Lisp function 'lcm' ('least common multiple').
In this case we want to be able to divide a beat in either 1, 2, 3, 4, or 6 units. which results in a beat division of 12 units. Using this information we can generate algorithmically an input score. (see the 'gen-score' abstraction).
The desired beat structures are defined in the rule 'simplify rtms'. You can see the effect of this rule by turning on or off the 'simplify rtms' rule using the 'rule-filter' box.

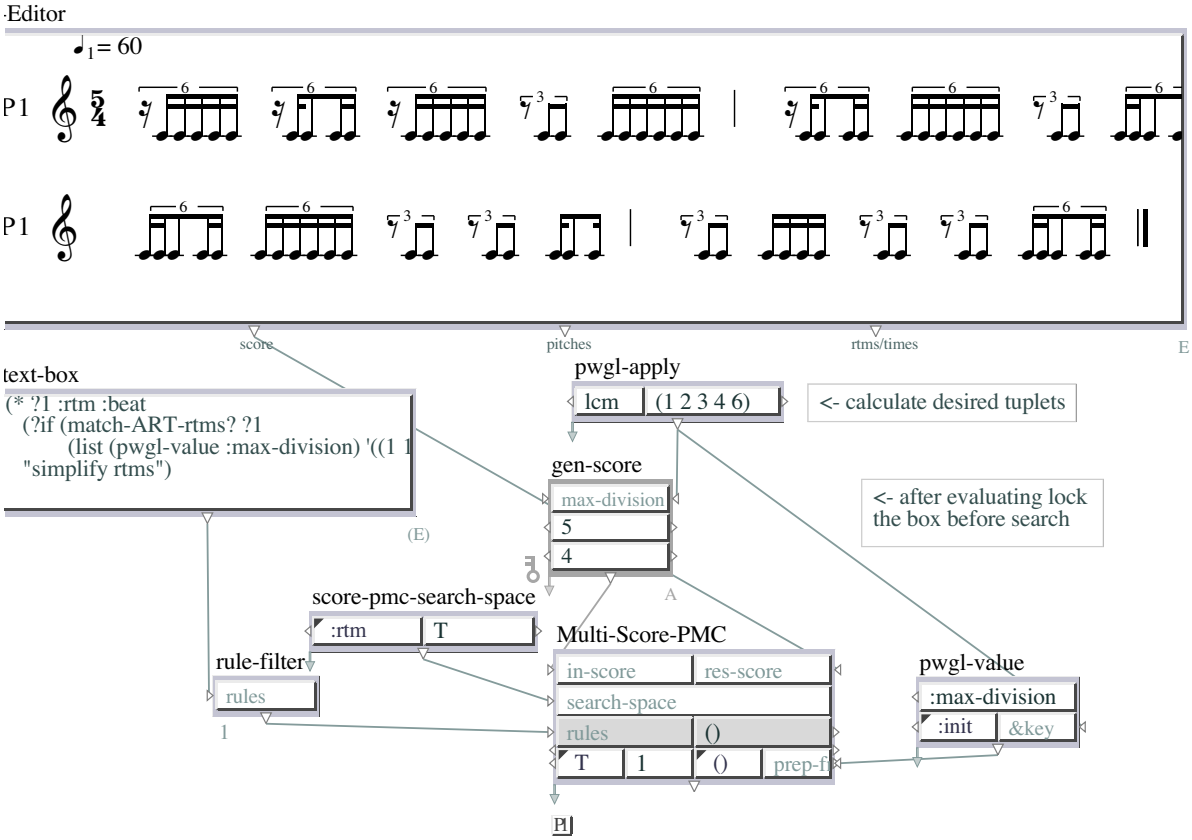

Figure 2.116: 03-reduce-rtm

2.5.6.5 8-Voice-Attack-Dens

This example has eight parts. The idea is to generate a symmetric score that starts from the bottom parts, in the middle all parts can have notes, and at the end again only bottom parts have notes. The same symmetric structure is applied also to the number of simultaneous attacks: in measure 1 only 1-note attacks are allowed, measure 2 contains 2-note attacks, etc. This process evolves up to measure 4, and after this the process proceeds backwards.

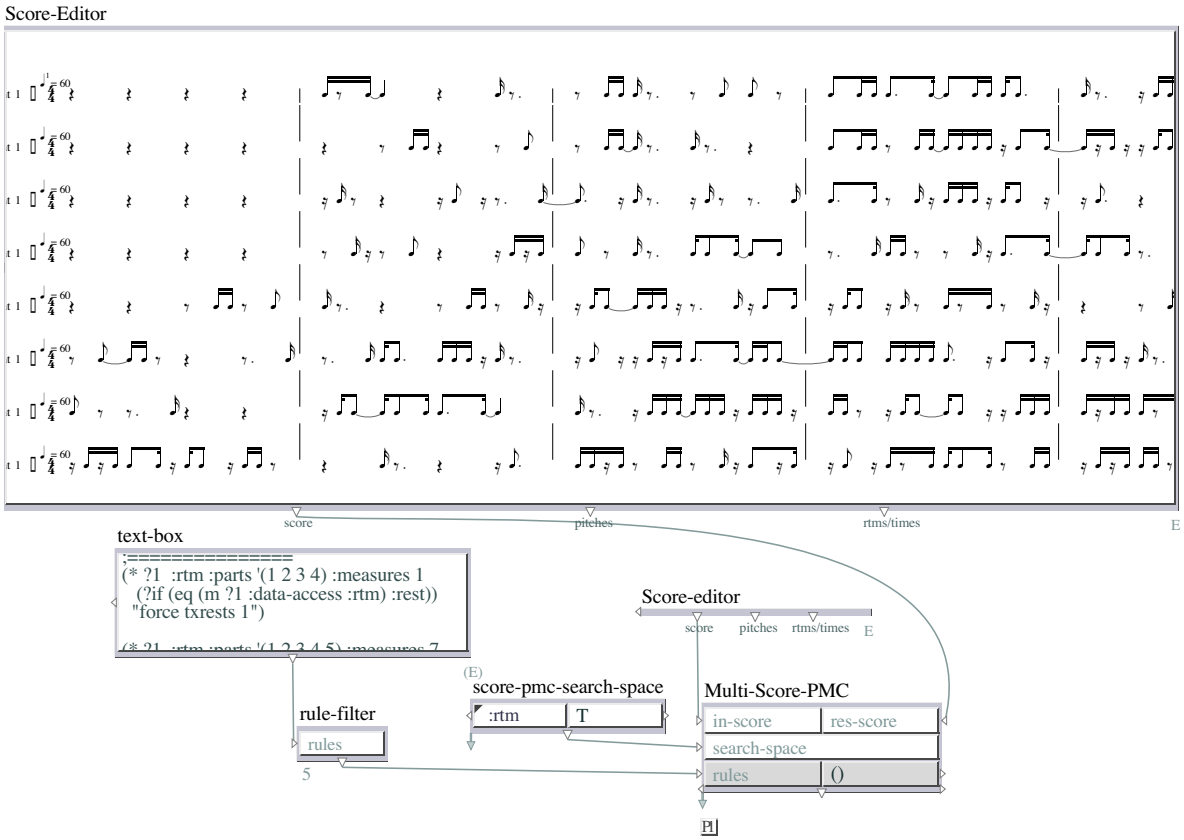

Figure 2.117: 04-8-voice-attack-dens

2.5.6.6 RTM-Simulation

Our next example demonstrates how Score-PMC can be used as a kind of textural 'analysis/resynthesis' tool. A musical excerpt is analysed by the user and the rhythmical and textural features are converted to rules. After this the search is run in order to reproduce the original musical texture. Typically the result is not an exact replicate of the original.
The example shows one result using 9 RTM rules (the original musical excerpt comes from the first part of Gyrgy Ligeti's 'Ten pieces for wind quintet').
Two rules define important characteristics of our example. The first rule states that there should be no downbeat attacks after measure one. The second rule, in turn, states that only single-note attacks are allowed after measure one.
Other rules control the exceptional status of measure one, the density of attacks and how rests are positioned in the resulting texture.

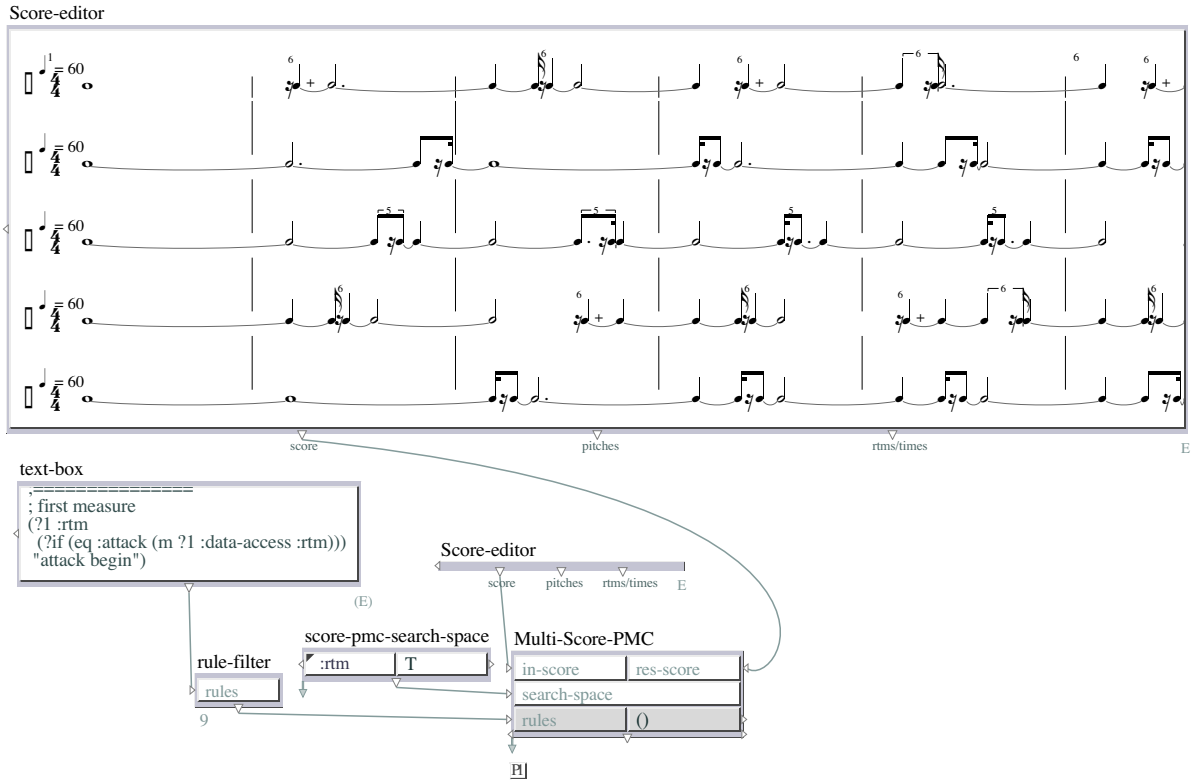

Figure 2.118: 05-RTM-simulation

2.5.6.7 RTM-Imitation1

The rules in the next two examples utilize a generic function called 'PMC-imitation' that generates canon-like textures in the result score. The input score contains group expressions that have a label string such as 'imit1'. The imitation rules in turn use this string, and two part numbers, 'p1' and 'p2', in order to constrain these groups within the given parts according to a lisp function 'fn'.
In these patch examples we use 'PMC-imitation' for RTM imitations (hence the optional argument ':rtm'). 'PMC-imitation' can also be used for other parameters such as pitch.

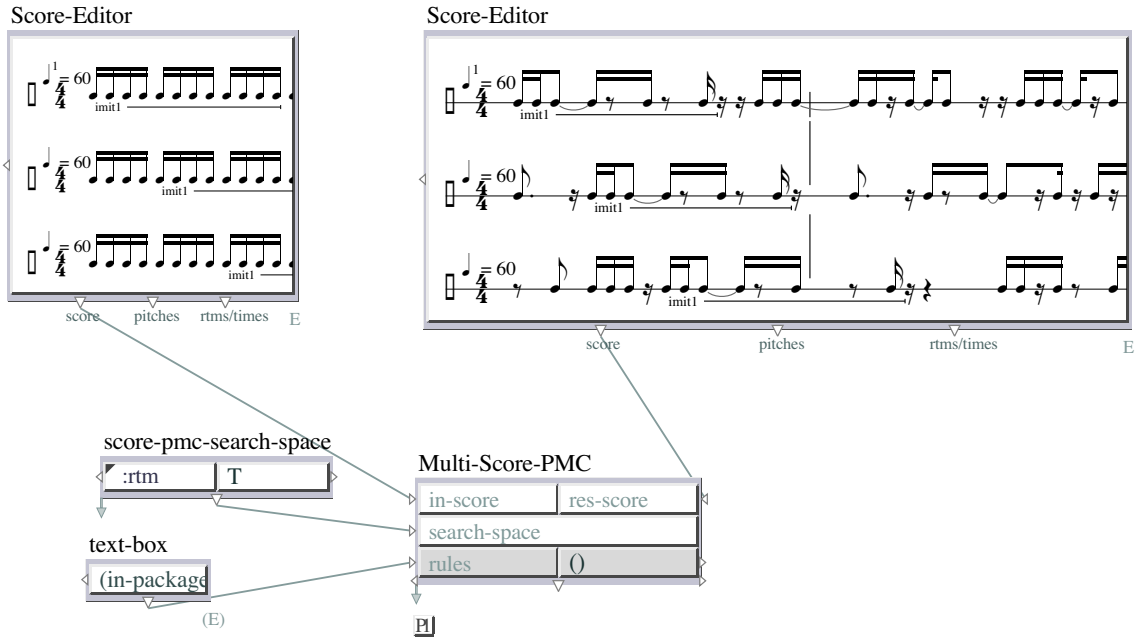

Figure 2.119: 06-RTM-imitation1

2.5.6.8 RTM-Imitation2

This example has an input score that has RTM pulses of 1/16-, 1/8- and 1/4-notes. This patch demonstrates that 'PMC-imitation' can also be used in contexts where the imitating part has a different speed has than the reference part.

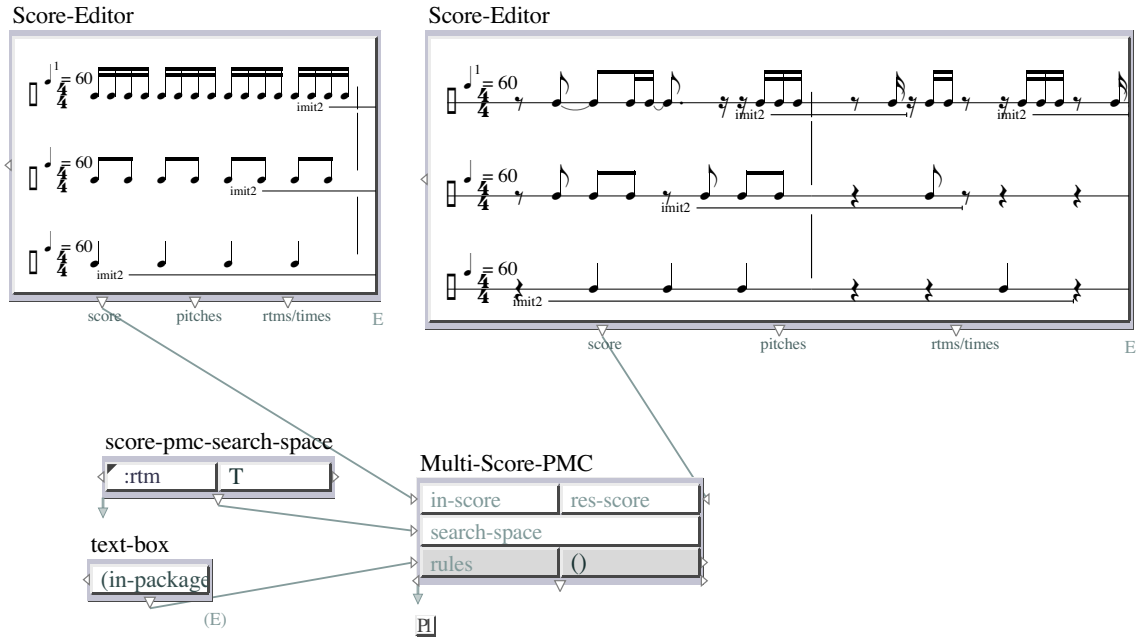

Figure 2.120: 07-RTM-imitation2

2.5.7 Expression-Access

2.5.7.1 Basic-Expression-Access

This is a basic example demonstrating how to access with the rule syntax expression information attached in the input score. Different kinds of profiles are defined with the help of accents and slurs. The textures are given contrasting profiles. Slurred texture uses small intervals and the accented texture, in turn, uses large leaps. Furthermore, we use a collection of supplementary rules where, for example, each group of four notes (i.e., each beat) uses the same set class, namely 4-7.

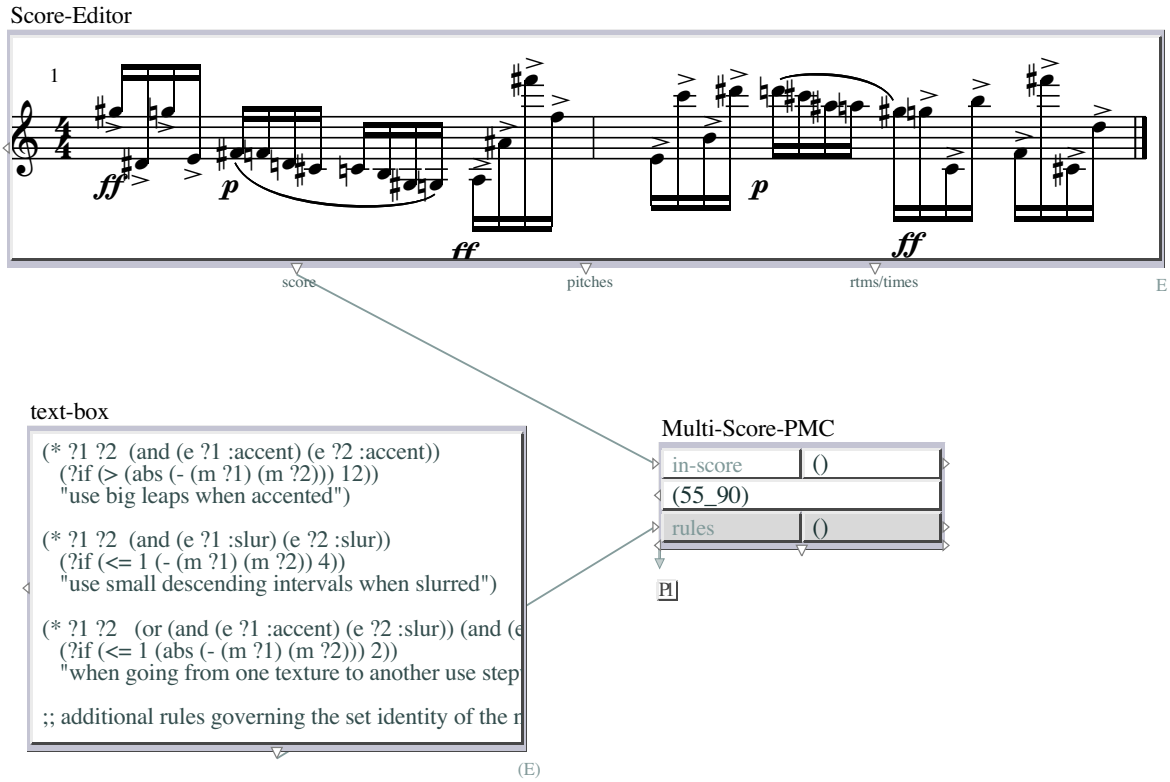

Figure 2.121: 01-basic-expression-access

2.5.7.2 Advanced-Expression-Access

Next, we give an advanced example of the use of ENP-expressions to define different kinds of melodic contours and textures. The two bpfs found in the score (1) are used in the rules to create melodic contours. The group named ”repetition” (2) is used to indicate the span of a repetitive gesture.
Note also that we use a micro tonal scale (see the search space definition) and a harmonic series (see the rules) to create more interesting tonal landscape.

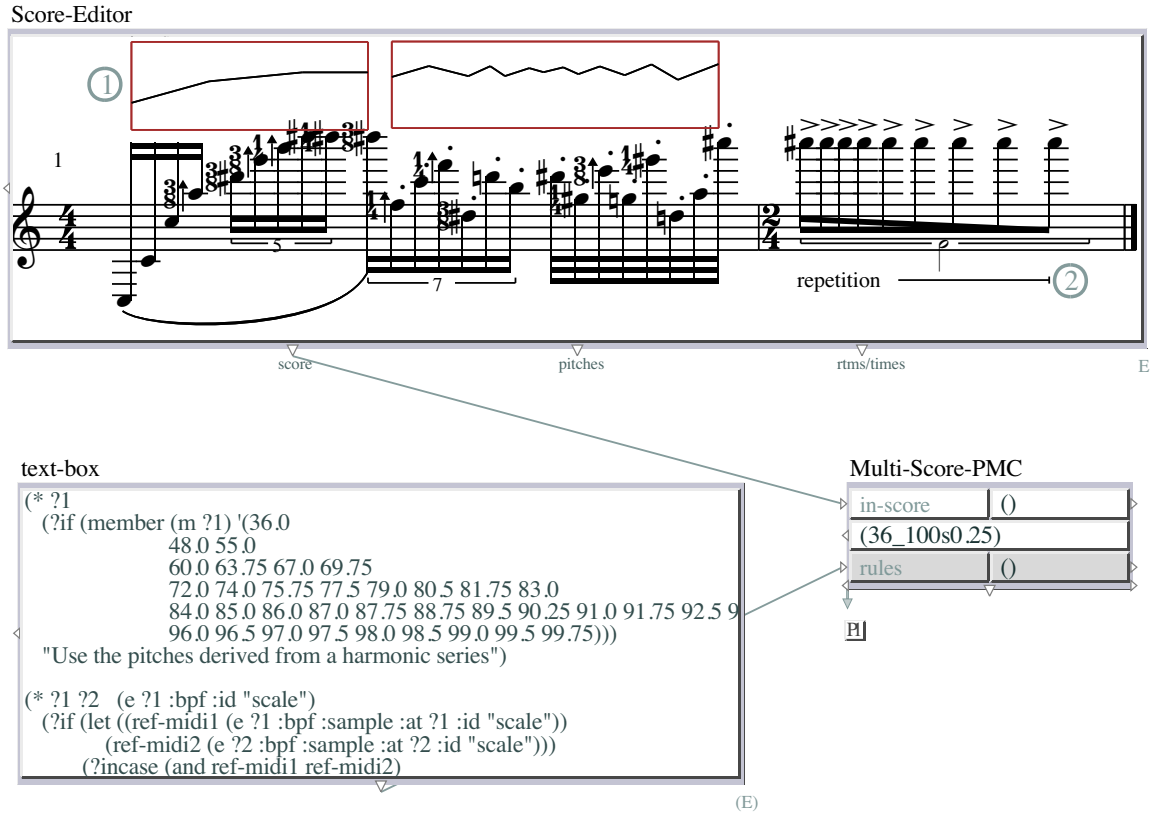

Figure 2.122: 02-advanced-expression-access

2.5.7.3 Sample-Score-BPF

The two break-point functions inside the Score-bpf (1) are used to define an area inside which the passage of three note chords must reside. The individual break-point functions are named (”high” and ”low” and also colored in red and blue respectively). Furthermore, a ’switch’ box (2) can be used to change the pitch class set identity of the chords. This, in turn, is passed to the ’Multi-Score-PMC’ using the ’pwgl-value’ box (3).

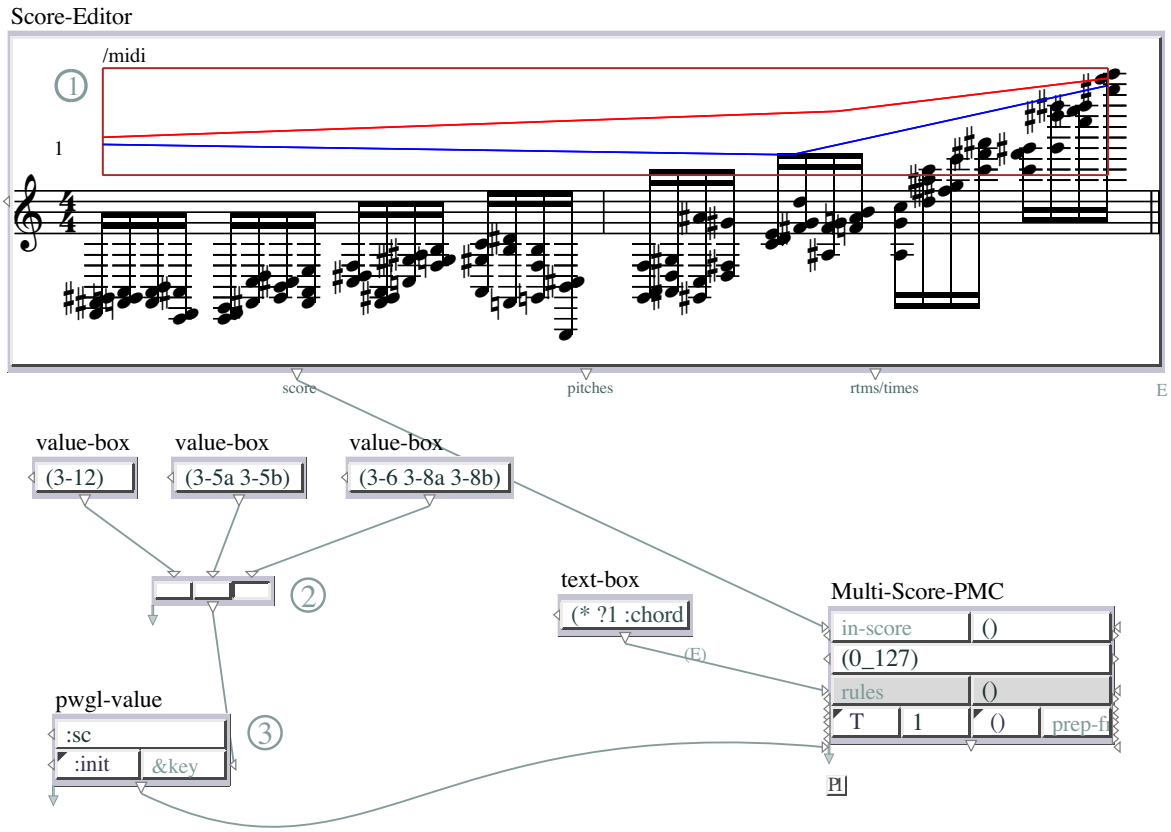

Figure 2.123: 03-sample-score-bpf

2.6 PC-Set-Theory

2.6.1 Exploring-ICV

Here we explore the ICV (interval-class vector of set-classes) to filter set-classes (SC) with specific interval-class properties. E.g. we could ask for all hexachords that exclude interval-class 6 (tritone).
'scs/card' gives us all SCs according to given cardinality (1). Then we choose the interval-class from 1 to 6 to be explored (2) in relation to the number of interval-classes to be found in the vector (3).
The results are represented here as SC-names (4) and corresponding prime forms as midis (5) starting from middle C (60).

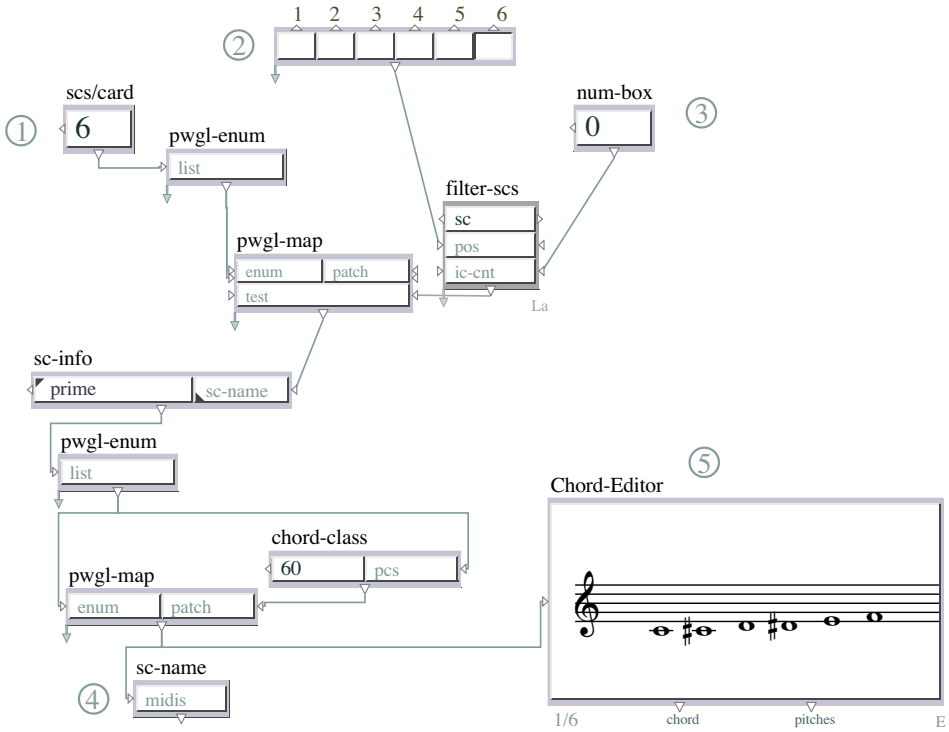

Figure 2.124: 01-exploring-icv

2.6.2 Subsets-Distribution

Here our PC-set-theoretical package is used in combination with the Multi-PMC search engine. A search is used to find the subset distribution of a given superset (1). The cardinality of the desired subsets is given in (2). The search (3) looks for all possible permutations based on the pitch-class contents of the prime-form of the superset (see the 'search-space' input). In order to avoid redundant solutions a rule is given that forces all solutions to be in ascending order ('rules' input).
The result is processed and sorted according to the subset distribution in the abstraction 'build-statistics' (4). The final result is found in the 'text-box' (5), where each sublist or row gives the number of subsets found, the subset name, and modulo 12 transposition offsets as a list.

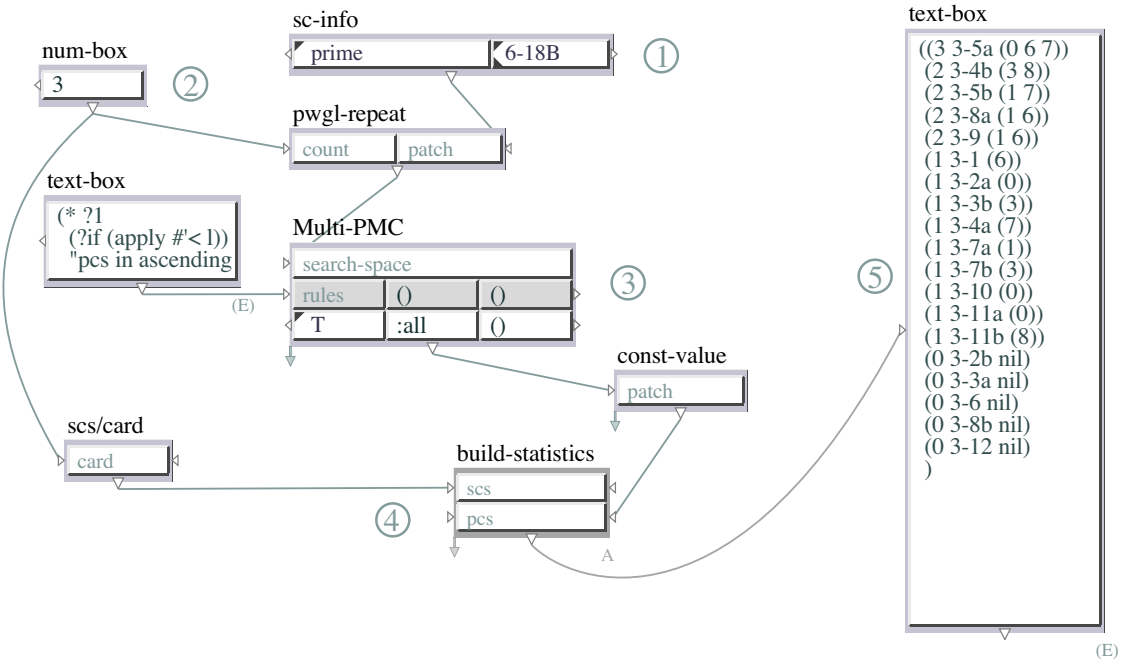

Figure 2.125: 02-subsets-distribution

2.6.3 Supersets-Distribution

This patch is similar to the previous one. This time we look for all possible supersets that contain a given subset (1). The cardinality of the desired supersets is given in (2). The search (3) uses as a search-space that consists of pitch-classes that are not found in the original prime-form (see the 'set-difference' box). We also remove redundant solutions by forcing the results to be in ascending order.
In (4) we append the prime-form to the search results which results in all possible supersets that contain the given subset. The final result (6) is built in the 'count-scs' abstraction (5). This result contains sublists that give the number of transpositions of a found superset containing the subset, the superset name, and modulo 12 transposition offsets.

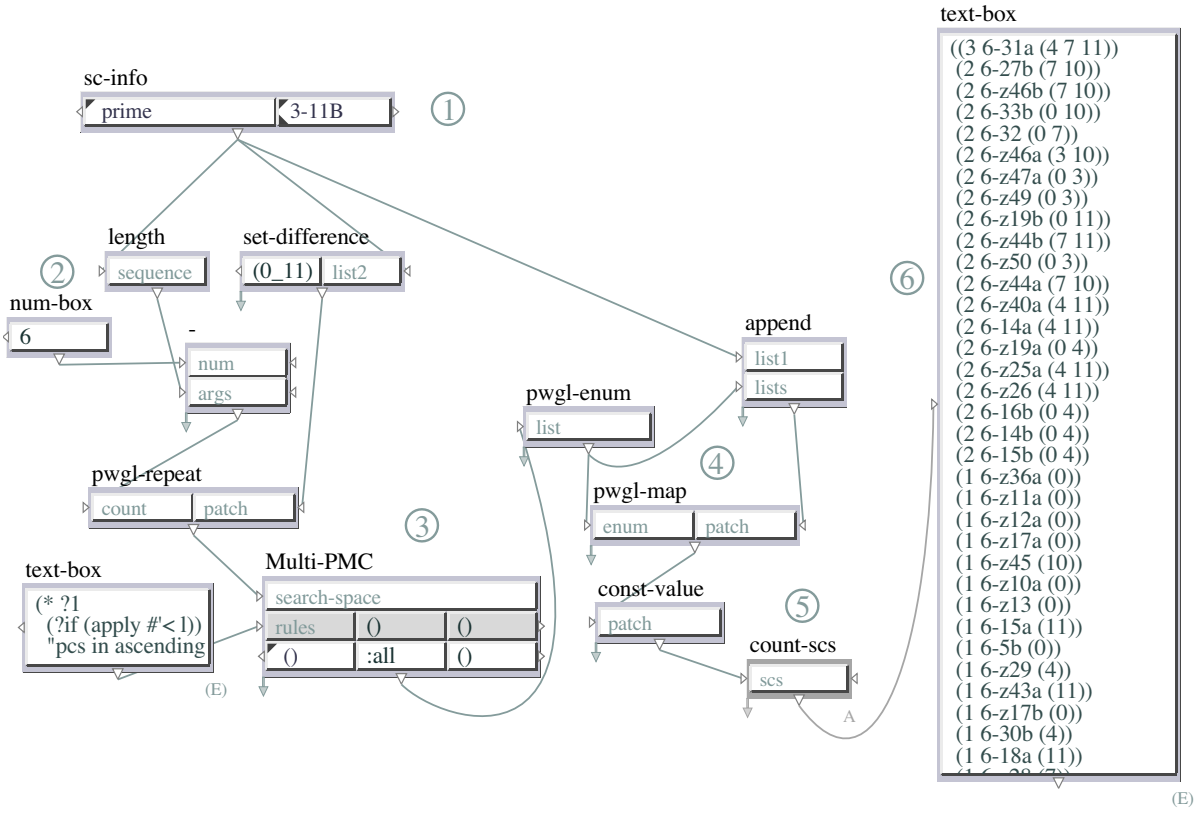

Figure 2.126: 03-supersets-distribution

2.7 Synth

2.7.1 Introduction

A PWGL synth patch is a is a graph structure consisting of boxes and connections. Boxes, in turn, can be categorized in two main box types from a synthesis point of view.

2.7.1.1 Synth Boxes

The first box type consists of ordinary PWGL boxes. These boxes can be found at the leaves of a synthesis patch and they are typically evaluated once before the synthesis patch is run. A special case of this category are sliders which can dynamically change the current value while the synthesis is running.
The second box type consists of boxes, marked with an 'S', that represent synthesis boxes that are used for the actual sample calculation. 'S' boxes support vectored inputs and outputs. Mono signals are only a special case where the vector length is equal to 1. A synthesis patch always contains a special 'S' box, called 'synth-box', at the root of the graph, which represents the output of the sample calculation. This output can either be

sent to audio converters in real-time, or the output can be written to a file.

2.7.1.2 Multichannel Signals

Several tutorial sections (for example 'Vector' and 'Copy-synth-patch') demonstrate how multichannel signals are represented in our system. The patches give a visual clue that helps to distinguish between mono signals (vector length is equal to 1) and vectored signals. The connections between vectored boxes are drawn using a thicker line width and a stipple pattern that contains holes. The vector length is specified by the inputs at the leaves (i.e. inputs which are not connected to a 'S' box) of the patch. These inputs can be Lisp expressions (typically lists) or slider-banks which allow a separate real-time control of each individual vector element. If the lengths at the inputs differ, then the shortest vector determines the current vector length.

2.7.1.3 Developer Tools

Two programming tools allow the user to extend PWGLSynth with new C++ modules, Extension Developer Kit and Visual Patch Compiler. In the first one the user operates with ordinary textual C++ programming. The system also allows to reuse in the code any existing synthesis modules. The Extension Developer Kit plus its documentation can be found in:
In the second tool, Visual Patch Compiler, the user first defines visually a synthesis patch with the help our abstraction scheme. The abstraction is then automatically translated to a binary file that represents the new synthesis box. An example patch is given in this tutorial.

2.7.2 Basic

2.7.2.1 Sine

This is a basic real-time synthesis patch with a 'sine' module and a 'synth-box'. A synthesis patch must have a 'synth-box' which represents the output. The inputs of the 'sine' module are static after the synthesis engine has started. For a dynamic case see the next tutorial.
When the synthesis engine is running the current synth status information is shown in the lower part of the 'PWGL output' window.

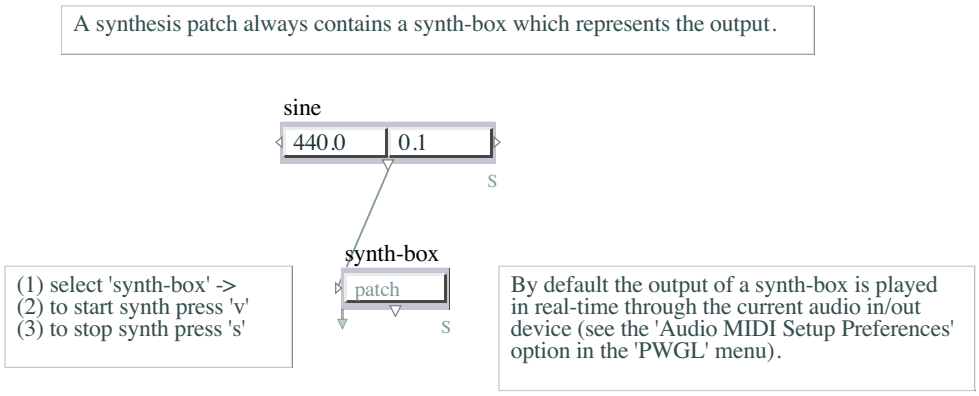

Figure 2.127: 01-sine

2.7.2.2 RT-Sliders

In this patch sliders are used to control in real-time the ’freq’ and ’amp’ inputs of the ’sine’ box.
All synth box inputs can be connected automatically to a slider using the input-box popup menu-item ’add-slider’. This will assign the title, range-values, the current value, etc. of the slider according to the database provided by the synth input-box.

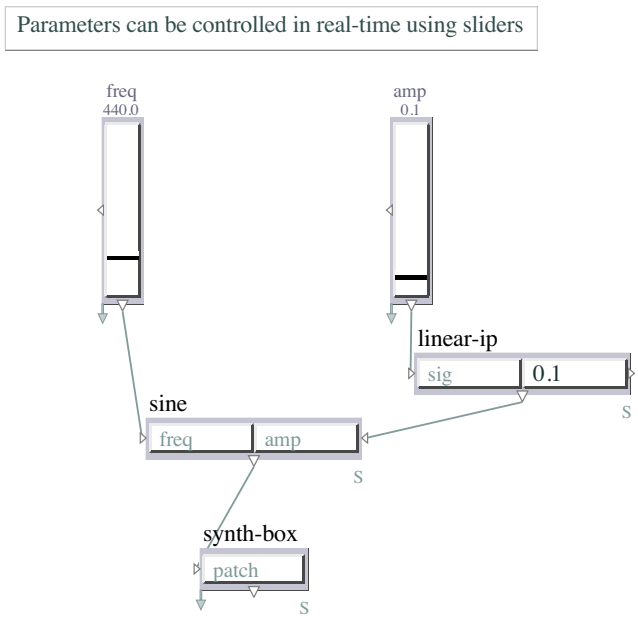

Figure 2.128: 02-rt-sliders

2.7.2.3 Vibrato

This example contains two 'sine' modules where the upper one adds a vibrato to the lower 'sine' module. There are four real-time sliders that control: vibrato frequency and amplitude (upper 'sine' module); the main frequency and amplitude (lower 'sine' module).

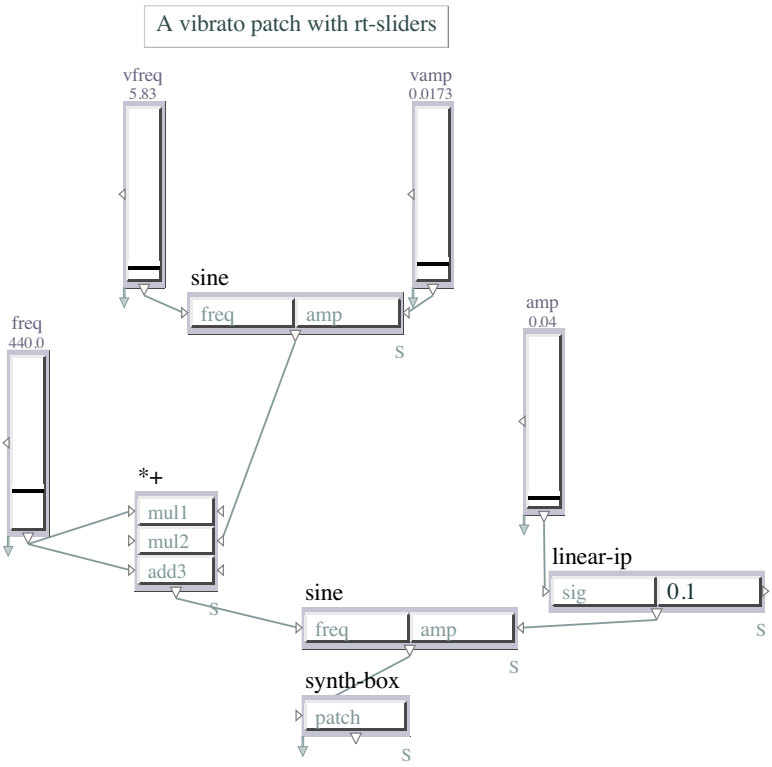

Figure 2.129: 03-vibrato

2.7.2.4 File-Mode

This example is a non real-time patch where the 'synth-box' has been extended so that it contains four inputs. The last two specify: (1) the output mode–in our case ':file'–which means that the output is written to a file; (2) the last input has the value '0.2', which gives the length of the resulting sound file in seconds. Other options dealing with the sound file, such as file format, bit-width, sample rate, etc., can be given in the box-editor (the box-editor can be opened by a double-click or with the box popup menu).

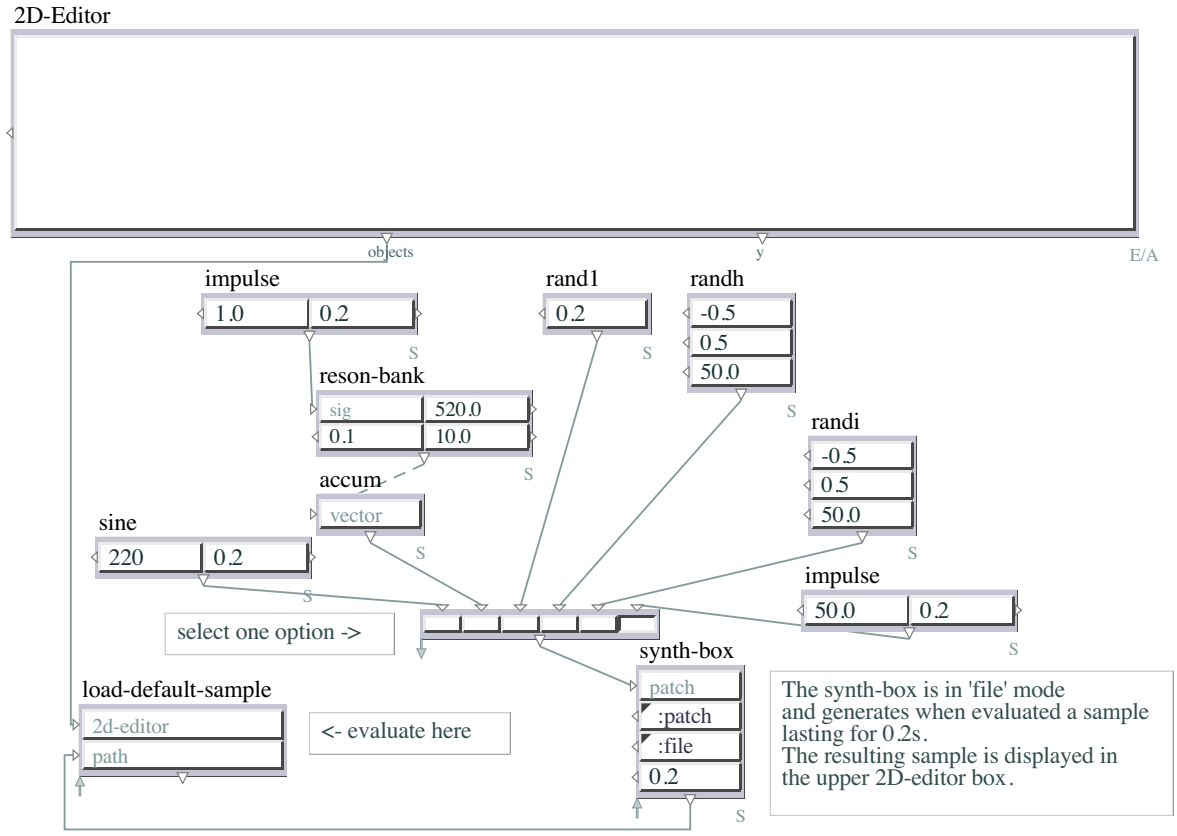

Figure 2.130: 04-file-mode

2.7.2.5 Envelope

This patch demonstrates how envelopes can be realized in our system using the 'envelope-trigger' box. There are two options: (1) a static list of x-y values; (2) a visual break-point function where the points can be edited directly in the patch.
This patch differs from the previous ones as it does not generate any output when the 'synth-box' is started. To get sound the user must trigger the envelope either by clicking on the second input, called '<<trig>>', or by pressing '1' from the keyboard.
In PWGL the numerical keys 1-5 are reserved for triggers. All boxes with a trigger–typically these boxes have a button called '<<trig>>'–can be assigned a number that is given as a user-string in the box-editor.

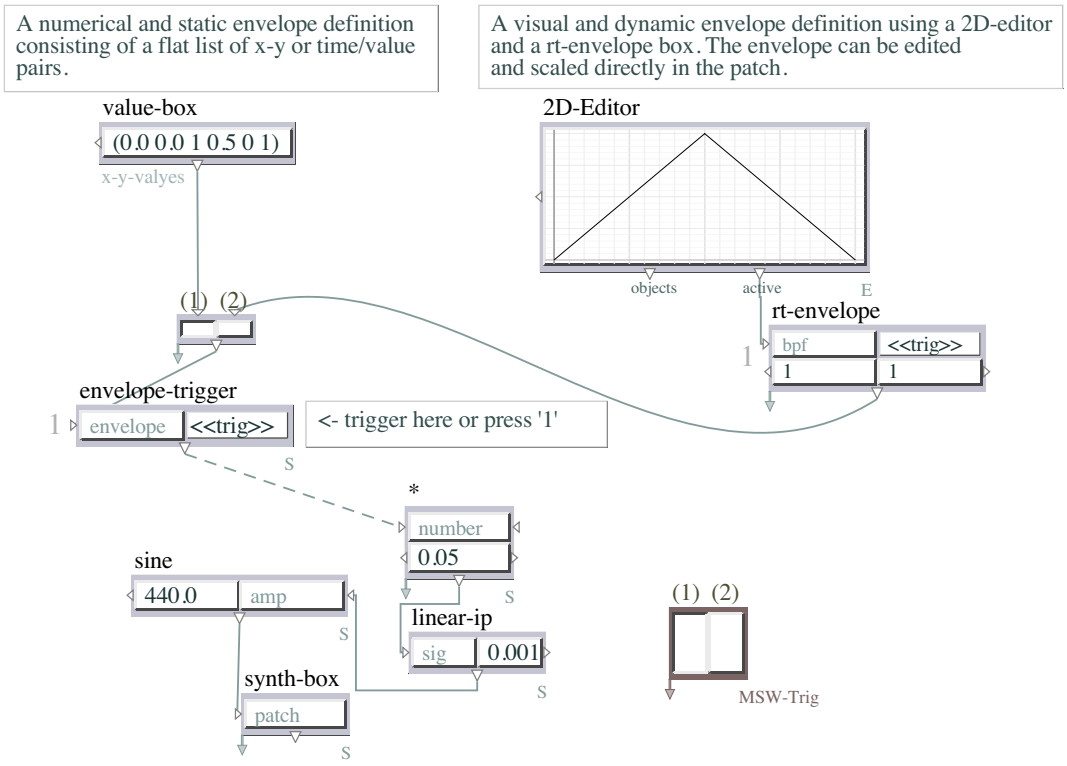

Figure 2.131: 05-envelope

2.7.2.6 Sample-Load

Samples can be loaded to PWGL using lisp code, using the ’2D-constructor’ box, or in the 2D-editor. This example utilizes the ’2D-constructor’ box (note that the first input of the ’2D-constructor’ box is here’:sample’). There are two options: (1) the second input is (), and thus the system opens a file-dialog where the user can choose the file to be loaded; (2) the second input contains a pathname, which is used to load the file.

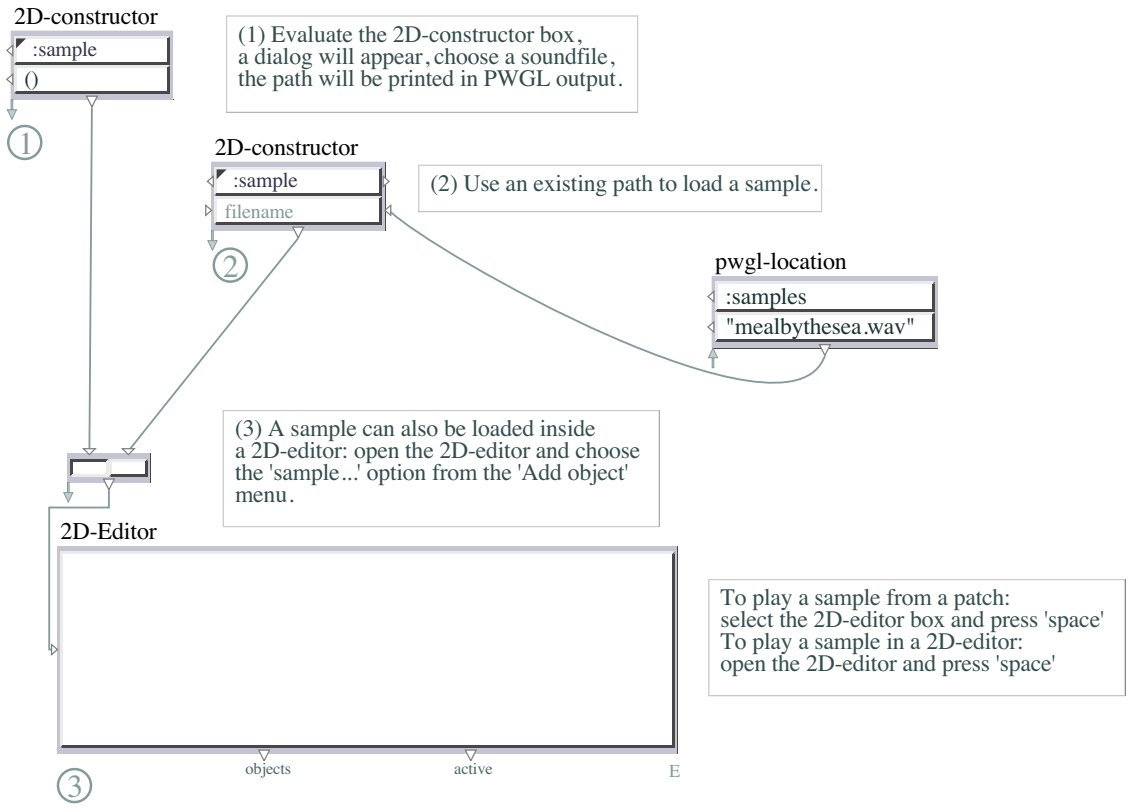

Figure 2.132: 06-sample-load

2.7.2.7 Sample-Play

In this example a sample is first loaded into the system. In order to play this sample from a patch we use typically the 'sample-player' box having four required inputs: 'sample' is the unique ID number that is assigned to every sample when it is loaded (here we use '2D-sampleid' to access this ID), 'fscaler' is a scalar that determines the playback speed (1.0 is the original speed, negative values reverse the signal); 'amp' scales the amplitude of the sample; '<<trig>>' is used to trigger the playback. The optional input 'offset' gives the start position in seconds where the playback should start when the sample is triggered.

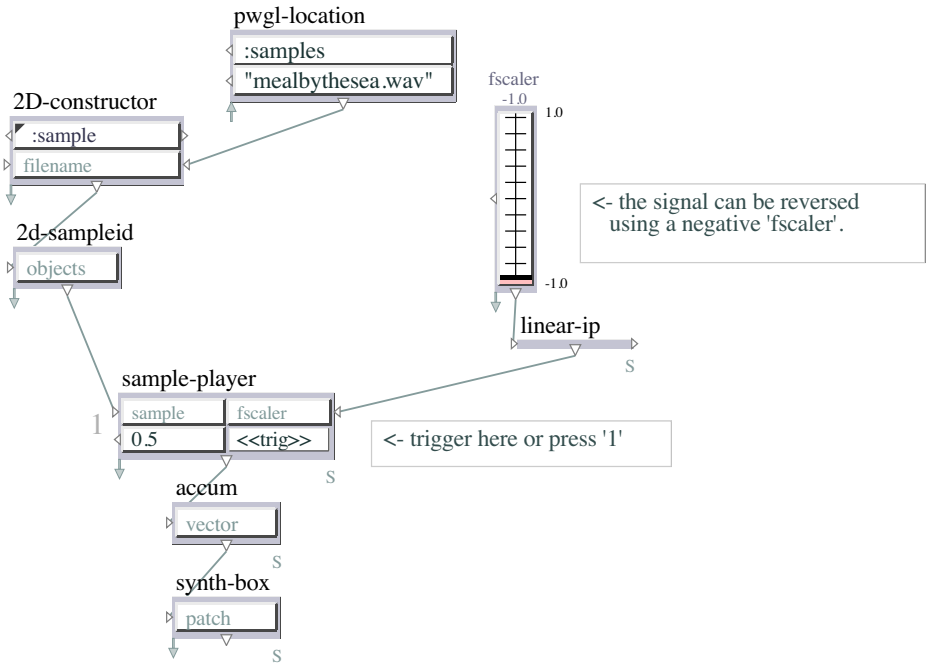

Figure 2.133: 06b-sample-play

2.7.2.8 Interpolation

This patch demonstrates the various interpolation methods you can use to smooth control data. First, a step response is generated using an impulse oscillator, delay and an integrator (1). Interpolator boxes (2) respond to signal changes, which are detected in this case by the detect-steps box. The step response is patched directly to output, along with three interpolated responses. These are combined into a vectored signal which you can see in 2D-Editor once you evaluate 'load-default-sample' (3).
You can then visually see the different response curves:
- No interpolation - Exponential interpolation - Linear interpolation - Parabolic interpolation

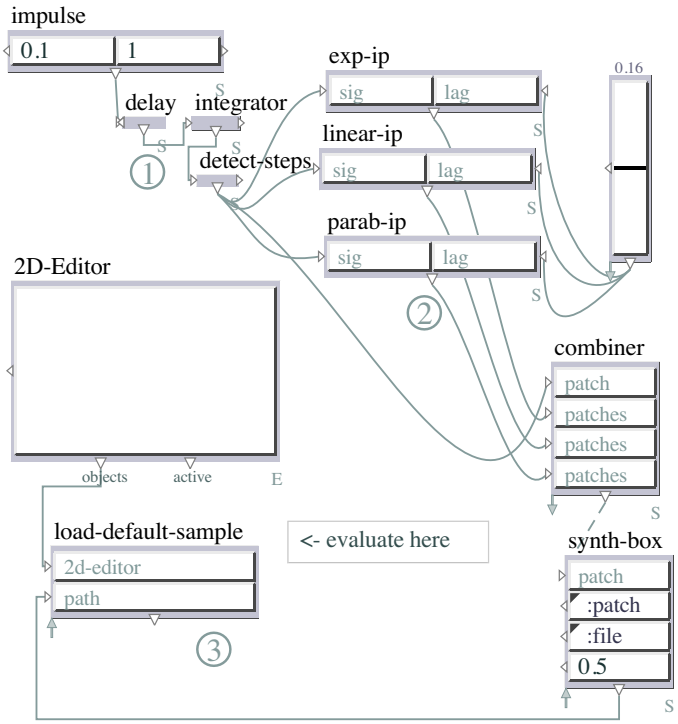

Figure 2.134: 07-interpolation

2.7.3 Vector

2.7.3.1 Basic-Vector

2.7.3.1.1 Slider-Bank-Drummer

A bank of 4 resonators (’reson-vector’) is excited with a bank of 4 impulse generators (’impulse-vector’). The frequency input of the ’impulse-vector’ box is controlled with a slider-bank, and the amplitudes of the impulses are controlled with a bank of sine wave oscillators (’sine-vector’). The slider-bank input of the ’freq’ input of the ’impulse-vector’ is of special interest as it allows to control in real-time the frequency of each impulse generator individually. The other inputs of the ’reson-vector’ box (i.e. frequencies, amplitudes, and bandwidths) are given as static Lisp lists, each containing 4 elements. The output of the ’reson-vector’ box is connected to an ’accum-vector’ box that accepts as input any vectored signal and mixes it to a signal that has the length that is given by the second input (here it is 2) The accumulator iterates through the elements in the input vector, adding each one to the corresponding element of the output vector. When the length given by the second input is reached, the accumulator starts again from the first element of the output vector. Thus in this case the final output is a stereo signal,

impulse generators 1 and 3 are heard on the left while generators 2 and 4 are heard on the right.

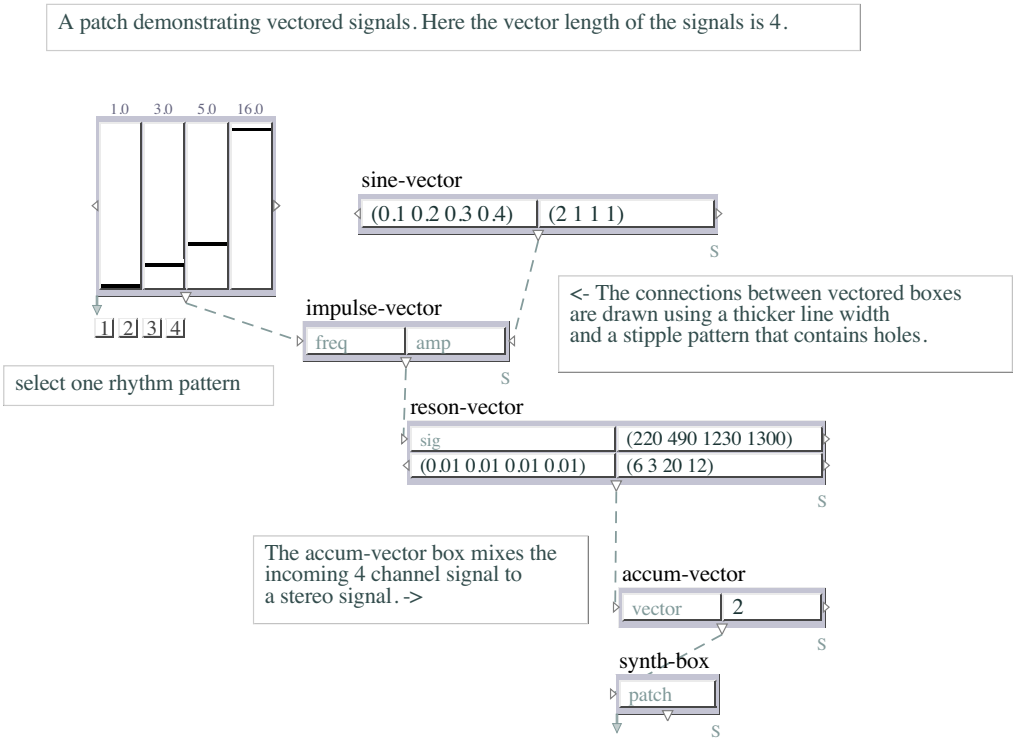

Figure 2.135: 01-slider-bank-drummer

2.7.3.1.2 Randi-Bell

This patch example demonstrates how the PWGL environment can be used to calculate input values for vectored 'S' boxes. The two first inputs, 'low' and 'high', of the 'randi-vector' box contain special PWGL shorthand expressions, '(14*(0.995))' and '(14*(1.005))', for generating lists (here we get 2 lists of 14 elements consisting of the values 0.995 and 1.005). The third input of the 'randi-vector' box, called 'freq', is connected to an 'interpolation' box, that returns a list of 14 values (the result of interpolating values from 5.0 to 20.0). Thus the vectored output of the'randi-vector' has 14 elements which are fed to the first input of a 'mul-vector' box. The second input of the latter box is connected to a standard PWGL 'value-box' that returns a list of 14 frequency values. The output of the 'mul-vector' box consists of 14 frequency values where each value is individually modulated by an interpolating random number generator. This output is connected to the 'freq' input of a 'reson-bank' module. Like 'reson-vector' given in the previous example, 'reson-bank' is a bank of resonators. The first input is different, however, as it accepts only a mono signal (here a simple impulse) instead of a vectored

input. The other inputs of the 'reson-bank' module, amplitudes and bandwidths, are lists of 14 values. Finally, the output of the 'reson-bank' box is mixed to a mono signal with a 'accum' box.

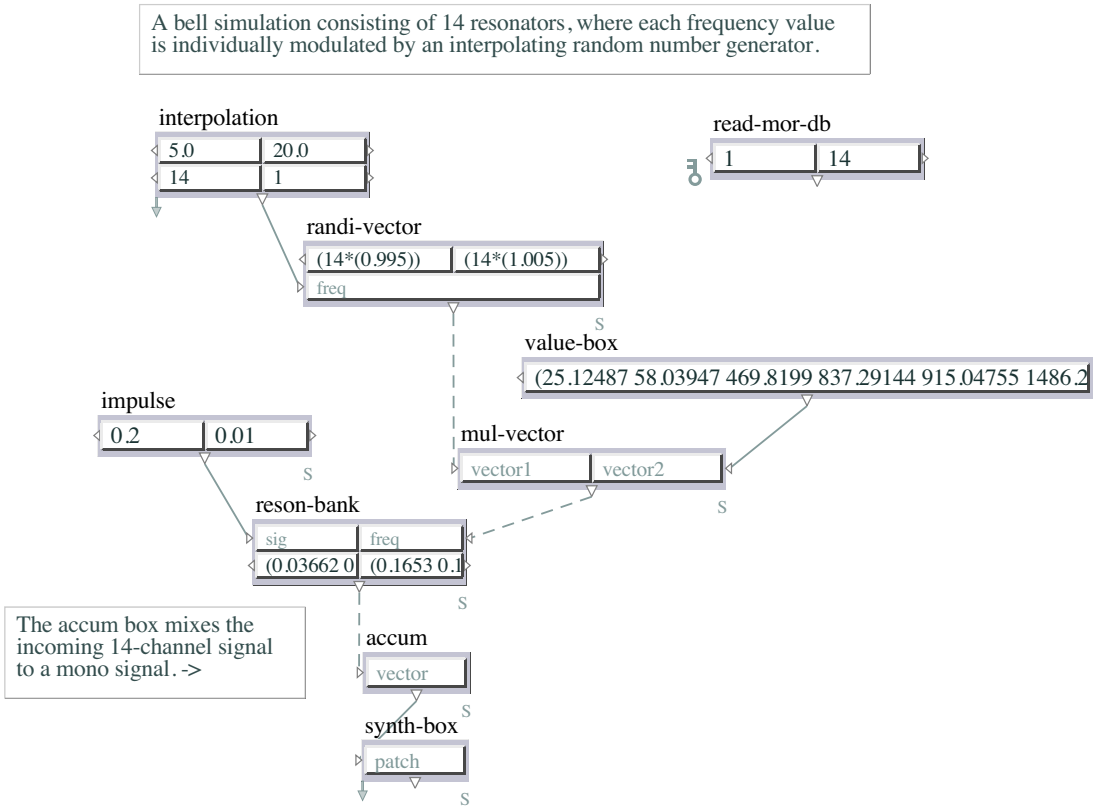

Figure 2.136: 02-randi-bell

2.7.3.1.3 Combiner

Vectors can be combined to a single vector using a box called 'combiner', that can have an arbitrary number of inputs. A typical application of this box is when the user wants to combine several mono boxes so that the resulting vector can be fed to a vectored box. Thus in this example a 'combiner' box combines two mono random number generators, and the resulting vector is fed to an 'impulse-vector' box. This vector is in turn mixed to a mono signal resulting in a stream of 7 impulses per second where every seventh impulse is strongly accented (see the first 'freq' input of the 'impulse-vector' box containing the list '(1 7)').

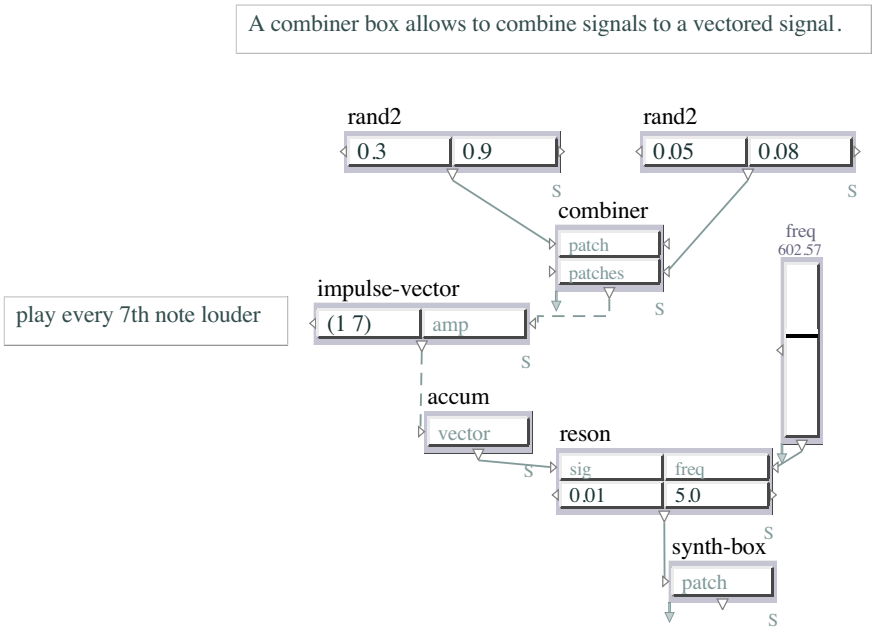

Figure 2.137: 03-combiner

2.7.3.1.4 Indexor

Our next example shows how vectors can be split into sub-vectors by using the 'indexor' box. This box has 3 inputs, 'vector', 'index', and 'len'. The starting point is a bank of 29 resonators. The vectored output is split into two sub-vectors so that vector elements 0-15 (the indexing starts from 0) form the first sub-vector (see the 'indexor' box to the left), and the remaining elements form the other sub-vector (the 'indexor' box to the right). After this both sub-vectors are mixed to 2 mono signals, which are in turn fed to 2 spatialization boxes, called 'stereo-pan'.

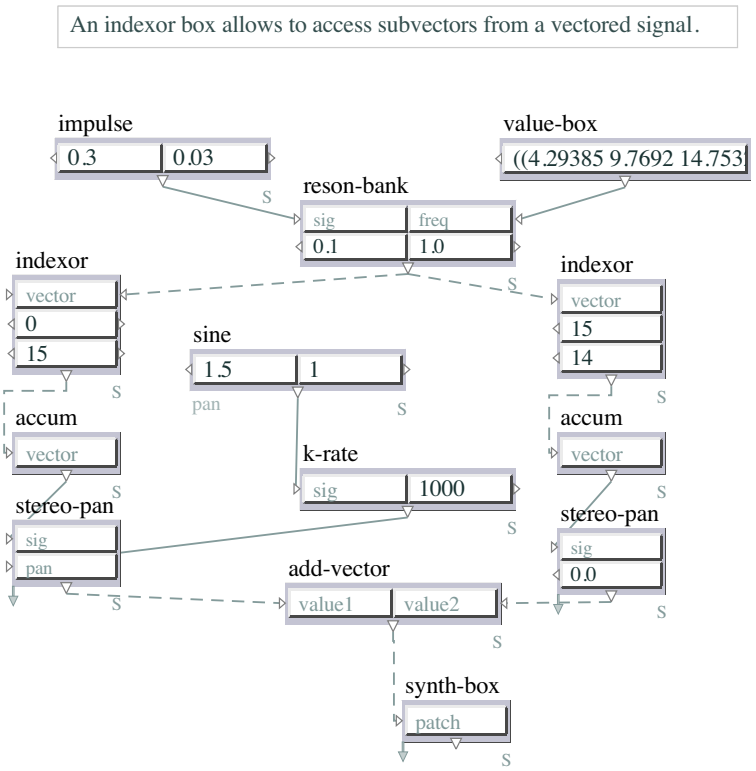

Figure 2.138: 04-indexor

2.7.3.1.5 Envelope-Vector

This example shows how the 'envelope-trigger' box can have a vectored output (an evelope-trigger' box tutorial is also found in the previous section). This output can be connected to boxes that require vectored inputs.
There are two options here: (1) Envelopes that are collected from 2D-Editors are merged to a vector using the box 'combiner'. (2) The envelope data can also be given as a Lisp list. The master switch 'comb/list' can be used to switch between these options.

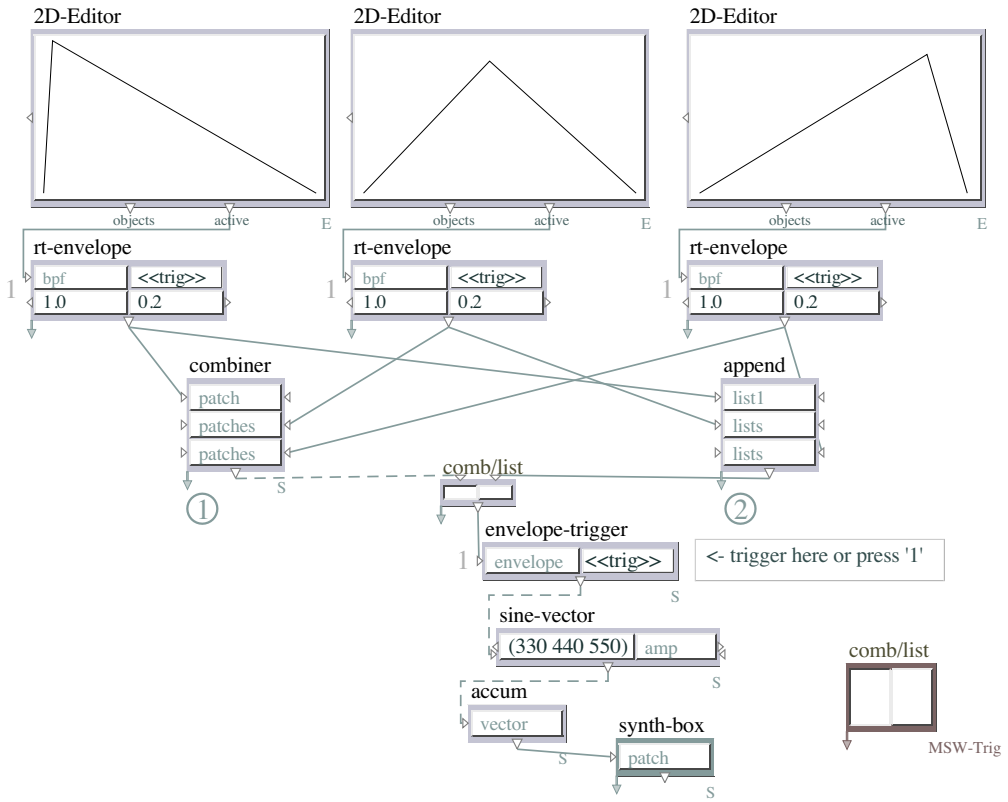

Figure 2.139: 05-envelope-vector

2.7.3.2 Multichan

2.7.3.2.1 Multichan-Drummer

The number of channels at the output depends on the synth box that is connected to the input of the 'synth-box'. This is a basic test patch where the number of channels can be controlled directly using the top-most 'num-box'. This changes the vector length of all the inputs of the 'reson-vector' box.

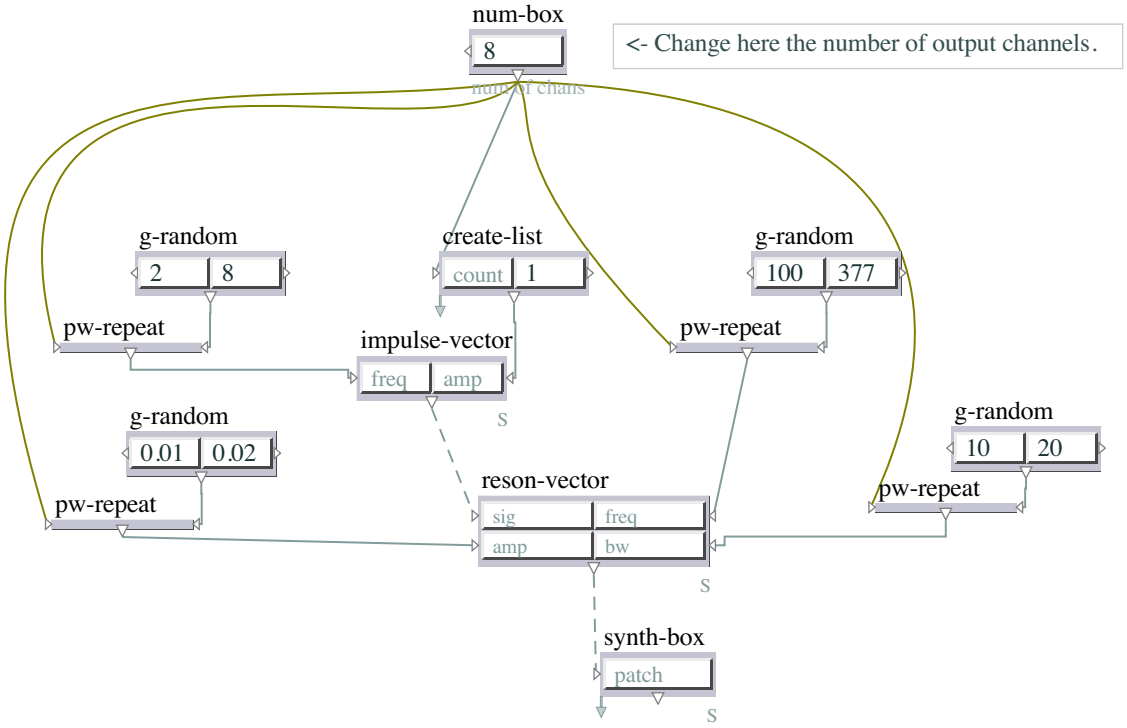

Figure 2.140: 01-multichan-drummer

2.7.3.2.2 VBAP2D

This is another test patch where the output can be switched between: (1) mono output; (2) stereo output; and (3) quad output. In the two latter cases the multi-channel output is created using two 'vbap2d' boxes (see also the 'indexor' example in the previous section).
The number of channels that are output from this box and also the global speaker setup is determined by a companion box called 'vbap2d-conf'. This box must be called once just before the synthesis starts as it creates all necessary data structures that will be used by the vbap2d' boxes while the synthesis is running. In order to guarantee that the configuration is done in a proper order the patch is started here with a 'pwgl-progn' box that evaluates its inputs sequentially (thus the synth will start always after the configuration).
There are here two speaker configurations. In the stereo case the speaker setup is -45 and 45 degrees or (-45 45), and in the quad case the four speakers are positioned at (-35 35 145 -145). Note that stereo is a special case where the speakers must always be positioned at -45 and 45 (in multichannel cases there are no such restrictions).
The patch contains two pan sliders–one for the stereo case and one for the quad case–

that are connected to the azimuth inputs. The sliders can be used to pan the sound source which is here a noise generator.

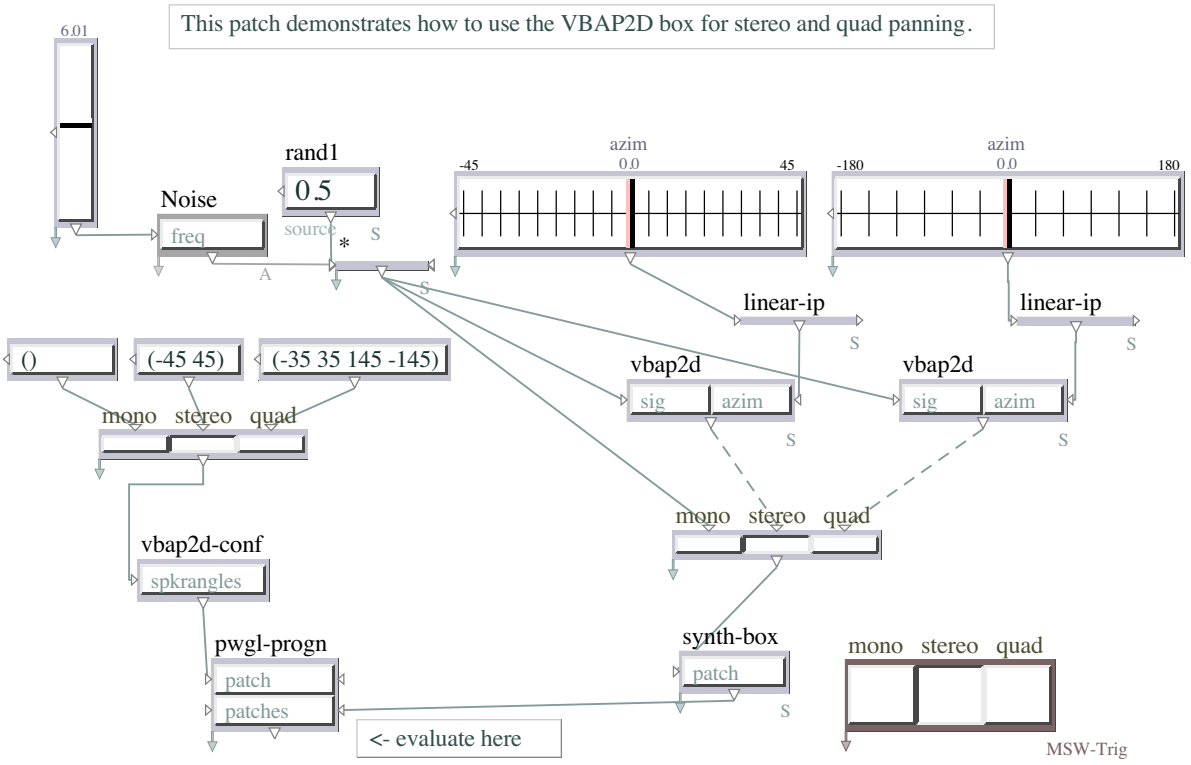

Figure 2.141: 02-VBAP2D

2.7.3.2.3 Combine-Stereo-Signals

In this patch we use two sources that are panned individually with two 'stereo-pan' boxes. The stereo outputs are combined to a 4-channel signal (see the 'combiner' box). Finally the output is mixed down to stereo signal (see the second input of the 'accum-vector' box which is = 2).

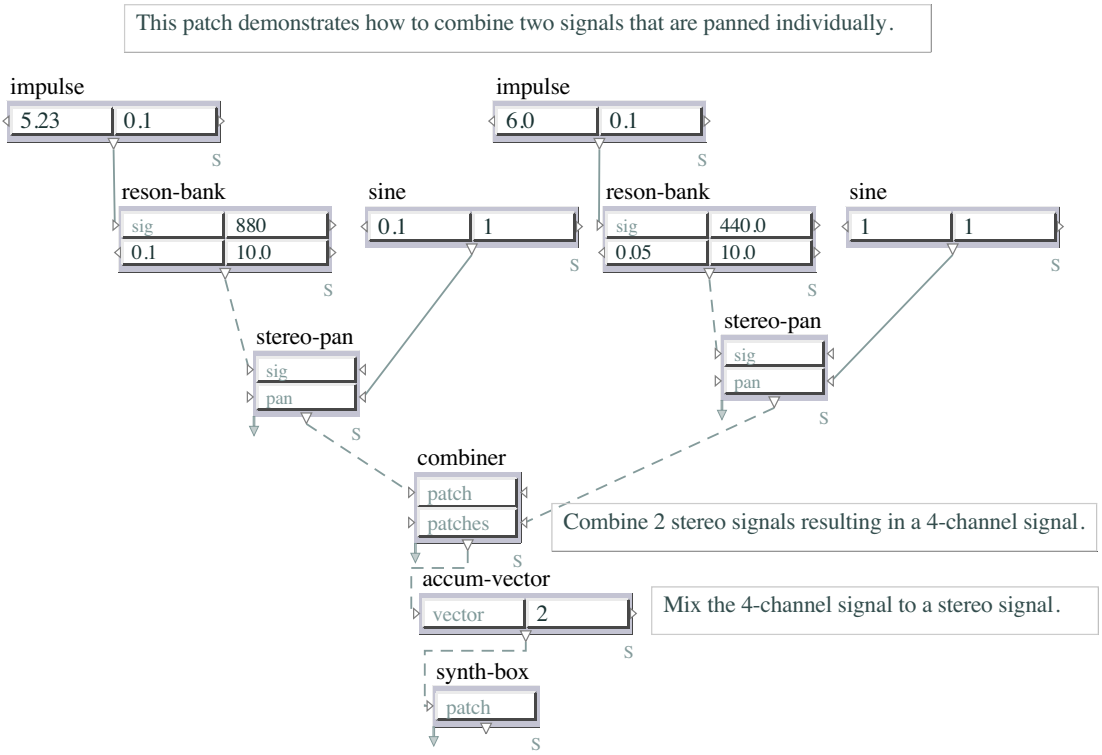

Figure 2.142: 03-combine-stereo-signals

2.7.3.2.4 Distance

This patch simulates with mouse x-y movements a stereo panning/distance effect with the help of a synth box called 'vbap-2d-dist' (the mouse x - see the 'mouse-x' box - movement simulates the panning, while y - see the 'mouse-y' box -movement the distance).
This box has two extra inputs when compared to a 'vbap2d' box: 'dist' that approximates the distance of the source, and 'revsc' that can be used to scale the additional reverb signal. 'vbap2d-dist' scales the output signal according to the 'dist' parameter, and it returns an extra channel that can be used as a mono reverberation output.
The output signals (the stereo signal and the reverb signal) are accessed with 2 'indexor' boxes. The reverb signal is in turn fed to a 'Reverb' abstraction that contains a reverberation sub-patch definition.
This patch is roughly based on the ideas that were originally presented by John Chowning during the 70s.

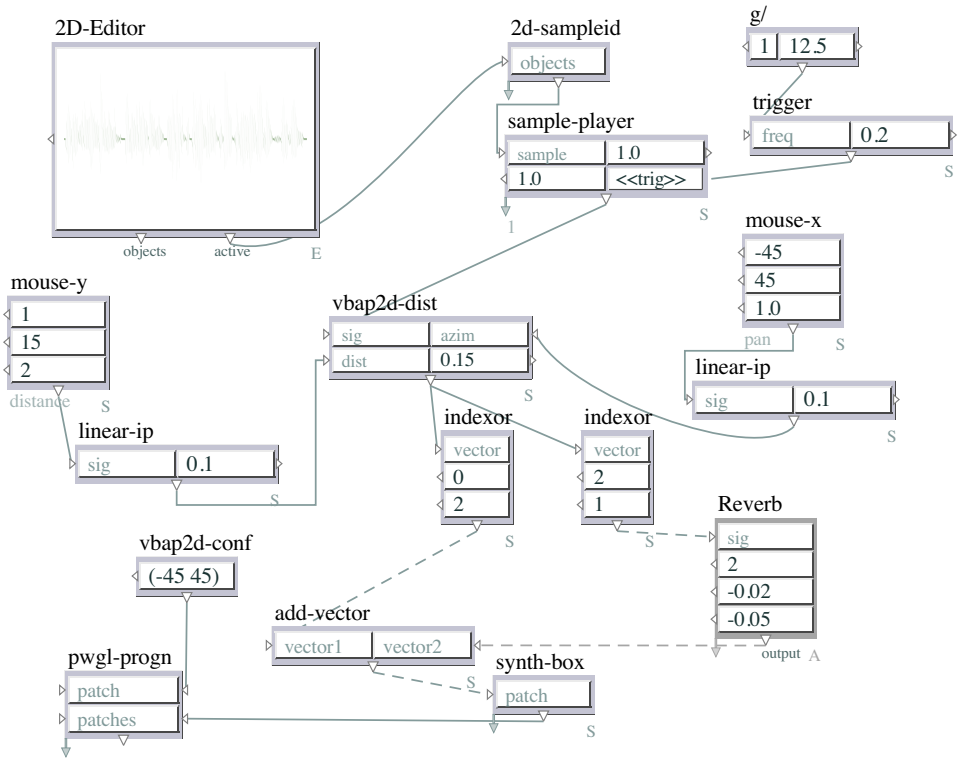

Figure 2.143: 04-distance

2.7.3.3 Vector-Applications

2.7.3.3.1 Intpol-Filterbank

This patch calculates algorithmically all incoming data (i.e. frequencies, amplitudes and bandwidths) for a 'reson-bank' box. The 25 resonators are excited with a noise generator. There are two options: (1) 'interpolation' boxes are used to calculate the data; (2) the frequencies are calculated with a random number generator. These options can be chosen with the master switch box 'intpol/random'.
The resulting signal is modulated–see the 'mod-delay' abstraction–with a delay module where the delay-time input is controlled with a 'sine' box. Finally the original signal from the resonators and the scaled modulated signal are mixed together.

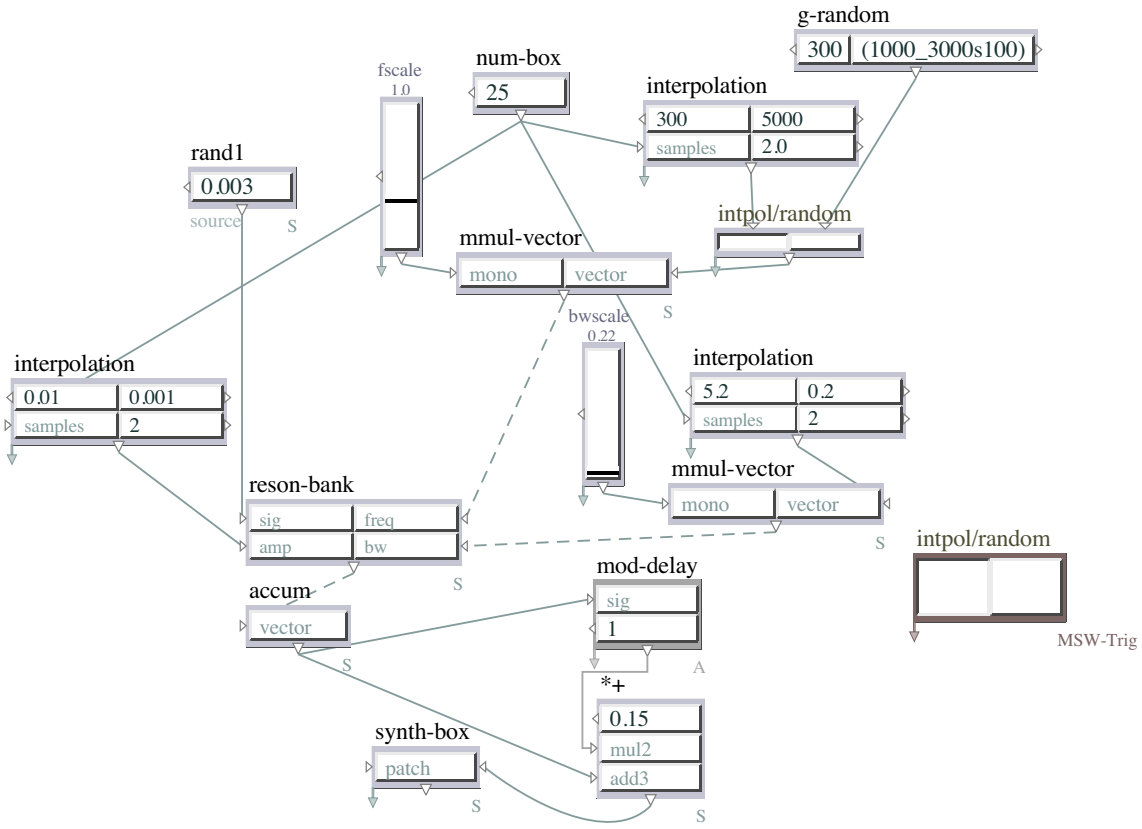

Figure 2.144: 01-intpol-filterbank

2.7.3.3.2 Reson-Mix

This example differs from the previous one as we use here data that has been analyzed from instrument samples. The two cases used are: (1) a piano string, 'pianoA0'; (2) bell, 'rclova-s2'. Both models are excited with a noise source. The balance of the resonators can be controlled with a slider called 'Mixer'.
We give here the data to the 'reson-bank' boxes in a slightly unorthodox form which can be handy in some cases. The data consists of a list of three sub-lists thus defining all the required information for the resonator banks. Thus in this case the inputs 'amp' and 'bw' are ignored

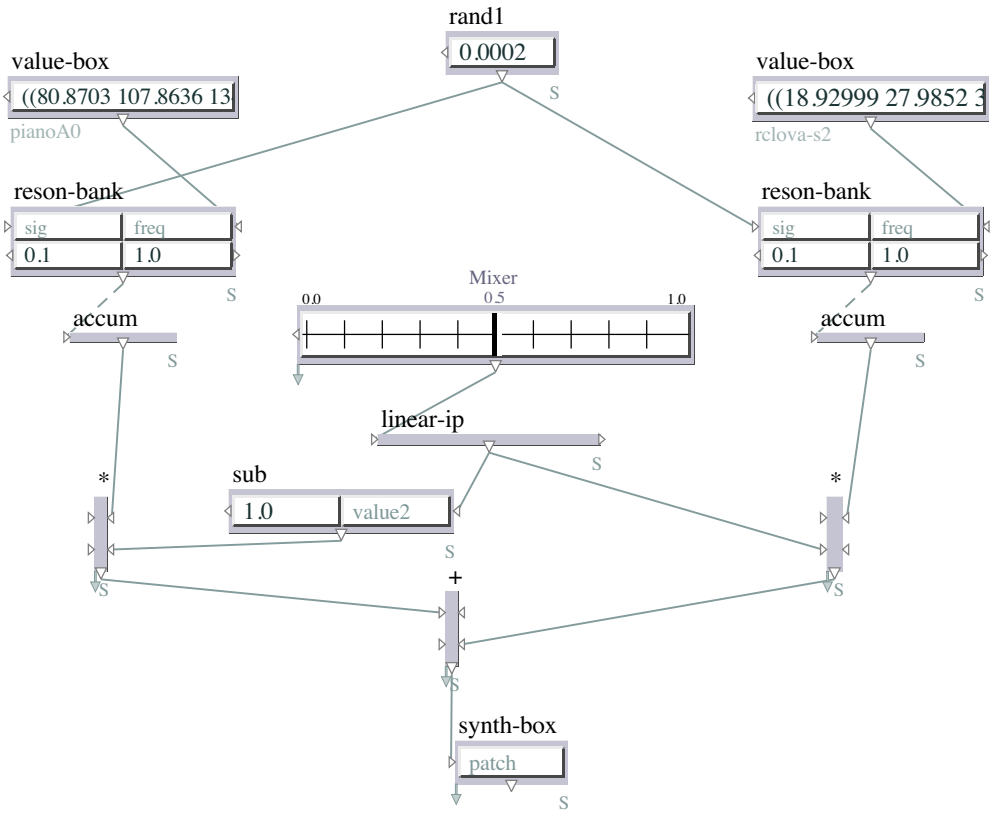

Figure 2.145: 02-reson-mix

2.7.3.3.3 Masterswitch

This is a companion patch with the previous one. The main difference is that here the user can choose the excitation source (either impulse or noise) with the master switch box 'i/n'. Furthermore in the second option the frequencies are multiplied with 2.

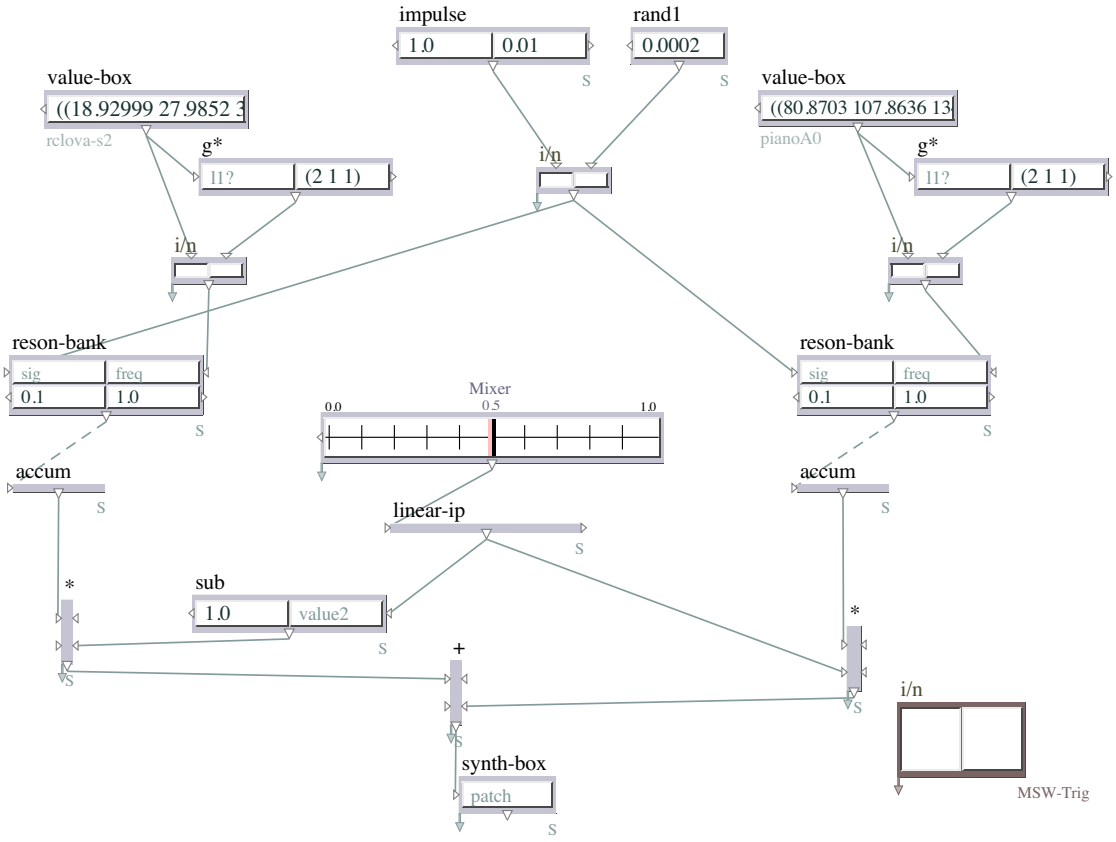

Figure 2.146: 03-MasterSwitch

2.7.4 Copy-Synth-Patch

2.7.4.1 Copy-Synth-Patch

The 'copy-synth-patch' scheme in conjunction with the vectored signal representation is one of the corner stones of our system.
This patch introduces this scheme that allows to combine any collection of mono or vectored boxes. The collection can be copied and the output of the result will be one or several vectored signals. We use for this purpose a special box called 'copy-synth-patch' with 2 required inputs, 'count' and 'patch'. A third, optional, input can be given for a name string. A 'copy-synth-patch' box duplicates a patch connected to the patch input 'count' times. The output of the box is a vector having the length which is determined by the 'count' input (internally the system uses the 'combiner' box).
Here we copy a 'sine' module twice ('count' = 2). This results in a bank of sine oscillators. In order to be able to distinguish between different 'sine' instances we connect the 'freq' input to a 'synth-plug' box that generates automatically pathnames (e.g. ':1/freq' and ':2/freq') that are used to by the 'update-plug-value' boxes. Initial values can be set using the 'copy-patch-index' box (here the 'amp' inputs get the values 0.03 and 0.02).

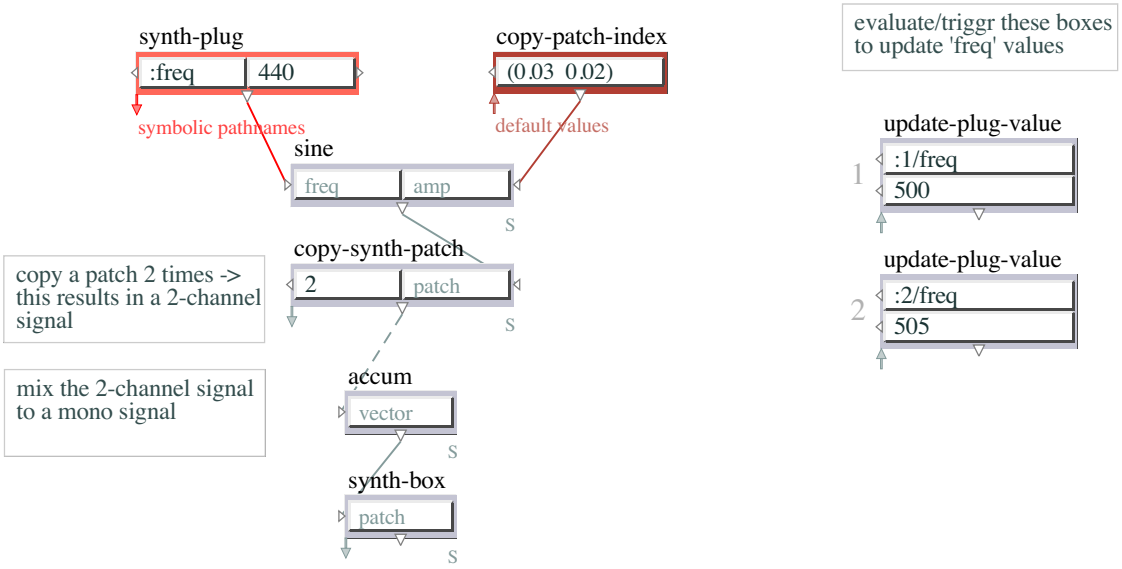

Figure 2.147: 01-copy-synth-patch

2.7.4.2 CSP-Bells

This patch duplicates the 'randi-bell' example given in the 'Basic-vector' section N (1-4) times using the copy-synth-patch (CSP) scheme. The master switch box '1 2 3 4' gives the current number of bells. The abstraction 'stereo bell' contains the bell definition. The 'copy-synth-patch' box returns a vector that has a length equal to N*2 (thus if N = 4, then the vector length is = 8). The final result is a stereo signal (see the 'accum-vector' box).

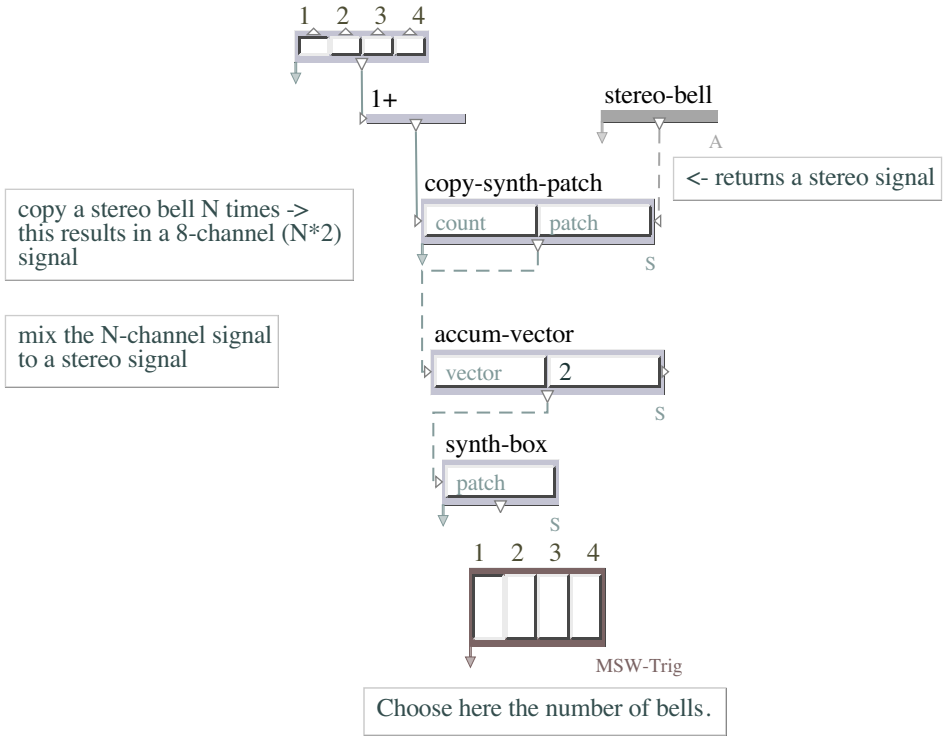

Figure 2.148: 02-csp-bells

2.7.5 Synthesis-Methods

2.7.5.1 Additive

This section presents some common synthesis techniques (e.g. additive, subtractive, fm, formant and granular).
In the first patch a bank of 10 oscillators(see the 'sine-vector' box) is used to generate harmonic series examples. The amplitudes of the individual harmonics can be controlled with a 'slider-bank' box.
There are here two cases: (1) all adjacent harmonics are present (upper row of the 'slider-bank' setups); (2) only even harmonics are present (lower row of the 'slider-bank' setups).
Note that this is a non real-time patch (the synth-box' is in ':file' mode). The resulting signal is shown in the '2D-Editor'.

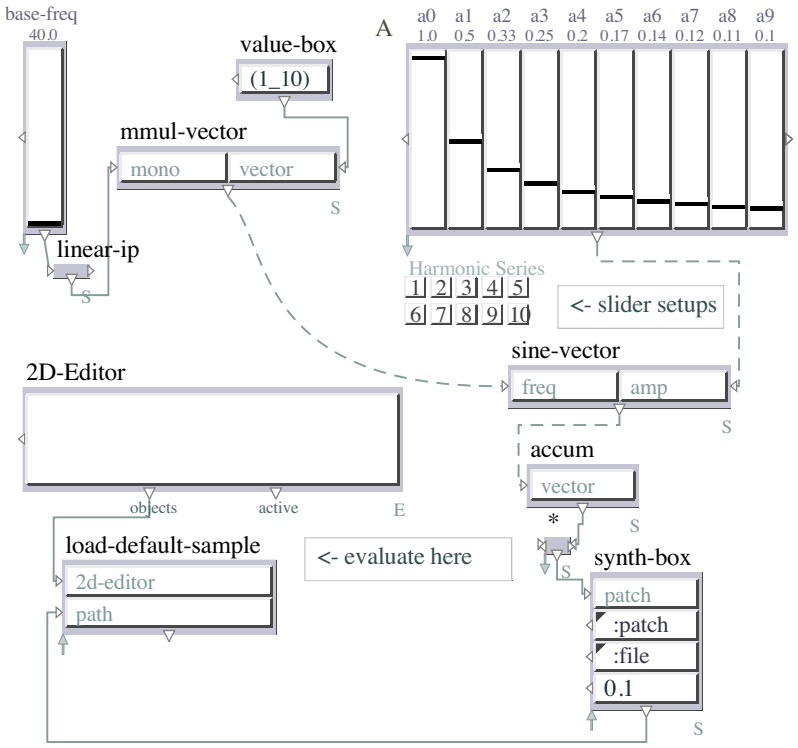

Figure 2.149: 01-additive

2.7.5.2 Subtractive

This patch gives a subtractive synthesis example. A geometrical sawtooth wave is generated by applying a leaky integrator to a band-limited impulse signal ('bl-impulse'). This source signal is then filtered with a 'moog-ladder' filter. The 'q' input is controlled with an envelope that can be triggered by clicking the '<<trig>>' button or by pressing '1'.

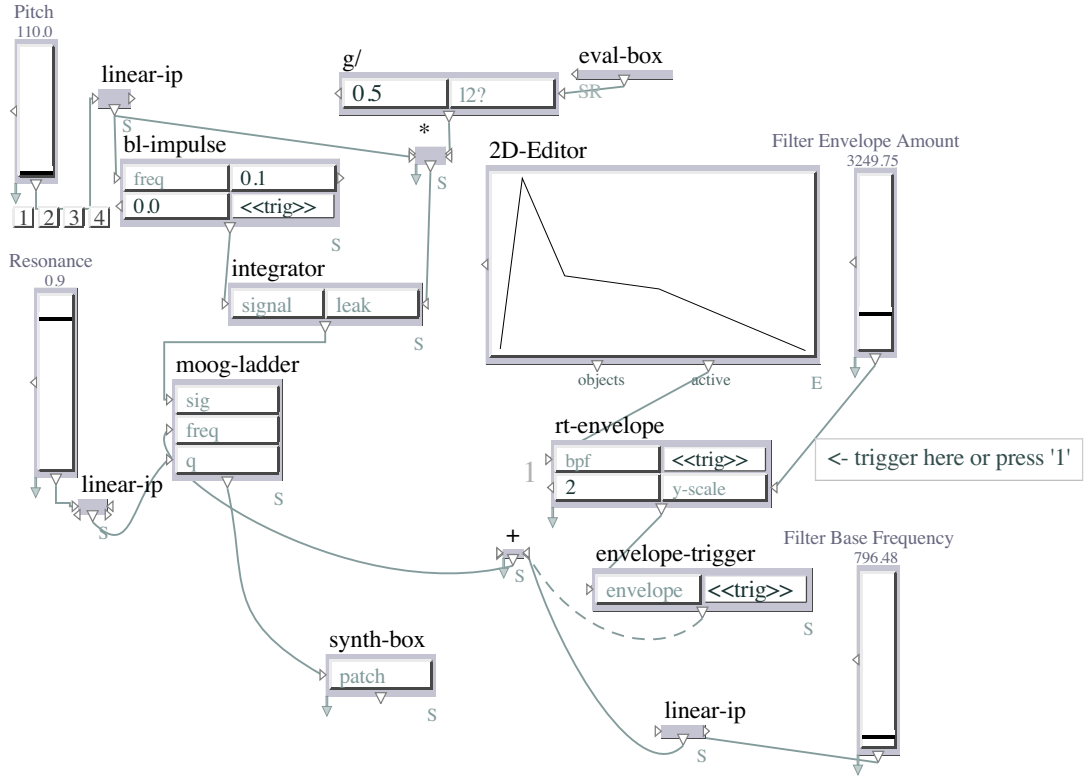

Figure 2.150: 02-subtractive

2.7.5.3 Fm

This patch is based loosely on some fm instrument examples that were originally presented by John Chowning. There are three instruments: (1) trumpet; (2) clarinet; (3) bell. The current instrument can be selected using the master switch box 'trump clar bell'. In order to trigger the envelopes press '1'.

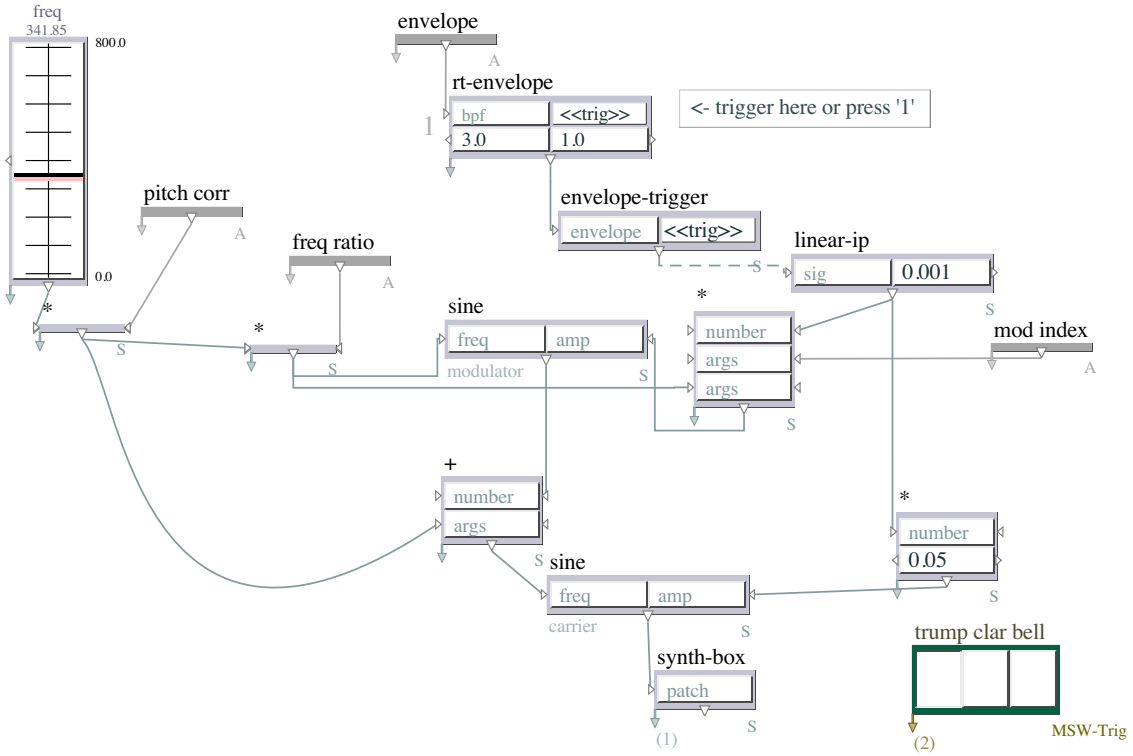

Figure 2.151: 03-fm

2.7.5.4 Formants

This example is based on formant synthesis where we simulate some basic vowels. The source is a 'bl-impulse' box which is filtered by two resonators ('reson-bank') that aim to simulate the two lowest formants of the human voice. The frequency input of the resonators are controlled with the current mouse x-y position. The labels of the vowels 'I', 'E', 'U' and 'A' represent the approximate mouse x-y positions for these vowels.

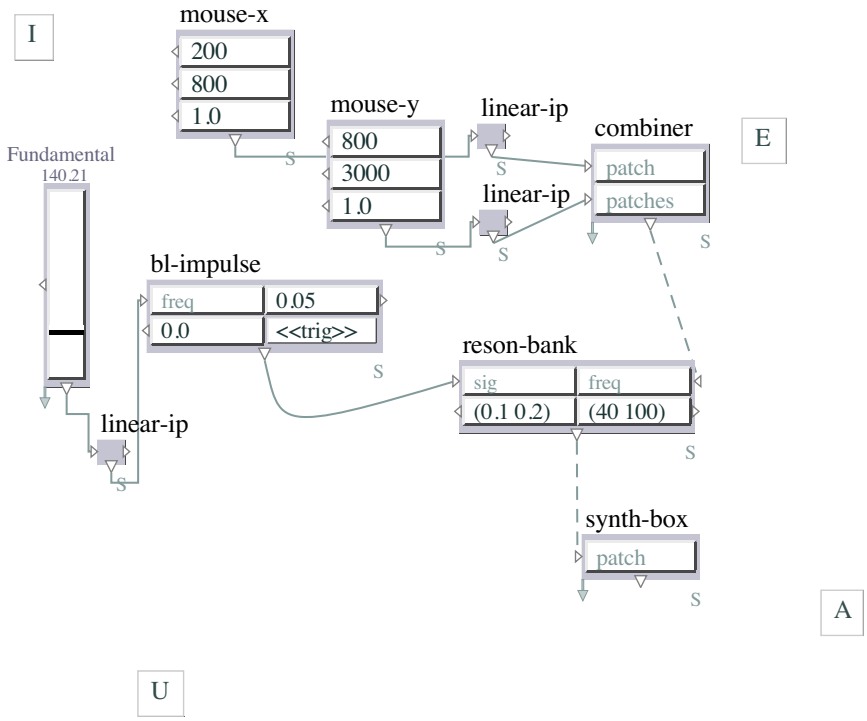

Figure 2.152: 04-formants

2.7.5.5 Granular

The 'granular-player' module plays a grain stream based on a loaded sample. 'range-start' and 'range-end' represent points in the loaded sample between which the start of each grain is randomly chosen. Value 0 represents the beginning of the sample waveform and value 1 represents the end.
In this example, the range length is zero, thus exactly specifying the grain location inside the waveform. This location is driven by the 'line-trigger' box which, when triggered, emits a continuous signal ranging from 0 to 1 in a specified time interval.
In this application, granular synthesis is used to perform a rudimentary time stretch and pitch shift.

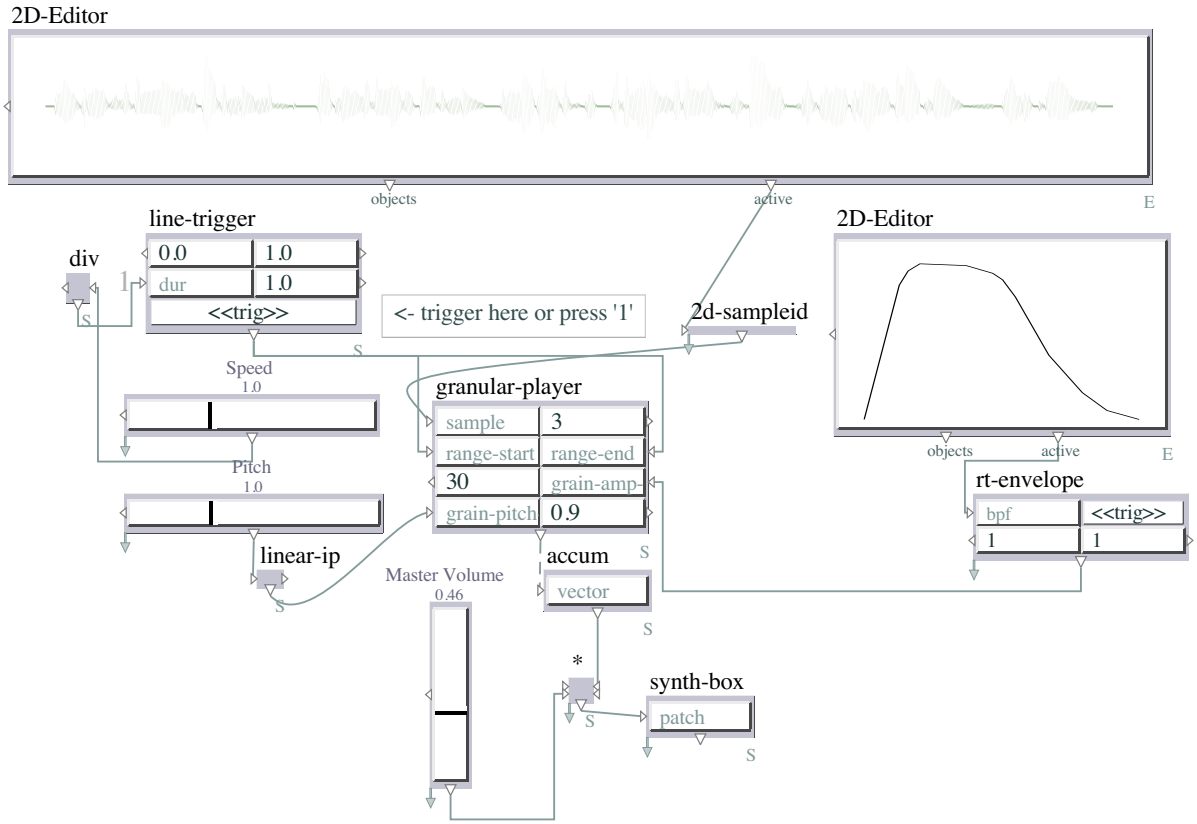

Figure 2.153: 05-granular

2.7.6 MIDI

2.7.6.1 MIDI-Membrane

This is a basic test patch that demonstrates two modules that are related to MIDI control: 'midi-trigger' and 'midi-cc'.
'midi-trigger' sends a trigger event and produces a scaled impulse upon receiving MIDI data within specified note number and velocity limits.
'midi-cc' (Control Change) follows a MIDI controller according to the desired MIDI channel and control change number and outputs a signal that is bound by the 'min' and 'max' inputs. The 'curve' input defines the response curve. Note that the MIDI pitch bend controller is represented here with a controller number 129.

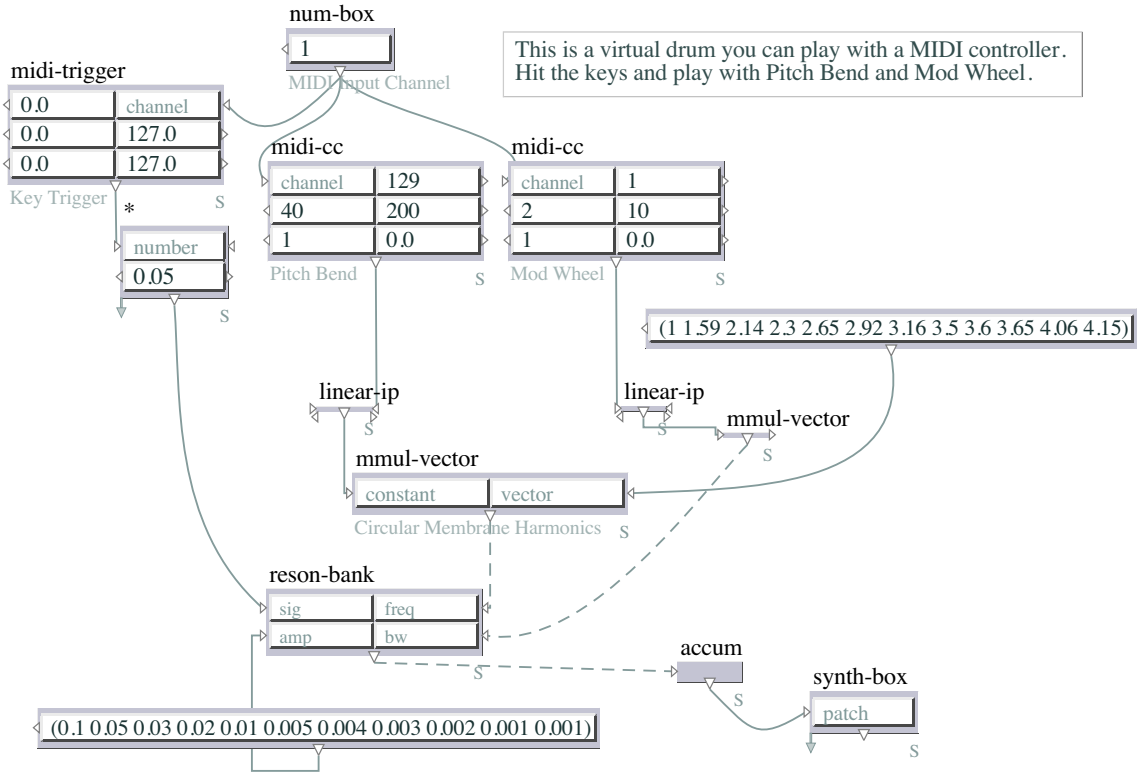

Figure 2.154: 01-midi-membrane

2.7.7 Compiler

2.7.7.1 Stereo-Bell

Currently this example works only in OS X Also you must have the Xcode development environment installed in your system.
This is an advanced example that demonstrates how visual PWGL abstraction boxes can be compiled to C code in order to define new synthesis boxes. Here an abstraction (1) called 'stereo-bell', having 4 inputs, contains internally a synthesis patch. In order to compile the abstraction choose the 'compile synth patch' option from the box menu of the abstraction. A dialog will appear where the user can edit various information, such as box-name, documentation and menu-name. The compilation produces in the background a new bundle file, that can be found in the 'PWGL-user/PWGLSynth/osx/' folder. The contents of this folder will be loaded automatically when launching PWGL.

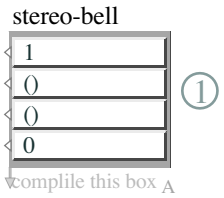

Figure 2.155: stereo-bell

2.7.8 RT-Sequences

2.7.8.1 Introduction

2.7.8.1.1 RT-Sequences and Compositional Sketches

This section demonstrates how the code-box can be used to calculate synthesis control information directly using Lisp code. The idea is to generate algorithmically typically relatively short musical textures. The user can improvise with various compositional ideas, adjust parameters, and listen to the results in real-time either individually or interleaved. This is achieved by utilizing a special code-box scheme that allows any textual Lisp expression to be interfaced to the visual part of the PWGL system. This scheme can also be used for score-based synthesis control.

2.7.8.1.2 With-Synth Macros

All code examples are wrapped inside the 'with-synth' and 'with-synth-instrument' macros that will send any synth events to the running synthesis patch. The 'with-synth-instrument' macro takes three arguments: (1) 'instrument', a pointer to a visual

abstraction box containing the instrument definition (2) 'max-poly', maximum number of voices required by the texture realization (3) 'body', the actual lisp code containing synth-events and synth-triggers.
A typical example using these macros would look like this:

```
(with-synth
   (with-synth-instrument ins 1
     <insert your code here>))
```

You can use several instruments inside the main 'with-synth' macro):

```
(with-synth
   (with-synth-instrument ins1 1
     <insert ins1 code here>)
   (with-synth-instrument ins2 1
     <insert ins2 code here>))
```

2.7.8.1.3 Synth-Events and Synth-Triggers

The synth events are created using the 'synth-event' and 'synth-trigger' methods.
'synth-event' has three required arguments: (1) 'time', a float, gives the delay before the event is sent; (2) 'name', a keyword, must match one of the 'synth-plug' names (the first input) of the running synth patch. (3) 'value', a float or a list of floats.
'synth-trigger', in turn, is used for simple trigger events, and it has two required arguments: (1) 'time' (2) 'name'
For example, we could trigger immediately a resonator (see the first example patch) using the following code:

```
(with-synth
  (with-synth-instrument ins 1
      (synth-trigger 0 :trig)))
```

Or, we could trigger the resonator after 2s:

```
(with-synth
  (with-synth-instrument ins 1
      (synth-trigger 2 :trig)))
```

We could also change the ':freq' input of the resonator:

```
(with-synth
  (with-synth-instrument ins 1
      (synth-event 0 :freq 200)))
```

2.7.8.1.4

Optional keyword arguments can be given changing the behavior of the methods. The ':id' keyword is used to automatically rename the 'name' parameter. This feature is used in conjunction with polyphonic instrument definitions. In the first patch all cases are strictly monophonic thus we are not using the ':id' keyword there. However, the

second patch uses the ':id' keyword in order to distinguish different box instances in a polyphonic situation. Furthermore an optional type specifier, ':type', can be given if the event is not a normal one. This can be used for instance to send MIDI events instead of ordinary synth events (the midi event list here consists of port, status, key, and velocity):

```
(with-synth
  (with-synth-instrument ins 1
;; send a midi note-on event (144), port 0, key 60, vel 100
      (synth-event 0 () '(0 144 60 100) :type :midi)))
```

2.7.8.1.5 Triggering RT-Sequences

The 'synth-trigger' method can also be used to trigger code-boxes that define RT-sequences. This can be done with the help of the trigger-string of the code-box. This string will be used to find the correct code-box from the current patch. Thus the following expression will trigger the code-box having the trigger-string "2" at 1.0s (note that in the previous synth-trigger examples the second argument was always a keyword and not a string):

```
(synth-trigger 1.0 "2")
```

The 'synth-trigger' accepts a keyword argument called :input-values. This allows one to override the named input values of a code-box (these are normally given at the patch level) without modifying the graphical representation of the patch. Each input-value is a list forming a name/value pair. Thus the following code will trigger the code-box having the trigger-string "1" at 0s, and the input values for the "low" input will be 60 and the value for "high" will be 100:

```
(synth-trigger 0 "1" :input-values '(("low" 60)("high" 100)))
```

2.7.9 RT-Seq1

This patch contains an abstraction box (1), called 'Reson-patch', that defines a resonator that can be controlled using the entry-points that are defined by the four 'synth-plug' boxes. The 'synth-plug' boxes are labelled with keywords (e.g. :pan, :freq, :amp, :trig) that are used by the synth-event and synth-trigger methods inside the five code-boxes.
Instrument inputs of the code-boxes must be connected to a pointer to an instrument definition patch, thus we need a special box, called 'pwgl-box-ptr', between the abstraction and the instrument inputs of the code-boxes.
All five code-boxes use here the same instrument definition. The code-boxes in turn define simple texture types. These textures can be modified either by editing the top-level inputs of a code-box, or by changing the actual code that is situated inside the code-box.
First start the synth (1). While the synth is running you can trigger any sequence (2-6) by evaluating one of the code-boxes. The five code-boxes have trigger-strings from 1 to 5, and thus they can be triggered with keyboard short-cuts using the keys from 1 to 5.

Finally the patch at (7) contains a special code-box that can trigger any of the normal code-boxes (2-6) found in the patch. Internally it uses the 'synth-trigger' method that can also be used to trigger other code-boxes containing RT-sequence information. For more details see the Introduction page of this tutorial section.

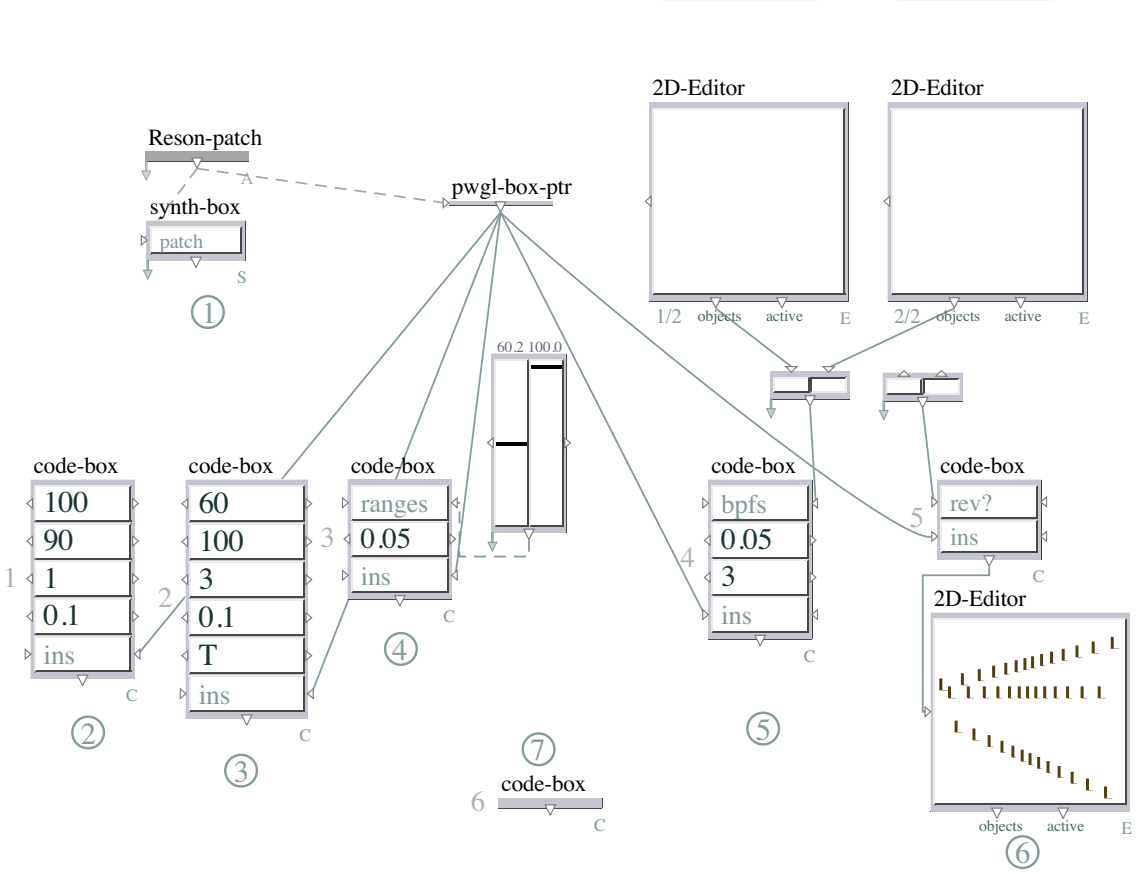

Figure 2.156: 01-rt-seq1

2.7.10 Poly-Seq

This patch defines a polyphonic instrument, called 'Reson-patch', and it utilizes the 'copy-synth-patch' box scheme to copy the synth patch 10 times (see the contents of the 'Reson-patch' abstraction).
In the code-box we use now the optional keyword argument ':id' in order to distinguish different box instances in this polyphonic situation. The current id number can be accessed automatically by calling the 'get-next-id' function.

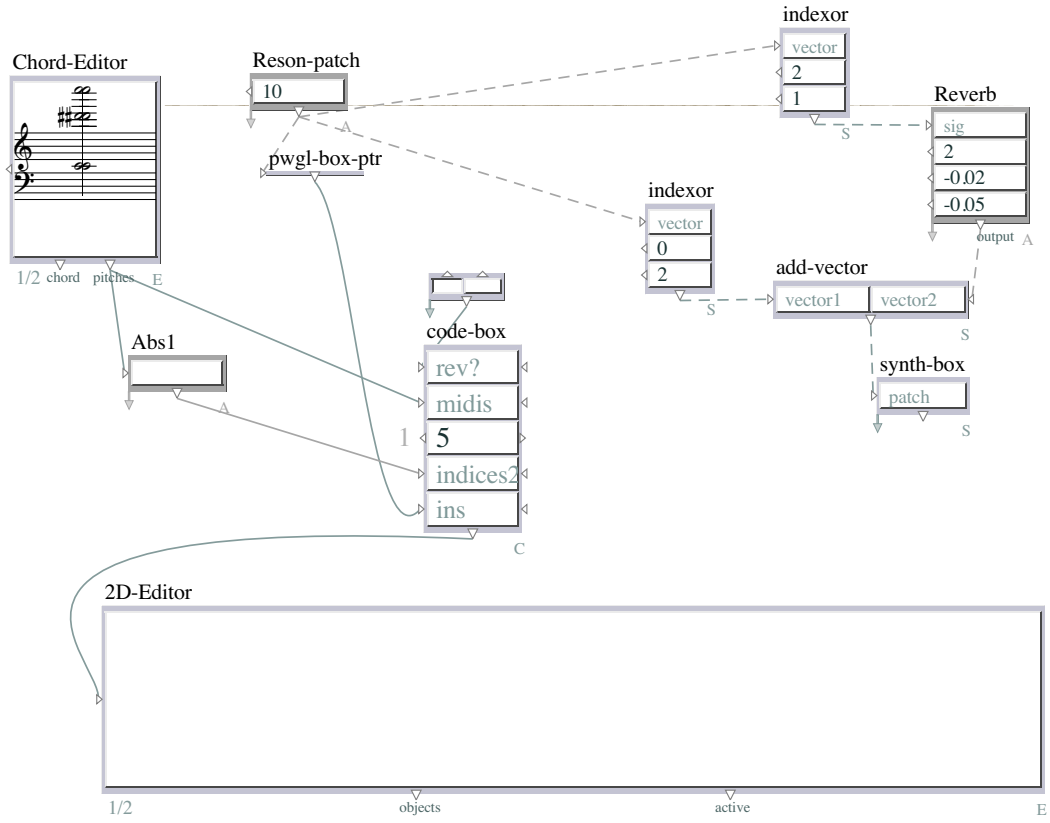

Figure 2.157: 02-poly-seq

2.7.11 Score1-Sine

This simple patch demonstrates how to control a sine module from a monophonic score. We have here three main entities: an instrument abstraction, a score and a code-box that translates the score information into synth events..
(1) The patch contains an instrument abstraction, 'Sine', which in turn contains a sine wave generator. A 'synth-plug' box controls the frequency of the 'sine' module. Note that the first input of the synth-plug' box is labelled as ':freq'.
In (2) we have a score, that contains a monophonic melodic line.
Finally, in (3), a code-box loops through the notes of the score. At each iteration the ':freq' parameter is updated.
To listen to the score first start the synth-box (4). After this you can trigger the code-box calculation by either evaluating it or by pressing '1' from the keyboard.

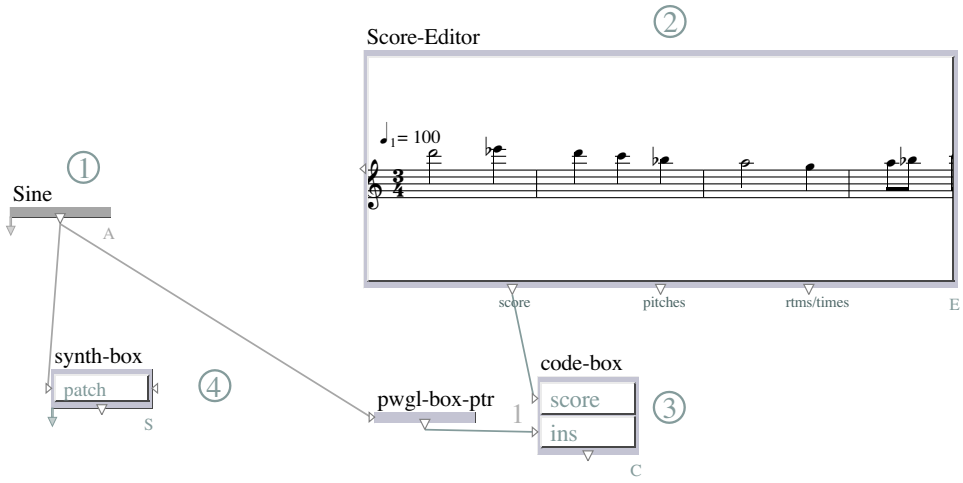

Figure 2.158: 10-score1-sine

2.7.12 Score2-Envelope

This patch is similar to the previous one except here we demonstrate how to control a sine module with an amplitude envelope. This patch can also handle polyphonic scores. The abstraction in (1) contains a sine wave generator with an amplitude envelope. This sub-patch is duplicated internally by a 'copy-synth-patch' box count times (count = 3) so we can realize polyphonic scores. We have here three 'synth-plug' boxes that allow us to control the frequency and the amplitude parameters of each individual 'sine' module. For more explanation see the 'Copy-synth-patch' tutorials.
In (2) a 'Score-Editor' box contains a sequence of three-voiced chords.
In (3) the code-box is used to convert the score information to synth-events. Now the code is somewhat more complex than in the previous case as we deal here with amplitude envelopes and polyphony. We loop through the notes and for each note we update the ':freq' and 'amp' parameters. After this we trigger the amplitude envelope using 'synth-trigger'. Each synth event uses now the ':id' keyword parameter in order to be able to control each 'sine' module individually.

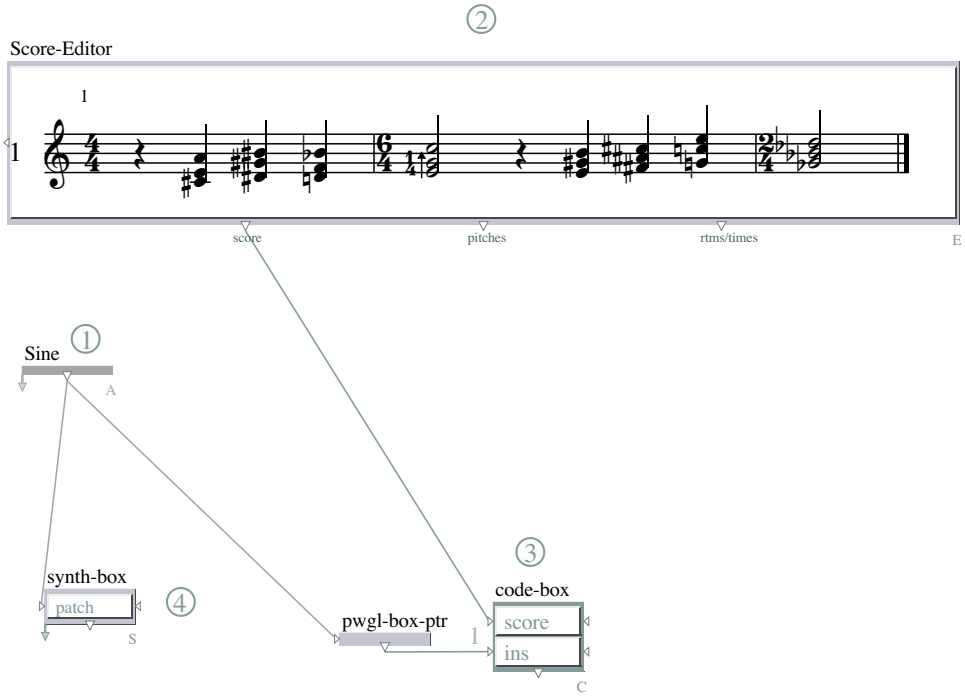

Figure 2.159: 11-score2-envelope

2.7.13 Score3-Expressions

This is a more advanced patch that shows how expression markings can be used to control a synthesis instrument.
The instrument abstraction (1) contains contains several 'synth-plug' boxes that both trigger and feed envelope information to the amplitude and frequency inputs of a sine-wave oscillator.
This patch contains two scores, (2) and (3), that contain different expression markings (staccato. slur, accent).
The code-box box (4) calculates amplitude envelope information according to the expression information found in the score. The code checks whether the current note belongs to a slurred group or not. In case of a slur the amplitude level does not drop to 0 thus producing a continuous sound during note transitions. Also the duration of a note is shortened if the note has a staccato expression. The 10-point frequency envelope contains a slight jitter around the fundamental frequency.

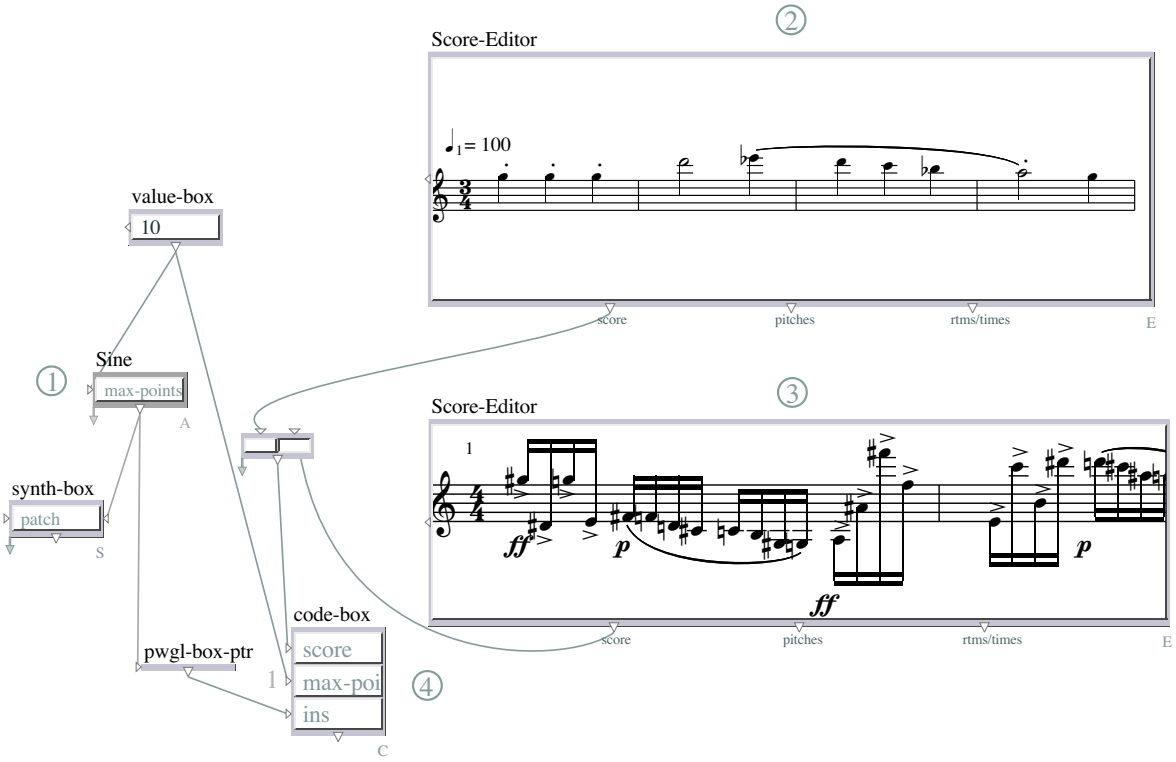

Figure 2.160: 12-score3-expressions

2.7.14 Score4-Vector

This patch demonstrates the use of vectored boxes (see the previous tutorial sections 'Vector' and 'Copy-synth-patch'). It shows how to control the amplitude and frequency envelopes of a 'sine-vector' module using a code-box.
The 'sines' abstraction (1) contains two vectored boxes ('envelope-trigger' and 'sine-vector'). We have also a 'stereo-pan' module which gets its pan position from the ':pan' synth-plug box.
The top-level patch (2) contains an 'accum-vector' box that mixes down the signals from the 'copy-synth-patch' box to a stereo signal. We have also a global reverb box (the reverb is as well a synthesis abstraction). The reverb output is mixed with the original dry signals by the 'add-vector' box.
The example contains two scores, (3) and (4). In (3) the pan is controlled by the midi-channel information of the notes. In (4), however, we have a break-point function expression (5) in the score, which allows the user to specify the pan parameter visually. Each note has 10 independent 10-point envelopes for amplitude and frequency. These envelopes are calculated by the code-box (6). Of special interest is also the panning of each note that is based either on midi-channel information, score (3), or on a panning

break-point function that is found in the score (4).

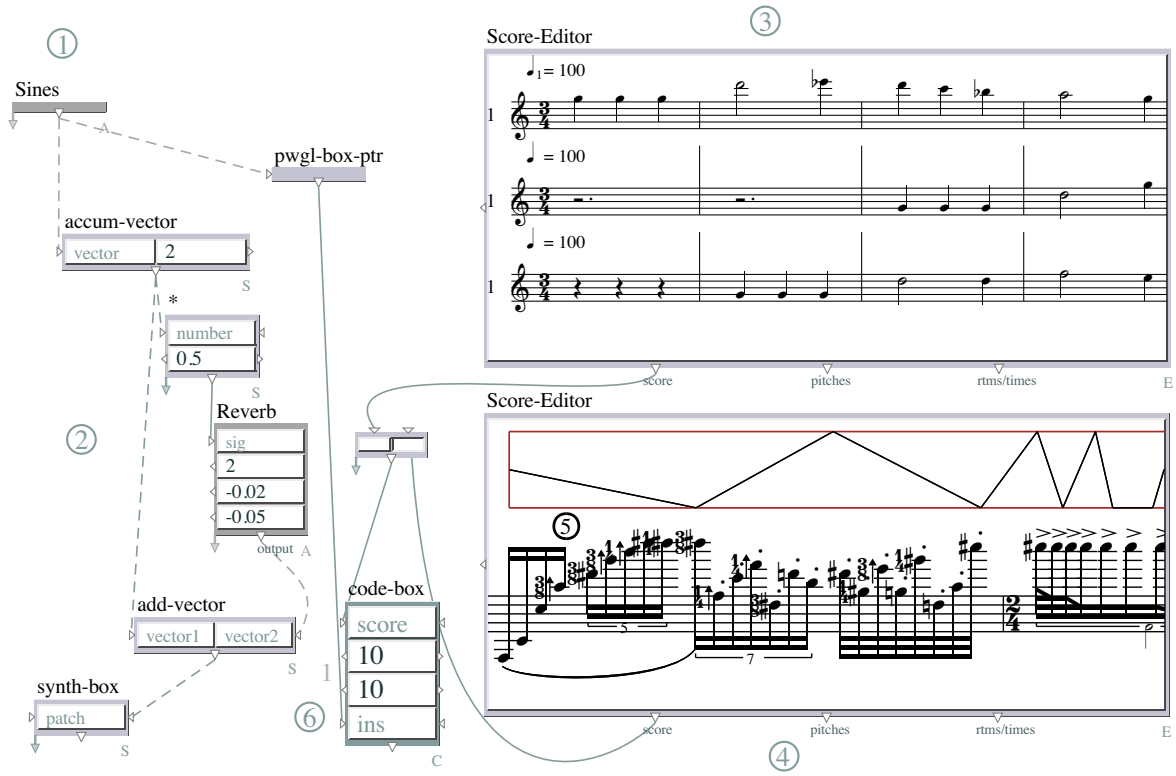

Figure 2.161: 13-score4-vector

Box Reference

*

arglist: `(&rest args)`
package: COMMON-LISP
menu: Synth▶math
no documentation

*+

arglist: `(mul1 mul2 add3)`
package: PWSYNTH
menu: Synth▶math
Calculates the sum of input 3 and the product of inputs 1 and 2.

+

arglist: `(&rest args)`
package: COMMON-LISP
menu: Synth▶math
no documentation

2d-sampleid

arglist: `(objects &optional index)`
package: SYSTEM
menu: Synth▶misc
returns 2D sampleID of a PWGL-sample-function 'objects'. 'objects' can be an atom or a list objects. In the latter case the the optional parameter 'index' (by default = 0) will be used to select the desired PWGL-sample-function object.

absolute

arglist: `(num1)`
package: PWSYNTH
menu: Synth▶math
Calculates the absolute value of the input.

accum

arglist: `(vector)`
package: PWSYNTH
menu: Synth▶vector
Accumulates all elements of vector and outputs a single element.

accum-vector

arglist: `(vector len)`
package: PWSYNTH
menu: Synth▶vector
No documentation available

add-vector

arglist: `(vector1 vector2)`
package: PWSYNTH
menu: Synth▶vector
Calculates the sum of two vectors.

all-subs

arglist: `(sc-name)`
package: SYSTEM
menu: PC-set-theory
returns all subset classes of the given SC (SC-name), accepts also a list of SCs in which case all subset classes of the given SCs are appended (all duplicates are removed from the result). The first input is a hierarchical menu-box, where the user selects the SC-name. When the input is scrolled, it displays all SC-names of a given cardinality. The cardinality can be changed by dragging the mini-scroll view in the right-most part of the input.

allpass

arglist: `(sig delay coef)`
package: PWSYNTH
menu: Synth▶filters
An allpass filter with a specified delay time (in seconds) and feedback coefficent.

allpass-vector

arglist: `(sig delay coef)`
package: PWSYNTH
menu: Synth▶filters
Vector of allpass filters with specified delay (in seconds) and feedback coefficents.

allpasscasc1

arglist: `(sig count coef)`
package: PWSYNTH
menu: Synth▶filters

First order allpass filter cascade. Count specifies the number of filters whose coefficents are given as a vector.

allpasscasc2

arglist: `(sig count a1 a2)`
package: PWSYNTH
menu: Synth▶filters
2nd order allpass filter cascade. Number of filters is specified by count and their coefficents are given as vectors a1 and a2.

append

arglist: `(&rest lists)`
package: COMMON-LISP
menu: Lisp
Construct a new list by concatenating the list arguments.

approx-midi

arglist: `(midis approx &optional ref-midi)`
package: PATCH-WORK
menu: Conversion
approx-m takes a midi value, midi ;and returns an approximation to the nearest division of the octave as defined by the user, approx. The value of resolution determines the resolution of approximation. An argument of 1, results in an output where all values are rounded to the nearest whole tone; 2, to the nearest semitone; 4, to the nearest quartertone; 4.5, to the nearest 4.5th of a tone, etc. When approx = 1, the optional argument ref-m in midi specifies the frequency resolution of the approximation. A value of 1.00 specifies semitone resolution, 0.50 specifies quartertone resolution, and so on.

arithm-ser

arglist: `(begin step end)`
package: PATCH-WORK
menu: Num series
Returns a list of numbers starting from begin to end with increment step. For example: (pw::arithm-ser 0 1 12) returns: PWGL->(0 1 2 3 4 5 6 7 8 9 10 11 12)

atan2

arglist: `(x y)`
package: PWSYNTH
menu: Synth▶math
Calculates the power of two inputs

band-filter

arglist: `(list val &optional pass?)`
package: PATCH-WORK
menu: List
<band-filter> passes or rejects all elements from <list> that fall inside a band of specified values of <val> . The range of values <val> is given either as a list of two numbers or as a list of lists of two numbers. Each pair of numbers define an interval. If <pass?> (the optional argument) is one (the default) only the element in list falling inside one of these intervals (of <val>) is selected. If <delete> is zero, elements in <list> not falling inside one of those intervals is selected. Intervals are defined by values inside list. For example, if <list> is (2 4 6 8 10 12 14 16 18 20) and <val> is ((1 3) (7 9)), <band-filter> returns (2 8), (the default is one). On the other hand (if the third input is open), if <list> is (2 4 6 8 10 12 14 16 18 20), <val> is ((1 3) (7 9)) and <pass?> is 0 (zero), <band-filter> returns (4 6 10 12 14 16 18 20). The argument list can be a list of lists. In this case the described behavior applies to each sublist.

biquad

arglist: `(signal a1 a2 b0 b1 b2)`
package: PWSYNTH
menu: Synth▶filters
Generid IIR biquad filter. Feedback coefficents are a1 and a2, while b0, b1 and b2 are feedforward coefficents.

bl-impulse

arglist: `(freq amp phase trig)`
package: PWSYNTH
menu: Synth▶osc
Band limited impulse oscillator suitable for audio rates. Freq specifies fundamental frequency, amp scales the amplitude, phase offsets the train within cycle (0-1) and <<trig>> retriggers the oscillator.

butlast

arglist: `(list &optional n)`
package: COMMON-LISP
menu: Lisp
Returns a new list the same as List without the N last elements.

cartesian

arglist: `(l1? l2? fun)`
package: PATCH-WORK
menu: Function

Applies the function fun to elements of l1? and l2? considered as matrices. Like g-oper ;fun may be a Lisp function (list, +, *, cons, etc.) or a function object created by the make-num-fun ;box . The result is a cartesian product of l1? by l2?.
(pw::cartesian 5 5 '+) will return PWGL->((10)) , (pw::cartesian '(1 2 3 4) '(5 6 7 8) '+) will return PWGL->((6 7 8 9) (7 8 9 10) (8 9 10 11) (9 10 11 12)) and (pw::cartesian '(1 2 3 4) '(5 6 7 8) 'list) will return PWGL-> (((1 5) (1 6) (1 7) (1 8)) ((2 5) (2 6) (2 7) (2 8)) ((3 5) (3 6) (3 7) (3 8)) ((4 5) (4 6) (4 7) (4 8)))

clipper

arglist: `(signal max min)`
package: PWSYNTH
menu: Synth▶waveshaping
Limits input signal between min and max by hard clipping.

code-box

arglist: `()`
package: SYSTEM
menu: Data
The code-box allows the user to express in textual form complex Lisp expressions and it is one of the most important tools (along with the Lisp-code-box and the text-box) to interface Lisp with the graphical part of our system.
The user can open a text-editor by double-clicking the box. In the text editor, while the user writes the code, the text is simultaneously analysed. This analysis consists of basic syntax checks and extraction of free variables and function names that result in a parameter list of the final Lisp expression. This scheme provides the main interface to PWGL and allows the user to access information from the visual part of the system. The appearance of the box is calculated automatically based on this analysis.
If a free variable name starts with a '!', then it will have a special status in the Lisp expression. This variable type, called 'multi-eval', will be re-evaluated each time it is encountered in the expression (this scheme will also apply every iteration step in a loop). This can be used to dynamically extract new values from PWGL boxes (a typical box example would be 'g-random').
By default the code-box is called 'code-box'. This name can be changed by the user. In order to distinguish this box from the ordinary ones, there is a label 'C' at the low-right corner of the box.

collect-enp-objects

arglist: `(object type &key no-rest-p no-tied-p only-selected-p)`
package: SYSTEM
menu: Editors
collect-enp-objects is used to collect enp-objects (i.e. notes, chords, beats, measures, voices, parts) from 'object'. The 'type' input determines the type of the collected enp-objects.

The result can be filtered using the keyword arguments 'no-rest-p', 'no-tied-p' and 'only-selected-p'.

comb-allpass

arglist: `(sig delay maxdelay coef)`
package: PWSYNTH
menu: Synth▶delay
A building block for reverberation algorithms. Features a delay line fed into a first order allpass filter. Delay is given in seconds and can't exceed the specified maximum. Coef specifies the allpass filter coefficent

comb-allpass-vector

arglist: `(sig delay maxdelay coef)`
package: PWSYNTH
menu: Synth▶filters
A vector of building block for reverberation algorithms. Features a delay line fed into a first order allpass filter. Delay is given in seconds and can't exceed the specified maximum. Coef specifies the allpass filter coefficent.

combiner

arglist: `(patch &rest patches)`
package: PWSYNTH
menu: Synth▶vector
Combines an arbitrary number of elements into a vector. It is possible to make multiple connections to the input.

cons

arglist: `(car cdr)`
package: COMMON-LISP
menu: Lisp
no documentation

const-value

arglist: `(patch &optional loopmode)`
package: SYSTEM
menu: Control
keeps the value coming from the 'patch' input constant even when the output of 'const-box' is connected to several other PWGL-boxes (i.e. 'const-box' is evaluated only once and after this the box returns the previously calculated value). By default 'const-box'

will re-evaluate at each top-level evaluation of a patch. When inserted in a loop, 'const-box' will behave depending on the optional input 'loopmode'. If 'loopmode' is: - ':once', (the default) the value is kept constant during looping - ':loopinit', 'const-box' will be re-evaluated at each loop start
and after this the value will be constant - ':eachtime', 'const-box' will be re-evaluated at each loop iteration
cycle.
See 'PWGL Help' for more details.

copy-patch-index

arglist: `(&optional list)`
package: SYSTEM
menu: Synth▶copy synth patch
returns the current copy-patch-index (an integer ranging from 0 - count-1, where count is given by the current copy-synth-patch box) when evaluating the patch. If the optional arg list is given then the nth element of the list arg is returned according to the current copy-patch-index

copy-synth-patch

arglist: `(count patch &optional name extra-vector)`
package: SYSTEM
menu: Synth▶copy synth patch
copies a sub-patch given in 'patch' 'count' times

create-list

arglist: `(count elem &optional list-fill)`
package: PATCH-WORK
menu: List
Returns a list of length <count> filled with repetitions of element <elem>

db->lin

arglist: `(dbs)`
package: PATCH-WORK
menu: Conversion
<dB->lin> takes a number <dbs> in decibels and converts it to linear. The input can be a list of numbers. In this case a list of linear values is returned.

dc

arglist: `(sig)`
package: PWSYNTH

menu: Synth▶filters
No documentation available

delay

arglist: `(sig time max-delay)`
package: PWSYNTH
menu: Synth▶delay
A simple delay without interpolation. Works well when no modulation of delay time or sub-sample accuracy is needed. Delay time is given in seconds.

delay-prime

arglist: `(sig delay)`
package: PWSYNTH
menu: Synth▶delay
A static delay line that can't be modulated. Delay time is rounded to the nearest prime number (in samples).

delay-prime-vector

arglist: `(sig delay)`
package: PWSYNTH
menu: Synth▶delay
Vector of static delay lines that can't be modulated. Delay time is rounded to the nearest prime number in samples.

detect-steps

arglist: `(sig treshold)`
package: PWSYNTH
menu: Synth▶control
When input signal 'sig' changes by more than 'treshold' during one sample frame, emits both a trigger and a refresh event. Signal is passed through unchanged.

differentiator

arglist: `(signal)`
package: PWSYNTH
menu: Synth▶filters
Differentiator: a high bias filter with 6dB/oct slope.

disk-writer

arglist: `(sig path)`
package: PWSYNTH
menu: Synth▶misc
disk-writer streams in real-time the audio input 'sig' to disk. The pathname of the resulting file can be given in the second input, if it is () then the default synth pathname will be used instead.

div

arglist: `(value1 value2)`
package: PWSYNTH
menu: Synth▶math
Calculates the quotinent of the two inputs

div-vector

arglist: `(vector1 vector2)`
package: PWSYNTH
menu: Synth▶vector
Calculates the quotinent of two vectors element by element.

duplicate-instance

arglist: `(object)`
package: SYSTEM
menu: Utilities▶misc
duplicates object

dx->x

arglist: `(start dxs)`
package: PATCH-WORK
menu: Num series
Constructs a list of numbers from <start> with the consecutives intervals of <dxs>. <dxs> can also be a list of lists of intervals. For example
(pw::dx->x 0 '(0 4 5 9 6 2 3 3)) will return PWGL->(0 0 4 9 18 24 26 29 32)
and (pw::dx->x 8 '(0 4 5 9 6 2 3 3)) will return PWGL->(8 8 12 17 26 32 34 37 40)

enp->synth

arglist: `(score rules &optional macro-notes out-format)`
package: SYSTEM
menu: Synth▶ENP interface
calculates a global ENP control list

enp-object-composer

arglist: `(type object/s)`
package: SYSTEM
menu: Editors
Converts any enp object or a list of enp objects into the object indicated by 'type'.

enp-score-notation

arglist: `(score incl/excl keywords)`
package: SYSTEM
menu: Editors
Collects ENP-score-notation list.

enp-script

arglist: `(score rules selection? &optional prepare-fns+args out-type)`
package: SYSTEM
menu: Constraints
'ENP-script' box can be used to produce various side-effects (such as adding expressions, analytical information, etc.) to an input-score. The position and type of the side-effects are defined by the 'rules' input. For more details see 'PWGL Help' and 'ENP Help'.
The 'selection?' input determines whether these side-effects are applied to the whole score or only to the selected areas in the input-score.
The 'prepare-fns+args' input can be used to customize the scripting process.
The 'out-type' input determines the result type of the 'ENP-script'. If it is equal ':string', then the output consists of all side-effects in textual form. If it is ':object', then the output consists ENP-objects that were created during the scripting process.
The side-effects can be undone using the 'enp script history...' option in the 'ENP-script' box popup-menu.

envelope-trigger

arglist: `(envelope trig)`
package: PWSYNTH
menu: Synth▶control
No documentation available

eval-box

arglist: `(value)`
package: SYSTEM
menu: Data
returns (eval <value>)

eval-when-load

arglist: `(patch &rest patches)`
package: SYSTEM
menu: Data
evaluate inputs when loading a patch

exp-ip

arglist: `(sig lag)`
package: PWSYNTH
menu: Synth▶control
Smooths input signal with exponential interpolation. Can't interpolate between positive and negative values or zero and nonzero. Response lag is given in seconds.

expand-lst

arglist: `(list)`
package: PATCH-WORK
menu: List
Expands a list by one (or both) of the following:
1. Repeating each item number times following a pattern
of the form: number*
2. Creating a sequence of numbers going from n to m by steps of k, indicated by the pattern n-m s k. A step of 1 can be omitted.
For example the list (3* (2 4) 0_8), returns
(2 4 2 4 2 4 0 1 2 3 4 5 6 7 8),
and the list (2* (a z 2*(4 12) (1_5)) 0_16s2) returns
(a z 4 12 4 12 (1 2 3 4 5) a z 4 12 4 12 (1 2 3 4 5) 0 2 4 6 8 10 12 14 16).

f->m

arglist: `(freqs &optional approx ref-midi)`
package: PATCH-WORK
menu: Conversion
Converts frequency ;to midi. It takes a frequency (Hz) or list of frequencies and returns corresponding midi values. The optional approx argument lets one limit returned values to a given approximation (see approx- m). When approx = 1, the optional argument ref-m in midi specifies the frequency resolution of the approximation. A value of 1.00 specifies semitone ;;; resolution, 0.50 specifies quartertone ;resolution, and so on.

fft

arglist: `(sig frame-size step window zero-pad)`
package: PWSYNTH
menu: Synth▶analysis

No documentation available

fft-partials

arglist: `(fft)`
package: PWSYNTH
menu: Synth▶analysis
No documentation available

fibo-ser

arglist: `(seed1 seed2 limit &optional begin end)`
package: PATCH-WORK
menu: Num series
Returns a list of numbers in the Fibonacci series ;where the first element is seed and the additive factor is seed2. The limit parameter is the limit of this list. It is also possible to specify two parameters begin and end which delimit the calculation of the series. For example:
(pw::fibo-ser 0 1 337) returns
PWGL->(0 1 2 3 5 8 13 21 34 55 89 144 233),
(pw::fibo-ser 0 4 337) returns PWGL->(0 4 8 12 20 32 52 84 136 220) and
(pw::fibo-ser 0 4 337 3 6) returns PWGL->(12 20 32 52)

first

arglist: `(list)`
package: COMMON-LISP
menu: Lisp
no documentation

flat

arglist: `(lst)`
package: PATCH-WORK
menu: List
Takes off every parenthesis. There should be no dotted pair.

flat-low

arglist: `(list)`
package: PATCH-WORK
menu: List
Flattens lowest level sublists. Ex: '(((1 2 3) (4 5 6)) ((7 8 9) (10 11 12))) becomes: ((1 2 3 4 5 6) (7 8 9 10 11 12))

flat-once

arglist: `(list)`
package: PATCH-WORK
menu: List
flattens the first level of a list of lists.Ex: '(((1 2 3) (4 5 6)) ((7 8 9) (10 11 12))) becomes: ((1 2 3) (4 5 6) (7 8 9) (10 11 12))

funcall

arglist: `(function &rest arguments)`
package: COMMON-LISP
menu: Lisp
Calls Function with the given Arguments.

g*

arglist: `(l1? l2?)`
package: SYSTEM
menu: Arithmetic
PWGL version of *. Both arguments can be numbers/bpfs or lists of any depth

g+

arglist: `(l1? l2?)`
package: SYSTEM
menu: Arithmetic
PWGL version of +. Both arguments can be numbers/bpfs or lists of any depth

g-

arglist: `(l1? l2?)`
package: SYSTEM
menu: Arithmetic
PWGL version of -. Both arguments can be numbers/bpfs or lists of any depth

g-abs

arglist: `(l?)`
package: SYSTEM
menu: Arithmetic
PWGL version of abs. The argument can be number/bpf or list of any depth

g-alea

arglist: `(list percent)`
package: PATCH-WORK
menu: Num series
Add a uniform random function to the list <list> of some depth according to a percentage <percent> indicated.

g-average

arglist: `(xs weights?)`
package: PATCH-WORK
menu: Arithmetic
average value of <xs>, weighted by linear <weights> or 1. <xs> and <weights> may be trees. Trees must be well-formed. That is, the children of a node must be either all leaves or all nonleaves.

g-ceiling

arglist: `(l?)`
package: SYSTEM
menu: Arithmetic
PWGL version of g-ceiling. The argument can be number/bpf or list of any depth

g-div

arglist: `(l1? l2?)`
package: SYSTEM
menu: Arithmetic
Integer division of two numbers or trees, Euclidean division. Both arguments can be numbers/bpfs or lists of any depth

g-exp

arglist: `(l?)`
package: SYSTEM
menu: Arithmetic
PWGL version of exp. The argument can be number/bpf or list of any depth

g-floor

arglist: `(l?)`
package: SYSTEM
menu: Arithmetic
PWGL version of floor. The argument can be number/bpf or list of any depth

g-log

arglist: `(l?)`
package: SYSTEM
menu: Arithmetic
PWGL version of log. The argument can be number/bpf or list of any depth

g-max

arglist: `(list)`
package: SYSTEM
menu: Arithmetic
PWGL version of max. The argument can be number or list of any depth

g-min

arglist: `(list)`
package: SYSTEM
menu: Arithmetic
PWGL version of min. The argument can be number or list of any depth

g-mod

arglist: `(l1? mod)`
package: SYSTEM
menu: Arithmetic
PWGL version of mod. Both arguments can be numbers/bpfs or lists of any depth

g-oper

arglist: `(fun obj1? &optional obj2?)`
package: PATCH-WORK
menu: Function
Applies fun to leaves of trees of obj1? and (optionally) obj2?. fun may be a Lisp function (list, +, *, cons, etc.) or a function object created by the make-num-fun box.
For example: (pw::g-oper '+ 4 5) will return PWGL->9 , (pw::g-oper 'list 4 5) will return PWGL->(4 5) ,
(pw::g-oper '+ '(1 2) '(3 4)) will return PWGL->(4 6) and
(pw::g-oper (pw::make-num-fun '(f(x y)= (+ (* x 2) (* y 3)))) '(1 2) '(3 4)) will return ? PW->(11 16)

g-power

arglist: `(l1? power)`
package: SYSTEM
menu: Arithmetic

PWGL version of expt. Both arguments can be numbers/bpfs or lists of any depth

g-random

arglist: `(low high)`
package: SYSTEM
menu: Arithmetic
Returns a random value between val1 and val2 inclusive. Both arguments can be numbers or lists of any depth

g-round

arglist: `(l1? &optional decimals l2?)`
package: PATCH-WORK
menu: Arithmetic
Rounds a number or tree. This module allows many operations, since it is extendible. The input decimals sets the choice of number of decimal places to round to. I2? specifies division before rounding.

g-scale%

arglist: `(l1? l2?)`
package: SYSTEM
menu: Arithmetic
Divides by 100 the product of <l1?> and <l2?>. Both arguments can be numbers or lists of any depth

g-scaling

arglist: `(vals? minout maxout &optional minin maxin)`
package: PATCH-WORK
menu: Num series
Replaces all the <vals?> considered between the minimum value of the list and the maximum value of the list, by the values proportionally placed between <minout> and <maxout>. If the list in question is a part of a larger list, or <vals?> is a variable that takes a value within a known interval, one can specify the minimum and maximum values.

g-scaling/max

arglist: `(list max)`
package: PATCH-WORK
menu: Num series
scales <list> (may be tree) so that its max becomes <max>. Trees must be well-formed: The children of a node must be either all leaves or all nonleaves.

g-scaling/sum

arglist: `(list sum)`
package: PATCH-WORK
menu: Num series
scales <list> (may be tree) so that its sum becomes <sum>. Trees must be well-formed. The children of a node must be either all leaves or all nonleaves.

g/

arglist: `(l1? l2?)`
package: SYSTEM
menu: Arithmetic
PWGL version of /. Both arguments can be numbers/bpfs or lists of any depth

geometric-ser

arglist: `(seed factor limit &optional begin end)`
package: PATCH-WORK
menu: Num series
The geometric-ser module returns a geometric series ;of numbers in which the first element is seed and the multiplicative coefficient is factor. The limit parameter is the limit of this list. It is also possible to specify two parameters begin and end which delimit the calculation of the series. For example:
(pw::geometric-ser 10 2 2000) will return PWGL->(10 20 40 80 160 320 640 1280) and if one sets begin to 2 and end to 5
(pw::geometric-ser 10 2 2000 2 5) one obtains: PWGL->(40 80 160 320)

gm-instrument

arglist: `(chan ins)`
package: SYSTEM
menu: Utilities▶MIDI
set GM instrument on channel(s), chan can be a number or a list.

granular-player

arglist: `(sample voices range-start range-end freq grain-amp-env grain-pitch-env grain-dur)`
package: PWSYNTH
menu: Synth▶samplers
Plays a grain stream based on a loaded sample. Chooses a grain out of the given sample from between range-start and range-end, which are normalized to 0..1. You can assign amplitude and pitch envelopes to individual grains. In these envelopes, envelope time is normalized to 0..1 for the grain duration.

group-lst

arglist: `(list group-lens)`
package: SYSTEM
menu: List
groups list into subsequnces, where group-lens indicates the length of each sublist. group-lens can be a number or a list of numbers. If list is not exhausted by group-lens, the last value of group-lens will be used as a constant until list has been exhausted.

hps

arglist: `(partials min-freq max-freq num-partials subtone num-products miss-coef)`
package: PWSYNTH
menu: Synth▶analysis
No documentation available

impulse

arglist: `(freq amp)`
package: PWSYNTH
menu: Synth▶osc
An oscillator that emits an impulse train. Frequency is freq rounded to the nearest integer period in samples. Amp is the amplitude of the impulse.

impulse-trigger

arglist: `(amp trig)`
package: PWSYNTH
menu: Synth▶control
Emits an unit impulse when triggered. The impulse amplitude is specified by the amp parameter.

impulse-vector

arglist: `(freq amp)`
package: PWSYNTH
menu: Synth▶osc
A vector of oscillators that emit an impulse train. Frequency is freq rounded to the nearest integer period in samples. Amp is the amplitude of the impulse.

included?

arglist: `(lst1 lst2 &optional test)`
package: PATCH-WORK
menu: Sets▶Combinations

This box compares two lists, returning true if all the elements in the first are also elements of the second. If the optional <test> argument is added, it is used as a predicate to detect equality between elements. Default value for <test> is the function 'equal.

indexor

arglist: `(vector index len)`
package: PWSYNTH
menu: Synth▶vector
Accesses a subvector of the input, staring at element number specified by index for a number of elements specified by len.

integrator

arglist: `(signal leak)`
package: PWSYNTH
menu: Synth▶filters
The box implements a leaky integrator. With leak value 0 the integrator doesn't leak, with value 1 it doesn't retain anything.

interpolation

arglist: `(begin end samples curves &optional format)`
package: PATCH-WORK
menu: Num series
Interpolates two lists of the same length. (If the lists are not the same length, the operation produces only the number of terms equal to the shorter list.) begin and end, in samples steps (i.e., samples is the number of steps). curve is an optional value that selects the type of interpolation:
1 = straight line,
< 1 = convex
> 1 = concave
If format is ':incl' the two extremes are included in the output. If format is ':excl' they are excluded.

inverse

arglist: `(xmin xmax value fun)`
package: PATCH-WORK
menu: Function
binary searches x in the interval [xmin,xmax] , such that fun(x)=value. fun must be either increasing or decreasing in the interval

k-rate

arglist: `(sig rate)`
package: PWSYNTH
menu: Synth▶control
Using k-rate, you can locally override the global control rate of the system. Connect the control signal into sig-input and use rate-input to specify the control rate for that input. The box you connect krate to will perform its control rate actions with the rate specified here.

lagrange-fun

arglist: `(l-x-y)`
package: PATCH-WORK
menu: Function
Retourne un polynome de Lagrange dfini par les points de liste <l-x-y>.

last-elem

arglist: `(list)`
package: PATCH-WORK
menu: List
returns the last element of <list>

length

arglist: `(sequence)`
package: COMMON-LISP
menu: Lisp
Returns an integer that is the length of SEQUENCE.

lin->db

arglist: `(amps)`
package: PATCH-WORK
menu: Conversion
<lin->db> takes a number <amps> and returns the corresponding value expressed in decibels. The input can be a list of numbers. In this case a list of db values is returned.

line

arglist: `(start end dur curve-expt rptme)`
package: PWSYNTH
menu: Synth▶osc

A line oscillator that produces an envelope segment from 'start' to 'end' in 'dur' seconds. The curve is a line segment raised to the power if 'curve-expt', with value 1 corresponding to linear segments, 2 corresponding to parabolic segments etc.

line-trigger

arglist: `(start end dur curve-expt trig)`
package: PWSYNTH
menu: Synth▶control
A line oscillator that interpolates between 'start' and 'end' in 'dur' seconds when triggered. Dur can be changed in real time, also during line segments. The curvature is defined by 'curve-expt'. Value of 1 corresponds to a linear segment, 2 to a parabolic segment and so on.

linear-fun

arglist: `(x0 y0 x1 y1 &optional print)`
package: PATCH-WORK
menu: Function
Calculate the parameters of the equation y = a x + b as a function of the two points (x0,y0) (x1,y1). The optional parameter print lets one print the function.

linear-ip

arglist: `(sig lag)`
package: PWSYNTH
menu: Synth▶control
Smooths input signal with linear interpolation. Response lag is given in seconds.

list

arglist: `(&rest args)`
package: COMMON-LISP
menu: Lisp
no documentation

list-explode

arglist: `(list nlists)`
package: PATCH-WORK
menu: List
list-explode divides a list into <nlist> sublists of consecutives elements. For example, if list is (1 2 3 4 5 6 7 8 9), and ncol is 2, the result is ((1 2 3 4 5) (6 7 8 9)), if list is (1 2 3 4 5 6 7 8 9), and ncol is 5, the result is: ((1 2) (3 4) (5 6) (7 8) (9)). If the number of divisions exceeds the number of elements in the list, the remaining divisions are returned as nil.

list-filter

arglist: `(test val list)`
package: PATCH-WORK
menu: List
list-filter removes elements from a <list> according to a predicate <test>. If the predicate is 'eq', all instances of <val> are removed from the list, regardless of their level. If, for example, the predicate is >, all elements of list which are greater than <val> are removed. Note that <val> can be a string, but only if the predicate <test> can handle a string. (list-filter '= 5 '(5 7 3 5 11 5 16 3 1 7 15 5 8 10 0 7 5 4 5 10)) will return PWGL->(7 3 11 16 3 1 7 15 8 10 0 7 4 10) , with test.

list-modulo

arglist: `(list ncol)`
package: PATCH-WORK
menu: List
<list-modulo> groups elements of a list that occur at regular intervals, and returns these groups as lists. <ncol> defines the interval between group members. For example, if we take the list (1 2 3 4 5 6 7 8 9) and give 2 for ncol, the result is ((1 3 5 7 9) (2 4 6 8)). In other words, every second element starting with the first, and then every second element starting with the second. If the number of <ncol> exceeds the number of elements in the list, the remaining lists are returned as nil. In effect, list-modulo divides <list> into <ncol> sublists containing elements modulo <ncol> according to their position in the list.

load-default-sample

arglist: `(2d-editor &optional path)`
package: SYSTEM
menu: Synth▶misc
Loads the default synth sample (given in the 'Synth Preferences') into 2D editor, If the optional argument 'path' is given then the sample is loaded from the path indicated by this input

m->f

arglist: `(midis)`
package: PATCH-WORK
menu: Conversion
Converts a midi pitches <midis> to frequencies (Hz).

madd-vector

arglist: `(mono vector)`
package: PWSYNTH

menu: Synth▶vector
Adds a mono signal to all vector elements.

make-num-fun

arglist: `(fexpr)`
package: PATCH-WORK
menu: Function
Creates a lisp function object from the ”functional” expr <fexpr> which is basically an infixed expression (see prefix-expr and prefix-help). When <fexpr> begins with something like (f(x)= ...), the formal arguments are taken from the given list, otherwise they are deduced from the body of <fexpr> and collected in the order they appear in it. Local variables are automatically handled. The resulting function is compiled when the value of *compile-num- lambda* is T (default). <make-num-fun> has the following syntax: the standard Lisp syntax, i.e., (f(x) = (- (* x x) x))). The variable name definition at the beginning of the function (f(x)= ...) is optional. If it is not included by the user, the program figures out which variables are involved.

mapcar

arglist: `(function list &rest more-lists)`
package: COMMON-LISP
menu: Lisp
Applies fn to successive elements of lists, returns list of results.

mat-trans

arglist: `(matrix)`
package: PATCH-WORK
menu: List
<mat-trans> transposes a matrix. That is, it interchanges rows and columns. Thus for example, (mat-trans ’((1 2) (5 7))) returns the list ((1 5) (2 7)), or if <matrix> is ((1 2) (3 4) (5 6) (7 8) (9 10)) <mat-trans> returns ((1 3 5 7 9) (2 4 6 8 10)). <mat-trans> behaves as if the sublists of matrix where arranged vertically. Then a sublist is constructed for each column resulting from this arrangement. the result is the list of all these sublists.

math-cos

arglist: `(phase1)`
package: PWSYNTH
menu: Synth▶math
Calculates the cosine of an angle. 2 Pi represents a period of this function.

math-sin

arglist: `(phase1)`
package: PWSYNTH
menu: Synth▶math
Calculates the sine of an angle. 2 Pi represents a period of this function.

matrix-constructor

arglist: `(labels data &optional key-data)`
package: SYSTEM
menu: Editors
construct out of labels and data a matrix-object that can be given to the first input of a matrix-editor box

mdiv-vector

arglist: `(constant vector)`
package: PWSYNTH
menu: Synth▶vector
Divides each vector element by the mono signal.

merge-partials

arglist: `(partials output-len)`
package: PWSYNTH
menu: Synth▶analysis
No documentation available

mf->chord-seq

arglist: `(&optional filename tempo-modif?  time-resol)`
package: SYSTEM
menu: Utilities▶MIDI
converts a midi file to a list of chord-sequences. 'filename' is the path of the midi file to be loaded. If 'filename' is equal to (), then an open-file dialog appears. If 'tempo-modif?' is T then the tempo function in the midi file track 0 is used to modify the start-times and durations of the resulting chords.

mf->score

arglist: `(&optional filename scopes time-sign tempo ignore-rests-p time-resol min-dur min-rest-dur)`
package: SYSTEM
menu: Utilities▶MIDI

converts a midi file to a list of parts. 'filename' is the path of the midi file to be loaded. If 'filename' is equal to (), then an open-file dialog appears. 'scopse' defines the complexity of the quantification of beats. This input can b a number (from 1 to 8), or a list of scope specs. The scope spec list consists of sublists that consist of: (1) a duration of the beat (1 = 1/4 note), (2) a list of allowed subdivisions, (3) a subdivision list of a beat into two halves. Thus a scope spec sublist: (1 (2 3 4) ()) allows subdivisions of a 1/4 note of 2,3 and 4, but disallows subdivision of the beat into two halves (i.e. (3) = ()).
'tempo' can be either a metronome value or a list of metronome values. In the first case the metronome value is used globally for all measures. In the latter case values are assigned to measures one by one. If the tempi list is exhausted before the measures then the last metronome value is kept constant for the remaining measures. Note that this input has no effect on the quantizing and it is used only to set the metronome values of the measures of the result. If 'tempo' is = (), then the tempo of the midi file is used.
'ignore-rests-p' controls whether rests are used in the result: if () then all rests are quantized, if T then rests are ignored. The latter case can be useful to simplify the result of the quantizing process.

midi->notename

arglist: `(midis)`
package: PATCH-WORK
menu: Conversion
midi->notename takes a midi value <midis> or list of midi values, and returns corresponding symbolic (ASCII) note names. Symbolic note names follow standard notation with middle c (midi 60) being C4. Semitones are labeled with a '#' or a 'b.' Quartertone flats are labeled with a '_', and quartertone sharps with a '+'. Thus, C4 a quartertone sharp (midi 60.50), would be labeled 'C+3'. Gradations smaller than a quartertone are expressed as the closest quartertone + or - the remaining cent value (i.e., midi 81.76 would be expressed as Bb5-24).

midi-cc

arglist: `(channel cc-num min max curve-expt port)`
package: PWSYNTH
menu: Synth▶input
MIDI Control Change follower. Choose desired MIDI channel and control change number to output signal bound by min and max, mapping the 7-bit MIDI value 0 to 'min' and 127 to 'max'. The values in between are on a continuous linear segment raised to the power of 'curve-expt' with 1 corresponding to linear response.

midi-controller

arglist: `(chan cc val)`
package: SYSTEM
menu: Utilities▶MIDI

set MIDI controller values on channel <chan>, chan and val can be numbers, chan and val can be lists, or chan can be a list and val a number

midi-trigger

arglist: `(port channel key-lo key-hi vel-lo vel-hi)`
package: PWSYNTH
menu: Synth▶input
MIDI key trigger. Sends a Trigger event and produces an impulse upon receiving MIDI data within specified note number and velocity limits.

mmul-vector

arglist: `(mono vector)`
package: PWSYNTH
menu: Synth▶vector
Multiplies each vector element with the mono signal.

mmuladd-vector

arglist: `(vector mul add)`
package: PWSYNTH
menu: Synth▶vector
Multiplies each vector element by the scale input, then adds the add input.

mod-delay

arglist: `(sig dtime &optional maxdtime)`
package: PWSYNTH
menu: Synth▶delay
Delay line with modulation and linear interpolation. Delay is given in seconds and can't exceed the specified maximum.

mod-delay-bank

arglist: `(sig time max-delay)`
package: PWSYNTH
menu: Synth▶delay
Bank of modulated delay lines with linear interpolation. Delay times are given as a vector whose width determines the number of delay lines in the system.

mod-delay-lg3

arglist: `(sig dtime &optional maxdtime)`
package: PWSYNTH
menu: Synth▶delay
An interpolating delay line with 3rd order lagrange interpolation. Offers improved sub-sample accuracy. Delay time is given in seconds, and can't exceed the specified maximum.

moog-ladder

arglist: `(sig freq q)`
package: PWSYNTH
menu: Synth▶filters
Implementation of a digital model of the classic Moog filter

mouse-x

arglist: `(min max curve-expt)`
package: PWSYNTH
menu: Synth▶input
Scales the current mouse x-position according to min-max and curve. Mouse cursor located at the left edge of the active window results in value 'min' wile the right edge corresponds to value 'max'. 'Curve' allows you to specify a polynomial response. 'Curve' 1 corresponds to a linear response, while 2 corresponds to a parabolic response.

mouse-y

arglist: `(min max curve-expt)`
package: PWSYNTH
menu: Synth▶input
Scales the current mouse y-position according to min-max and curve. Mouse cursor located at the top edge of the active window results in value 'min' wile the bottom edge corresponds to value 'max'. 'Curve' allows you to specify a polynomial response. 'Curve' 1 corresponds to a linear response, while 2 corresponds to a parabolic response.

msub-vector

arglist: `(mono vector)`
package: PWSYNTH
menu: Synth▶vector
Substracts the mono signal from all vector elements.

mtx-mul

arglist: `(input-vector matrix)`
package: PWSYNTH
menu: Synth▶vector
Multiplies input vector with another vector representing a n x n matrix

mul-vector

arglist: `(vector1 vector2)`
package: PWSYNTH
menu: Synth▶vector
Calculates the product of two vectors element by element.

multi-trigger

arglist: `(trigg)`
package: SYSTEM
menu: Synth▶misc
trigger all connected trigger inputs

num-box

arglist: `(num)`
package: SYSTEM
menu: Data
returns num; num must be a number

onepole

arglist: `(sig coef gain)`
package: PWSYNTH
menu: Synth▶filters
First order IIR filter component with unit delay feedback specified by coef and gain.

onepole-vector

arglist: `(sig coef gain)`
package: PWSYNTH
menu: Synth▶filters
Vector of first order IIR filter components with unit delay feedback specified by coef and gain.

ort16-mtx

arglist: `(vector)`
package: PWSYNTH
menu: Synth▶vector
Multiplies input vector by an 16x16 orthogonal matrix to create output vector.

ort8-mtx

arglist: `(vector)`
package: PWSYNTH
menu: Synth▶vector
Multiplies input vector by an 8x8 orthogonal matrix to create output vector.

pad-vector

arglist: `(vector length)`
package: PWSYNTH
menu: Synth▶vector
Truncates or zero-pads the incoming vector to match the specified number of elements.

parab-ip

arglist: `(sig lag)`
package: PWSYNTH
menu: Synth▶control
Smooths input signal with 2nd order polynomial interpolation. Response lag is given in seconds.

param-eq

arglist: `(sig freq gain q type)`
package: PWSYNTH
menu: Synth▶filters
A band of parametric EQ. Freq is the center frequency in Hertz, gain is a multiplier where applicable (0.5 is approximately -6dB) and Q specifies filter quality or sharpness. Various types of equalizer shapes can be accessed by the type parameter.

permut-circ

arglist: `(list &optional nth)`
package: PATCH-WORK
menu: Sets▶Combinations
Returns a circular permutation of a copy of <list> starting from its <nth> element, (<nth> is the argument of the second optional input) (which defaults to 1) , (<nth> = 0 means the first element of <list>, <nth> = 1 means the second element of <list>,

and so on) For example, if <list> is (1 2 3 4 5 6 7 8 9 10) <permut-circ> returns (2 3 4 5 6 7 8 9 10 1), (the default is one). On the other hand (if the second input is open, <nth>), if <list> is (1 2 3 4 5 6 7 8 9 10), and <nth> is 3 (zero) , <permut-circ> returns (4 5 6 7 8 9 10 1 2 3)..

permut-random

arglist: `(list)`
package: PATCH-WORK
menu: Sets▶Combinations
Returns a random permutation of list.

poly-shaper

arglist: `(signal coefs-pos coefs-neg)`
package: PWSYNTH
menu: Synth▶waveshaping
Shapes positive and negative parts of signal with two polynomials specified by vectors of coefficents. The polynomial coefficent vectors should be given in an ascending order.

posn-match

arglist: `(list l-nth)`
package: PATCH-WORK
menu: List
<posn-match> can be a number or a list of numbers. Returns a copy of <l- nth> where each number is replaced by the corresponding element in the list <l>. For example, if <list> is (a b c d) and <l-nth> is (2 (0) 1) the box returns (c (a) b), where the list returned has the same structure as <l-nth>

posn-order

arglist: `(list funct)`
package: PATCH-WORK
menu: Sets▶Combinations
This module returns a list of positions of <list> ordered according to <funct>

pow

arglist: `(mantissa exponent)`
package: PWSYNTH
menu: Synth▶math
Calculates the power of two inputs

power-fun

arglist: `(x0 y0 x1 y1 &optional x2 y2 print)`
package: PATCH-WORK
menu: Function
Calculate the parameters of the equation y = a xb + c or y = a xb as a function of the points (x0,y0) (x1,y1) and (optional) (x2,y2) and create the corresponding function, either y = axb (for two pairs of points) or y = a xb + c (for three pairs of points).

prime-factors

arglist: `(number)`
package: PATCH-WORK
menu: Num series
Returns the prime decomposition of <number> in the form (... (prime exponent) ...) Primes known to the system are the 1230 primes ranging from 1 to 9973. They are in the global variable *prime-numbers*. You can change This variable by typing (setf *prime-numbers* (epw::gen-prime <max>)) where <max is an upper bound.

prime-ser

arglist: `(max)`
package: PATCH-WORK
menu: Num series
Returns the set of prime-numbers ranging from 0 upto max

prime?

arglist: `(n)`
package: PATCH-WORK
menu: Num series
Tests if n is a prime number. n must be smaller than 99460729.

pwgl-and

arglist: `(arg &rest args)`
package: SYSTEM
menu: Control
similar to the lisp macro 'and'. Evaluates the inputs in order. If any eval returns nil, quit and return nil. Else, return the value of the last input.

pwgl-circ

arglist: `(reset evalreset clist &rest clists)`
package: SYSTEM
menu: Control

circulates the contents of 'clist' and the optional arguments 'clists'. The circular lists can either be reset (1) manually by clicking the 'reset' button or by (2) evaluating a patch. In the latter case the reset option is active only if the second menu input-box contains ':yes'. The box is always reset when loading a patch or when the box is edited either by adding or removing inputs or when editing the contents of one of the inputs. The circular list inputs can contain atoms and lists - such as (1 2 3), (2_10s2), (1 (1 2 3) 3), (a (2_7) b) or (a s d f g) - and/or lisp expressions that are evaluated each time the lists are recirculated. This box is still under development.

pwgl-cond

arglist: `(testfn input test1 val1 else)`
package: SYSTEM
menu: Control
PWGL-cond tests pairwise - using 'testfn' as a test function - the second input, 'input', with 'test1', 'test2', etc. (left column of input-boxes) until the test function succeeds. The corresponding value input-box, 'val1', 'val2, etc. (right column of input-boxes) is returned. If the test function fails then the 'else' input is returned.

pwgl-if

arglist: `(test patch1 patch2)`
package: SYSTEM
menu: Control
PWGL if, if 'test' returns T, then the box returns 'patch1', otherwise 'patch2'

pwgl-or

arglist: `(arg &rest args)`
package: SYSTEM
menu: Control
similar to the lisp macro 'or'. Evaluates the inputs in order. If any eval returns non-nil, quit and return that value. Else, return nil.

pwgl-pop-circ

arglist: `(circ-list)`
package: SYSTEM
menu: Control
circulates a circ-list object

pwgl-pprint

arglist: `(obj)`
package: SYSTEM

menu: Utilities▶misc
pretty-prints obj in the PWGL output browser

pwgl-print

arglist: `(obj)`
package: SYSTEM
menu: Utilities▶misc
prints obj in the PWGL output browser

pwgl-progn

arglist: `(patch &rest patches)`
package: SYSTEM
menu: Control
evaluates each input in turn and returns the value of the last evaluation

pwgl-repeat

arglist: `(count patch)`
package: SYSTEM
menu: Control
PWGL repeat loop that evaluates 'patch' input 'count' times. It collects the results and returns them as a list.

pwgl-sample

arglist: `(object no-of-points &optional include-x?)`
package: SYSTEM
menu: Editors
sample following 2D-objects: bpf, bezier, sound-sample and scrap collection. pwgl-sample first calculates an internal sampling interval according to the min and max x values and the 'no-of-points' argument of the 2D-object in question. After this a train of sampling pulses are generated and the respective y values are read at each pulse (x) value. Normally, if 'include-x?' is 'nil', the box returns y values; if it is 'T' then the box returns x-y value lists.
- bpf returns a list of y values
- bezier returns a list x-y values
- sound-sample returns a list of y values (an approximation of an envelope of the sound sample).
- scrap collection returns a list that contains sublists for each scrap-box (a polygon) contained in the scrap collection. Each scrap-box sublist, in turn, contains a list where the first item is the name string of the scrap-box, the second item is a list where the first item is a x value followed by a list of all y values of the polygon at that specific x value, for instance: (('S1' ((0.165 (0.451 0.678)) (0.248 (0.4439 0.672)) ..) ('S2' ((0.3319 (0.227 0.341)) (0.41491032 (0.227 0.34)) ....))

pwgl-value

arglist: `(value-key &key init write)`
package: SYSTEM
menu: Control
'pwgl-value' allows to use pseudo-local variables or functions in PWGL. The values are stored in a hash table and they can be accessed anywhere in a patch. Note that the hash table is cleared with every top-level patch evaluation.
The 'value-key' parameter is a keyword. If the 'init' or the 'write' input is not given then the value stored under 'value-key' is returned.
An initial value can be stored under 'value-key' using the optional argument 'init'. After this the value can be accessed by other 'pwgl-value' boxes or from textual code in a patch (for instance in scripting or constraints rules). If the initial value needs to be updated after the initialization then use the 'write' argument.
The 'init' or 'write' input can be any lisp value (i.e. number, list, etc.). or it can be also a lisp lambda expression (i.e. ' #'(lambda () (random 1.0))'), or a lisp closure (in this case use the expression '#.' before the expression to force evaluation, i.e.: ' #.(let ((x 0)) #'(lambda ()(prog1 x (incf x 2))))')
If 'value-key' is a list of keywords and 'init' or 'write' is a list of values then all keywords and values are stored pairwise.
If 'value-key' is a list of keywords then the stored values given by the 'value-key' list are returned as a list.

qt-controller

arglist: `(chan ctrl val)`
package: SYSTEM
menu: Utilities▶MIDI
only for mac, set QT controller values on channel <chan>, chan and val can be numbers, chan and val can be lists, or chan can be a list and val a number

rand1

arglist: `(amp1)`
package: PWSYNTH
menu: Synth▶osc
Outputs white noise ranging from -amp to +amp.

rand2

arglist: `(low high)`
package: PWSYNTH
menu: Synth▶osc
Outputs white noise between the specified low and high boundaries.

randh

arglist: `(low high freq)`
package: PWSYNTH
menu: Synth▶osc
A noise generator with output ranging from low to high combined with a sample and hold operation. Freq specifies how often the output changes. At these intervals, a noise generator is sampled for the next output value.

randi

arglist: `(low high freq)`
package: PWSYNTH
menu: Synth▶osc
A noise generator with output ranging from low to high combined with an interpolating sample and hold operation. Freq specifies how often the noise generator is sampled. The output consists of linear segments between these points.

randi-vector

arglist: `(low high freq)`
package: PWSYNTH
menu: Synth▶osc
A vector of noise generators with output ranging from low to high combined with an interpolating sample and hold operation. Freq specifies how often the noise generator is sampled. The output consists of linear segments between these points.

range-filter

arglist: `(list posn &optional delete)`
package: PATCH-WORK
menu: List
<range-filter> selects from a list <list> all elements falling inside a range of given positions <posn>. The range of positions <posn> is given either as a list of two numbers or as a list of lists of two numbers. Each pair of numbers define an interval. If <delete> (the optional argument) is zero (the default) any element in list falling inside one of these intervals is selected. If <delete> is one, elements in <list> not falling inside one of those intervals is selected. Intervals are defined by position inside list. For example, if <list> is (4 7 2 3 1 8 5) and <posn> is ((4 5) (0 2)), <range-filter> returns (4 7 2 1 8). On the other hand (if the third input is open), if <list> is (4 7 2 3 1 8 5), <posn> is ((4 5) (0 2)) and <delete> is 1, <range- filter> returns (3 5). The argument list can be a list of lists. In this case the described behaviour applies to each sublist.

read-mor-db

arglist: `(tr-factor max-cnt)`

package: SYSTEM
menu: Synth▶database
reads a database-file in Lisp (or 'LL') format and returns a list consisting of freqs, amps and bandwidths (= !/ring-times). tr-factor gives the multiplication factor of the freqs-list. max-cnt gives the length of the result lists. the database is sorted according to amplitude and the first max-cnt amplitudes are scaled so that their sum is equal to 1.0. The final result is sorted according to ascending freqs.

rem-dups

arglist: `(lst &optional test depth)`
package: PATCH-WORK
menu: List
<rem-dups> removes repetitions of elements in <lst>, according to <test> (if the second input is open by clicking on 'E'). <test> must be commutative. For example, the list (this this is my list list) returns (this is my list). Note that the last occurrence of a repeated element in a list is preserved; thus, the list: (1 2 3 1 4) returns (2 3 1 4). Returns a copy of <lst>.

remove

arglist: `(item sequence &key from-end test test-not start end count key)`
package: COMMON-LISP
menu: Lisp
Returns a copy of SEQUENCE with elements satisfying the test (default is EQL) with ITEM removed.

reson

arglist: `(sig freq amp bw)`
package: PWSYNTH
menu: Synth▶filters
A two pole digital resonator. Freq (frequency) and bw (3dB bandwidth) are given in Hertz. Output is scaled by the amp parameter.

reson-bank

arglist: `(sig freq amp bw)`
package: PWSYNTH
menu: Synth▶filters
A bank of two pole digital resonator. Freq (frequencies) and bw (3dB bandwidths) are given as vectors in Hertz. Output is scaled by the amp parameter. The number of resonators is specified by the width of parameter vectors.

reson-vector

arglist: `(sig freq amp bw)`
package: PWSYNTH
menu: Synth▶filters
A vector of two pole digital resonators. Freq (frequency) and bw (3dB bandwidth) are given in Hertz. Output is scaled by the amp parameter.

rest

arglist: `(list)`
package: COMMON-LISP
menu: Lisp
no documentation

reverb-block

arglist: `(signal delay max-delay filter-coef filter-gain allpass-coef allpass-delay allpass-maxdelay)`
package: PWSYNTH
menu: Synth▶delay
No documentation available

reverse

arglist: `(sequence)`
package: COMMON-LISP
menu: Lisp
Returns a new sequence containing the same elements but in reverse order.

ripple-delay-lg3

arglist: `(sig delay ripple ripdepth maxdelay)`
package: PWSYNTH
menu: Synth▶delay
An interpolated delay line with an additional ripple comb filter. Delay is given in seconds and can't exceed the specified maximum. Ripple specifies the second comb filter as a fraction of delay time, while rippledepth specifies its depth.

ripple-slide-delay-lg3

arglist: `(sig delay ripple ripdepth maxdelay)`
package: PWSYNTH
menu: Synth▶delay

An interpolated delay line with an additional ripple comb filter. Delay is given in seconds and can't exceed the specified maximum. Ripple specifies the second comb filter as a fraction of delay time, while rippledepth specifies its depth.

rt-envelope

arglist: `(bpf trigg x-scale y-scale &optional x-offset y-offset)`
package: SYSTEM
menu: Synth▶misc
RT-envelope

rule-filter

arglist: `(rules)`
package: SYSTEM
menu: Constraints
The rule-filter box is used to filter PMC rules. These rules must be in textual form and each rule should have a unique documentation string.
'rules' input is typically connected to a text-box that contains PMC rules. The output in turn is connected normally to a search/analysis/script box that accepts PMC rules as input.
The incoming rules can be filtered by double-clicking the rule-filter box. This action opens a dialog showing all the documentation strings of the incoming rules. Here the user can select any rule subset that will be output from the box.

sample-fun

arglist: `(fun xmin step xmax)`
package: PATCH-WORK
menu: Function
Returns the list of values of <fun> from <xmin> to <xmax> with <step>. For example: (pw::sample-fun 'sin 0 1 6.3) will return PWGL->(0.0 0.8414709848078965 0.9092974268256817 0.1411200080598672 -0.7568024953079282 -0.9589242746631385 -0.27941549819892586) and (pw::sample-fun (pw::make-num-fun '(f(x)= x + 1)) 0 1 10) will return PWGL->(1 2 3 4 5 6 7 8 9 10 11)

sample-info

arglist: `(sound-object property)`
package: SYSTEM
menu: Synth▶misc

sample-player

arglist: `(sample fscaler amp trig &optional offset)`
package: PWSYNTH
menu: Synth▶samplers
Plays the specified sample from the sample database when triggered. Playback speed is determined by the fscaler parameter, with value 0.5 resulting in half speed playback. Sample waveform amplitude is scaled by the amp parameter, and offset allows you to trigger the sample from a specified location in seconds.

sc+off

arglist: `(midis)`
package: SYSTEM
menu: PC-set-theory
returns a list containing the SC-name and the offset (i.e. the transposition relative to the prime form of the SC) of midis (a list of midi-values), midis can also be a list of lists of midis in which case SC+off returns the SCs with offsets for each midi-value sublist

sc-info

arglist: `(function sc-name)`
package: SYSTEM
menu: PC-set-theory
allows to access information of a given SC (second input, SC-name). The type of information is defined by the first input (function). This input is a menu-box and contains the following menu-items:
CARD returns the cardinality of SC PRIME returns the prime form of SC ICV returns the interval-class vector of SC MEMBER-SETS returns a list of the member-sets of SC
(i.e. all 12 transpositions of the prime form) COMPLEMENT-PCs returns a list of PCs not included in the prime form of SC
The second input is a hierarchical menu-box, where the user selects the SC name. When the input is scrolled, it displays all SC-names of a given cardinality. The cardinality can be changed by dragging the mini-scroll view in the right-most part of the input. The input accepts also a list of SC-names. In this case the SC-info box returns the requested information for all given SC-names.

sc-name

arglist: `(midis)`
package: SYSTEM
menu: PC-set-theory
returns the SC-name of midis (a list of midi-values), midis can also be a list of lists of midis in which case SC-name returns the SC-names for each midi-value sublist

score-pmc-search-space

arglist: `(prop vals)`
package: SYSTEM
menu: Constraints
returns a list of pmc-domain objects that contain internally an association list of property/value pairs. This box is used in conjunction with the 'multi-score-pmc' box, and its output is normally connected to the 'search-space' input of the 'multi-score-pmc' box.
The user can choose different search properties, thus allowing to specify searches with multiple parameters. These properties can be accessed during search in rules using the m-method with the keyword ':data-access'. After the search has been completed, the result score of the 'multi-score-pmc' box will be updated automatically.
The inputs are operated/expanded in input-pairs, where the first input is a menu-box that allows to choose a property, and the second input is used to give different value options for this property. If a value input is (), then the respective property is disregarded from the resulting search-space.
The following properties are supported directly:
:midi :rtm :fingering :enharmonic :vel :instrument :expression :notehead
User properties can be given by connecting a new keyword symbol to the 'prop' input. User properties will be stored in the 'plist' slot of a note.
The :midi, :fingering, and :enharmonic properties accept as value 'T'. In this case the respective value parameters are calculated automatically according to the score/instrument information of the input score of the of 'multi-score-pmc' box as follows:
:midi -> range of midis of the current instrument (:midi accepts
also directly a list of midi values) :fingering -> all possible fingerings of the current instrument
for all midi values (this property works only for string instruments) :enharmonic -> all possible enharmonics for all midi values
If :rtm = 'T', then the value list will be (:attack :rest :tie)
Other properties work as follows:
:vel accepts a list of velocity values :notehead accepts a list of ENP notehead keywords :expression accepts a list of ENP expression keywords

scs/card

arglist: `(card)`
package: SYSTEM
menu: PC-set-theory
returns all SCs of a given cardinality (card)

shuffle-vector

arglist: `(vector indices)`
package: PWSYNTH
menu: Synth▶vector

Shuffles vector elements by recombining according to indices input. For example, indices vector (2 1 0) reverses a three element vector.

sine

arglist: `(freq amp)`
package: PWSYNTH
menu: Synth▶osc
A sine oscillator. Outputs a sinusoid with frequency freq and ranging from -amp to +amp.

sine-vector

arglist: `(freq amp)`
package: PWSYNTH
menu: Synth▶osc
Vector of sine-oscillators. Frequency is given in Hertz and output ranges from -amp to +amp.

sort-list

arglist: `(lst &optional test key)`
package: PATCH-WORK
menu: Sets▶Combinations
This module sorts a list. By default, the order of the sort is ascending, but since the module is extensible, you can open a second entry <test> to set the choice of order. If <test> is '>' the order is ascending, '<' indicates descending, and '=' keeps the order the same. One can also open a third input <key> for a function. The function <key> evaluates each element of the list <lst> and the result is then sorted according to the parameter <test>. (sort-list '(3 13 15 17 9 10 16 3 6 7 1 12 6)) will return PWGL->(1 3 3 6 6 7 9 10 12 13 15 16 17),
(sort-list '(3 13 15 17 9 10 16 3 6 7 1 12 6) '>) will return PWGL->(17 16 15 13 12 10 9 7 6 6 3 3 1),
(sort-list '((13 13 4) (3 9 3) (16 16 1) (11 13 6)) '< 'first) will return PWGL->((3 9 3) (11 13 6) (13 13 4) (16 16 1)), and
(sort-list '((13 13 4) (3 9 3) (16 16 1) (11 13 6)) '< 'second) will return PWGL->((3 9 3) (13 13 4) (11 13 6) (16 16 1))

sound-in

arglist: `(first-channel num-channels)`
package: PWSYNTH
menu: Synth▶input
Outputs a vector of audio channels from the audio input device

stereo-pan

arglist: `(sig pan &optional pan-law-db)`
package: PWSYNTH
menu: Synth▶multichannel
Box that turns a monophonic signal into a stereo signal using panning. Pan-law adjusts the center image level. The range for 'pan' parameter is -1 for extreme left and +1 for extreme right. Pan-law corresponds to the center image level in decibels, ranging from -6 to -3.

streaming-player

arglist: `(sample fscaler amp trig)`
package: PWSYNTH
menu: Synth▶samplers
Works much like sample-player, but can stream partially loaded samples from the disk. The streaming player doesn't support sample loops or offsets. Please note that the performance requirements increase with 'fscaler'.

sub

arglist: `(value1 value2)`
package: PWSYNTH
menu: Synth▶math
Calculates the difference of the two inputs

sub-vector

arglist: `(vector1 vector2)`
package: PWSYNTH
menu: Synth▶vector
Calculates the difference of two vectors.

sub/supersets

arglist: `(sc-name card)`
package: SYSTEM
menu: PC-set-theory
returns all subset classes of SC (when card is less than the cardinality of SC) or superset classes (when card is greater than the cardinality of SC) of cardinality card.. The first input is a hierarchical menu-box, where the user selects the SC-name. When the input is scrolled, it displays all SC-names of a given cardinality. The cardinality can be changed by dragging the mini-scroll view in the right-most part of the input.

subseq

arglist: `(sequence start &optional end)`
package: COMMON-LISP
menu: Lisp
Returns a copy of a subsequence of SEQUENCE starting with element number START and continuing to the end of SEQUENCE or the optional END.

synth-box

arglist: `(patch &optional mode output time)`
package: SYSTEM
menu: Synth▶synth
select box and start/restart synth by pressing 'v'. To stop synth press 's'.

synth-connect

arglist: `(boxname)`
package: SYSTEM
menu: Synth▶copy synth patch
simulates a connection by using the boxname input to access the box that is expected to give its input to the synth-connect box

synth-ctrl-bpf

arglist: `(bpf &optional x-scale y-scale x-offset y-offset)`
package: SYSTEM
menu: Synth▶control interface
scales the incoming bpf and returns a bpf ctrl expression for a synth-plug box

synth-ctrl-if

arglist: `(if-exp patch1 patch2)`
package: SYSTEM
menu: Synth▶control interface
ctrl if

synth-ctrl-mapping

arglist: `(abst class &optional function)`
package: SYSTEM
menu: Synth▶control interface
creates control methods for synth

synth-ctrl-val

arglist: `(ctrl-proc time)`
package: SYSTEM
menu: Synth▶control interface
evaluates ctrl-process (ctrl-proc) at time, if time is a list time values then the ctrl-process is evaluated for each time value

synth-plug

arglist: `(keyword initval)`
package: SYSTEM
menu: Synth▶control interface
allows to define entry-points for symbolic parameters at the leaves of a synth patch

table-filter

arglist: `(test val list numcol)`
package: PATCH-WORK
menu: List
<list> is a list of sub-lists. <numcol> is an index into <list> that selects a sub-list. <test> is a test function that is tested against each element of that sublist and <val> (e.g. (<test> <val> <element>). Matching elements are deleted from that sublist, as well as elements at the same position in every other sub-list. Example: (table-filter '= 100 '((1 2 3) (4 100 6) (7 8 9)) 1) -> '((1 3) (4 6) (7 9))

time-box

arglist: `(patch)`
package: SYSTEM
menu: Utilities▶misc
Executes patch and prints the amount of time used in execution.

trigger

arglist: `(freq amp)`
package: PWSYNTH
menu: Synth▶osc
An oscillator that emits an impulse train. Frequency is freq rounded to the nearest integer period in samples. Amp is the amplitude of the impulse. Each impulse is accompanied with a Trigger event.

update-plug-value

arglist: `(keyword &optional val)`
package: SYSTEM
menu: Synth▶control interface
update-plug-value

update-slider-banks

arglist: `(index &rest slider)`
package: SYSTEM
menu: Synth▶misc
updates when evaluated all slider setups of all slider-banks connected from the second input onwards according to index. If 'index' is a list of lists: ((dtime1 index1)(dtime1 index2) ..) then the updates are scheduled until the list is exhausted

value-box

arglist: `(value)`
package: SYSTEM
menu: Data
returns value. value can be a number, a list, a list in expand-list notation, a symbol, a keyword, or a string

vbap2d

arglist: `(sig azim)`
package: PWSYNTH
menu: Synth▶multichannel
An implementation of 2D Vector Based Amplitude Panning (VBAP, by Ville Pulkki, Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology). Pan position is given in degrees ('azim') and a multichannel amplitude panned signal is generated based on the current vbap speaker configuration. This configuration is defined in a separate box called 'vbap2d-conf'. See the tutorial for more examples.

vbap2d-conf

arglist: `(spkrangles)`
package: SYSTEM
menu: Synth▶multichannel
configure VBAP2D speaker positions in degrees. A stereo setup (which is the default configuration) can be defined using the 'spkrangles' list (-45 45). In this case the left-most position is achieved with an azimuth value -45,the center position with 0, and the right-most position with 45. A typical quad configuration is for example (-35 35 145 -145).

vbap2d-dist

arglist: `(sig azim dist revsc)`
package: PWSYNTH
menu: Synth▶multichannel
An implementation of 2D Vector Based Amplitude Panning (VBAP, by Ville Pulkki, Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology). Pan position is given in degrees ('azim') and a multichannel amplitude panned signal is generated based on the current vbap speaker configuration. This configuration is defined in a separate box called 'vbap2d-conf'. See the tutorial for more examples.
This box has two extra inputs when compared to 'vbap2d' box: 'dist' that approximates the distance of the source, and 'revsc' that can be used to scale the additional reverb signal. 'vbap2d-dist' scales the output signal according to the 'dist' parameter, and it returns an extra channel that can be used as a mono reverberation output.

x->dx

arglist: `(xs)`
package: PATCH-WORK
menu: Num series
Returns the list of the intervals between the contiguous values of a list <xs>. <xs> can also be a list of lists of values. For example (pw::x->dx '(0 4 5 9 6 2 3 3)) will return PWGL->(4 1 4 -3 -4 1 0)

x-append

arglist: `(l1? l2? &rest lst?)`
package: PATCH-WORK
menu: List
appends lists or atoms together to form a new list. This box can be extended.

x-diff

arglist: `(l1? l2? &optional test key &rest list)`
package: PATCH-WORK
menu: Sets▶Combinations
This box compares l1? to l2? and then returns all elements present in l1? but not in l2?, as a list. If the optional <test> argument is added, it is used as a predicate to detect equality between elements. Default value for <test> is the function 'equal. If the optional <key> argument is added, it is used as an accessor (e.g. first, second etc.) into the elements of the lists, prior to executing the <test> function. Additional lists can be compared by extending the box. They have the same status as <l2?>

x-intersect

arglist: `(l1? l2? &optional test key &rest list)`

package: PATCH-WORK
menu: Sets▶Combinations
This box returns a list of elements which are common to both <l1?> and <l2?>.If the optional <test> argument is added, it is used as a predicate to detect equality between elements. Default value for <test> is the function 'equal. If the optional <key> argument is added, it is used as an accessor (e.g. first, second etc.) into the elements of the lists, prior to executing the <test> function. Additional lists can be compared by extending the box. Beware that this operation is not commutative. For example: (x-intersect '(1 2 4 5 4) '(2 4)) will return -> (2 4 4) (x-intersect '(2 4) (1 2 4 5 4)) will return -> (2 4)

x-union

arglist: `(l1? l2? &optional test key &rest list)`
package: PATCH-WORK
menu: Sets▶Combinations
This box merges 2 lists, <l1?> and <l2?>, into a single list, with no repetitions. If the optional <test> argument is added, it is used as a predicate to detect equality between elements and and avoid repetition. Default value for <test> is the function 'equal. If the optional <key> argument is added, it is used as an accessor (e.g. first, second etc.) into the elements of the lists, prior to executing the <test> function. Additional lists can be compared by extending the box.

x-xor

arglist: `(l1? l2? &optional test key &rest list)`
package: PATCH-WORK
menu: Sets▶Combinations
This box compares lists <l1?> and <l2?> for elements that are present in either one or the other list (but not in both), and then returns them in a list. If the optional <test> argument is added, it is used as a predicate to detect equality between elements. Default value for <test> is the function 'equal. If the optional <key> argument is added, it is used as an accessor (e.g. first, second etc.) into the elements of the lists, prior to executing the <test> function. Additional lists can be compared by extending the box.