Programmierung Elektronischer Musik in Pd

Johannes Kreidler

Programmierung Elektronischer Musik in Pd

Johannes Kreidler

Veröffentlicht 27-01-2009

Zusammenfassung

Pd ist eine professionelle, leistungsstarke Programmiersprache zur elektronischen Klangverarbeitung. Sie ist open source, also frei im Internet erhältlich. Dieses Handbuch ist für Einsteiger und Fortgeschrittene, die Pd im Selbststudium erlernen wollen.

Inhaltsverzeichnis

Vorwort

Dieses Buch entstand aus meiner Erfahrung beim Unterrichten Elektronischer Musik. Erst beim Lehren wurde mir bewusst, welche Verständnisprobleme gegenüber der Materie auftreten können – zumal bei Schülern, deren Muttersprache nicht die Unterrichtssprache ist.

Pd (Pure Data) ist eine professionelle, leistungsstarke Programmiersprache zur elektronischen Klangverarbeitung. Sie ist *open source*, also frei im Internet erhältlich. Dies hat den Nachteil, dass ihre Anwendung nur in Instituten oder Internetforen diskutiert wird. Die dort übliche Verwendung komplizierter (englischer) Fachsprache erschwert Einsteigern den Zugang erfahrungsgemäß massiv. Genau jenen aber soll mein Buch helfen, die ersten Hürden beim Erlernen von Pd zu überwinden.

Der Hauptverfasser von Pd, Miller Puckette, schreibt derzeit ebenfalls ein Buch über Theorie und Technik Elektronischer Musik unter Anwendung von Pd. Nun dürfte es wohl kaum einen kompetenteren Lehrer zur Einweisung in eine Programmiersprache geben als den Autor der Programmiersprache selbst; mit seinem primär wissenschaftlichen Ansatz deckt er auch tatsächlich systematisch die Materie ab. Ihre didaktische Vermittlung an seine Anwender gestaltet sich damit allerdings schwierig. Nach meiner pädagogischen Erfahrung stellt Puckettes Buch enorm hohe Ansprüche an das mathematische, informatische und terminologische Verständnis seiner Leser.

Dem gegenüber bietet sich mein Buch mit seiner pädagogischen Ausrichtung zum Selbststudium an, und zwar in erster Linie für Komponisten. Es beginnt an der Basis aller Programmier- und Akustikkenntnisse und tastet sich langsam bis zu den avanciertesten Techniken Elektronischer Musik vor. Einiges aus dem Bereich der Physik, das an sich in den Kanon gehören würde, bleibt bewusst außen vor. Methodisch orientiert sich mein Buch vor allem am Hören, wodurch sein Gegenstand für Musiker motivierender und schneller zu erfassen ist als anhand abstrakter Formeln. Mathematisch beschränke ich mich auf das, was für das jeweilige Verständnis unerlässlich ist. Die aufgezählten Techniken zielen auf eine kompositorische Anwendung ab, weniger auf eine informatische, mathematische oder physikalische Erarbeitung der Phänomene und Strukturen. Meine Entscheidungen und Kommentare sind diesbezüglich freilich subjektiv und damit angreifbar.

Mein Dank hinsichtlich der Realisierung dieses Buches gilt Prof. Mathias Spahlinger für seine Unterstützung des Antrags, Prof. Orm Finnendahl für seine fachliche Betreuung, der Pd-Community für ihre Ratschläge und Patches, Esther Kochte für ihr Lektorat und die Codierung in DocBook-XML, Mark Barden für die englische Übersetzung sowie der Musikhochschule Freiburg bzw. dem Land Baden-Württemberg für die Finanzierung des Projekts, die es allen Interessierten ermöglicht, dieses Buch im Geiste der Open-Source-Bewegung frei im Internet zu nutzen. So wird die Praxis Elektronischer Musik hoffentlich noch weitere Kreise ziehen und der ästhetische Diskurs der Neuen Musik dadurch indirekt bereichert werden.

Johannes Kreidler, Januar 2008

Einführung in die Methodik dieses Tutorials

Der folgende Stoff beginnt an der Basis aller Computerkenntnisse. Daher sind die ersten Schritte minuziös beschrieben.

Pd wird hier unabhängig von der Plattform behandelt, also unabhängig vom Betriebssytem, auf dem es läuft (wie Linux, OS X oder Windows). Probleme, die im Zusammenhang mit dem jeweiligen Betriebssystem auftreten, müssen hier außen vor bleiben, da dies den Rahmen des Tutorials sprengen würde und es zudem höchstwahrscheinlich mit der Zeit unabsehbare Änderungen geben wird. Es wird also vorausgesetzt, dass sich Pd bei der Installation funktionsfähig in die Plattform und Hardware-Umgebung integrieren lässt (bei Problemen hierzu kann man sich zum Beispiel an das Internet-Pd-Forum „Pd-list“ wenden).

Zur Methodik des Buches: Jede Lektion beinhaltet einen theoretischen Teil und einen Anwendungsteil sowie einzelne Aspekte, die in einem Appendix ergänzt werden. Für besonders interessierte, fortgeschrittene Anwender gibt es weiterführende Informationen, die für ein Grundverständnis von Pd jedoch nicht erforderlich sind. Ich empfehle, das Buch zunächst ohne Berücksichtigung diese Abschnitte durchzuarbeiten und sie zu gegebener Zeit nachzuholen.

Zwischendurch sind immer wieder allgemeine Grundlagen der Akustik eingestreut. Die Übungen beinhalten nicht nur gezielte kompositorische Fragestellungen, sondern auch Anwendungen, die für Musiker im Alltag relevant sind, wenn sie sich etwa auf Hilfsmittel wie das Metronom oder Stimmgerät beziehen. So ist das Tutorial nicht nur für Komponisten, sondern auch für Interpreten relevant.

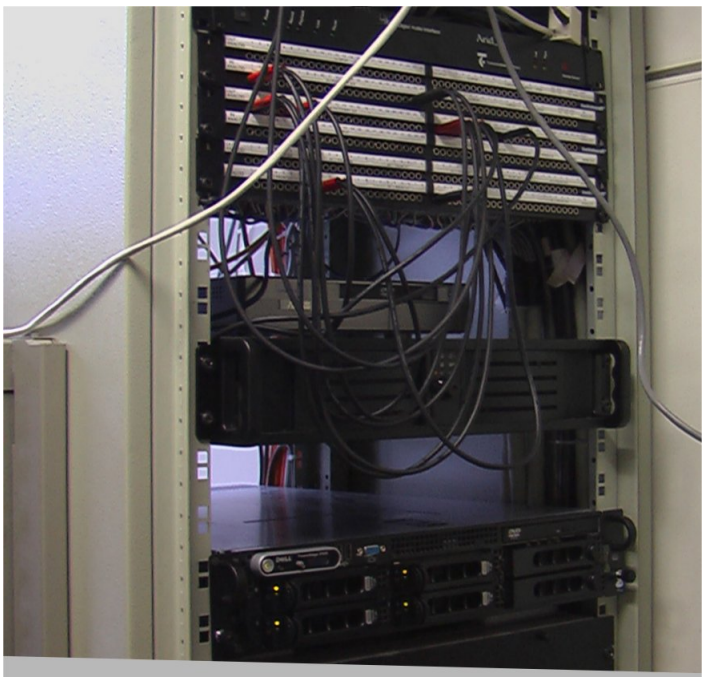
Kapitel 1. Einführung in Pd

1.1 Generelles

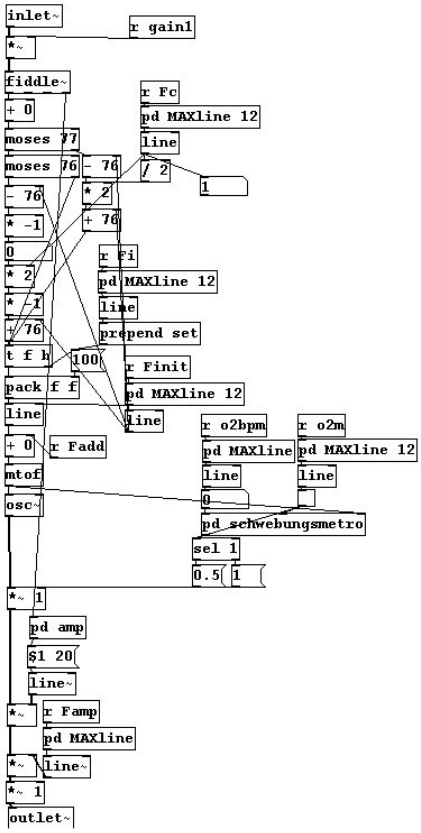
Pd (Pure Data) ist eine Programmiersprache für Elektronische Musik. Elektronische Musik am Computer zu erzeugen, heißt in der Fachsprache DSP (digital signal processing) oder zu deutsch: digitale Signalverarbeitung. „Digital“ bedeutet, dass die Information in Ziffern dargestellt wird – schließlich arbeiten Computer ausschließlich mit Zahlen. „Signal“ ist der Fachausdruck für eine spezielle Arbeitsweise des Computers im Umgang mit Klang. „Processing“ bezeichnet die Arbeitsweise des Computers.

Pd wurde vom Amerikaner Miller Puckette initiiert, der vormals die bekannte und ähnlich strukturierten Software Max/Msp mitentwickelt hat. Pd ist keine kommerzielle Software, das heißt, sie wurde nicht von einer Firma entwickelt und wird nicht verkauft. Statt dessen ist sie „open source“: Ihre Quelle liegt für jeden Interessierten offen. Der zu Grunde liegende Programmiercode ist also nicht (patentiertes) Eigentum eines Unternehmens, sondern für jeden frei verfügbar. Das bedeutet auch, dass jeder das Programm ändern kann, der über die entsprechenden Programmierkenntnisse verfügt. An der Weiterentwicklung von Pd sind neben Miller Puckette darum mittlerweile viele andere Programmierer, Musiker, Akustiker und Komponisten beteiligt. So gibt es auch keine endgültige, abschließende Version von Pd, sondern das Programm befindet sich in ständiger Entwicklung. Neben dem Vorteil, dass es frei und kostenlos im Internet verfügbar ist, wird es also „demokratisch“ auf professionellem Niveau laufend erweitert und optimiert. Der Nachteil dieser Lösung ist, dass es bislang keine ausführliche Einführung zu dem Programm gibt, die Interessierten ohne Programmiervorkenntnisse zumindest die Grundlagen erklären würde. Anders als eine Firma, die ein kommerziell bedingtes Interesse daran hat, dass die Anwendungen, die sie vertreibt, auch für Einsteiger verständlich sind, mangelt es der Open-Source-Bewegung an einer entprechenden Motivation. Dem soll dieses Buch Abhilfe schaffen.

Genauer gesagt ist Pd ein „real-time graphical programming environment for audio processing“, zu deutsch, eine „grafische Programmierumgebung zur Klangerzeugung in Echtzeit“. Traditionell arbeiten Programmierer mit textbasierten Programmiersprachen. Sie erstellen den sogenannten Code, starten die Verarbeitung des Computers und erhalten ein Ergebnis. Pd stellt für seine Programmfunktionen visuelle Objekte bereit, die der Anwender auf dem Bildschirm platziert und verändert. Diese visuellen Stellvertreter – kleine Kästchen, die miteinander verbunden werden – gehen auf die analogen Studios zurück, in denen vor dem Computerzeitalter Elektronische Musik produziert wurde: Verschiedene Geräte – jetzt symbolisiert durch unsere Kästchen – werden durch Linien miteinander verbunden, die - in Analogie zu den Kabeln - Verbindungen zwischen den Kästchen symbolisieren. (Wegen dieser Art von Verbindungen wird Pd eine datenstromorientierte Programmiersprache genannt.)



Ein analoges Studio – Geräte werden mit Kabeln verbunden.



Pd-Kästchen werden miteinander verbunden.

Der große Vorzug von Pd ist der Aspekt der „Echtzeit“. Das heißt, dass nicht – wie im traditionellen Programmierschema – erst ein Text eingeben wird, der dann vom Computer selbständig ausgeführt wird, sondern dass Änderungen während der Ausführungen gemacht werden können; wie an einem

klassischen Instrument hört der Benutzer sofort die Resultate und kann diese direkt ändern. Dadurch ist Pd hervorragend für live auf der Bühne agierende Künstler geeignet.

Mittlerweile ist Pd viel mehr als eine Programmiersprache für Elektronische Musik. Da sich Interessierte rund um den Globus an dem Projekt beteiligen können, gibt es nun auch eigens geschriebene Module für Video, Internet-Verbindung, Eingabe von Joysticks etc., die sogenannten „externals“. Es gibt sogar schon ganze Bibliotheken solcher Module („external libraries“). Einige davon sind bereits fester Bestandteil von Pd geworden.

1.2 Pd installieren und einrichten

Der Leser dieses Buches muss auf seinem Computer Pd verfügbar haben und alle hier besprochenen Vorgänge gleich im Programm umsetzen können. Ohne die begleitende praktische Anwendung wird dieses Tutorial schwer nachzuvollziehen sein.

Zunächst benötigen wir also einen Computer mit mindestens 128 MB Hauptspeicher, 500 Mhz-Leistung und ca. 500 MB Festplattenplatz (dies sind absolute Minimalwerte!). Als Betriebssystem kann man Linux, OS X oder Windows verwenden.

Wir laden die neueste Version von Pd-extended aus dem Internet herunter. Hierzu bitte nach „Pd extended“ bei einer Internet-Suchmaschine suchen. Da sich die Adresse des Download-Portals im Lauf der Zeit ändern kann, wird an dieser Stelle kein Link angegeben. Pd-extended ist eine durch zahlreiche Bibliotheken erweiterte Version des originalen Pd (das auch „Pd vanilla“ genannt wird). Die meisten hier besprochenen Übungen lassen sich zwar mit der originalen Pd-Version realisieren, aber doch nicht alle. Mit den zusätzlichen Objekten von Pd-extended ist vieles außerdem praktikabler. Unser Tutorial setzt Pd-extended mindestens ab Version 0.39 voraus.

Ist Pd installiert, starten wir es, in dem wir Pd im Verzeichnis Pd/bin/ ausführen. Es erscheint ein Fenster. Dies ist quasi die oberste Schaltzentrale. Hier testen wir zunächst, ob Pd funktioniert: Wir klicken im Haupt-Menü **Media # Test Audio and MIDI**. Wir klicken bei „Test Signal“ in das Kästchen neben „–40“, dann in das Kästchen neben „–20“. Jetzt sollte aus dem Lautsprecher des Computers ein Ton (der Kammerton a') erklingen. Funktioniert dies nicht, muss man die Hardware einrichten (bei **Media # Audio settings**). Weiteres hierzu kann an dieser Stelle nicht erklärt werden. Bei Problemen hilft die „Pd-list“ weiter, das Forum der Pd-Anwender im Internet. Wenn ein Mikrofon angeschlossen ist, sollten sich bei gewisser Lärmerzeugung zumindest in den zwei Kästchen links über „audio input“ die Zahlen gemäß der Lautstärke ändern. Funktioniert zumindest der Testton, können wir zunächst auch ohne Mikrofon mit dem Programmieren beginnen. (Spätestens in Kapitel 3 brauchen wir aber manchmal ein Mikrofon.)

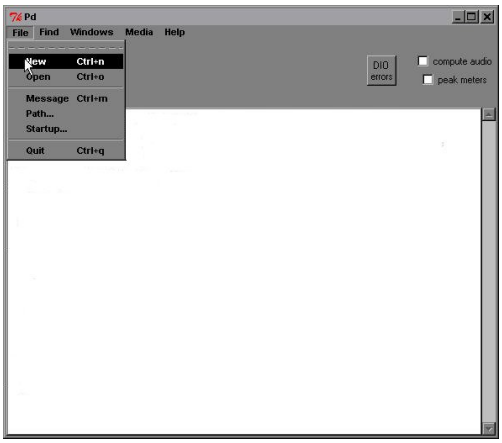
Kapitel 2. Erstes Programmieren mit Pd

In diesem Kapitel geht es noch nicht direkt um die Herstellung von Musik, sondern zunächst um computertypische bzw. Pd-typische Arbeitsweisen mit Daten. Da solches jedoch für Musiker erfahrungsgemäß etwas abstrakt und unattraktiv erscheint, werden wir möglichst viel von diesem Stoff gleich auf ein klingendes Beispiel anwenden – allerdings noch ohne dabei zu erklären, wie Klang im Computer eigentlich erzeugt wird. Das soll erst im folgenden Kapitel 3 Gegenstand sein. Die Beispielpatches muss man selber in Pd nachbauen. Dies hat einen guten Lerneffekt. Ab Kapitel 3 sind dann größere Patches auch als eigene Dateien verfügbar unter www.kreidler-net.de/pd/patches/patches.zip.

2.1 Einführung

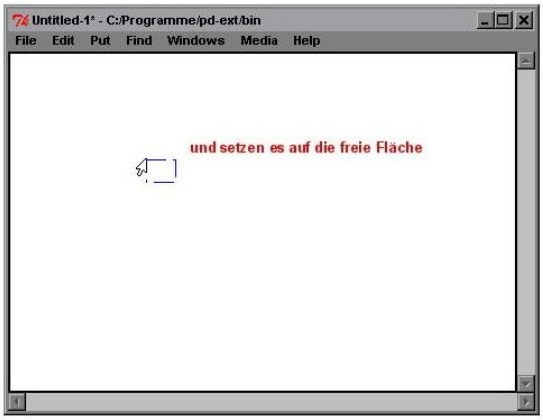
2.1.1 Ein einfaches Beispiel

Nachdem wir Pd gestartet haben, erscheint auf dem Monitor das Pd-Hauptfenster. Wir holen uns eine neue Programmierfläche, indem wir im Menü oben auf **File** und dann **New** klicken.



Ein neues Fenster geht auf. Wir setzen darin eine Objekt-Box: **Put # Object**, oder mit der Tastatur, wie dahinter steht: **Ctrl-1** (hier die Windows-Kürzel; sie sind auf anderen Plattformen eventuell anders).

... woraufhin am Maus-Zeiger eine blaue Box hängt ...

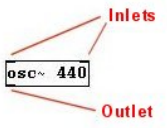


Anschließend klicken wir irgendwo auf die weiße Fläche des neuen Fensters, um die Maus wieder von der Objekt-Box zu entkoppeln. In die Box tippen wir ein: „osc~ 440“.

Um das Geschriebene zu fixieren, klicken wir zuletzt irgendwo auf die weiße Fläche außerhalb der Box:

(Das Zeichen „~“ bedeutet „Tilde“; speziell in Pd brauchen wir es oft.)

Wir sehen nun eine rechteckige Box mit kleinen schwarzen Rechtecken in den Ecken oben und unten. Die oberen Rechtecke heißen „Inlets“ (oder „Eingänge“), das untere Rechteck ist ein „Outlet“ (oder „Ausgang“).

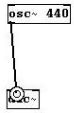


Wenn wir mit dem Cursor auf das Rechteck des Outlet gehen, verändert er sein Erscheinungsbild zu einem Kreis (quasi ein Steckplatz für ein Kabel).

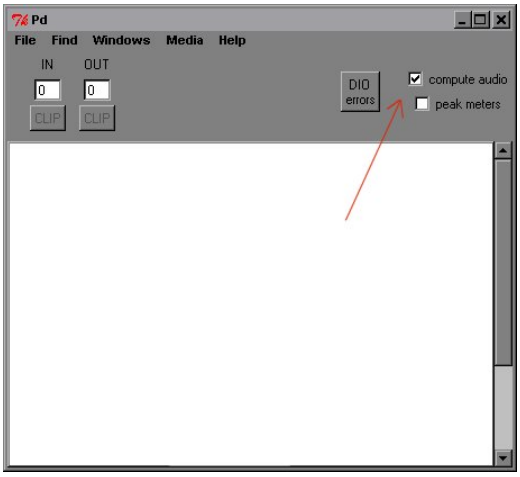
Klicken wir nun auf das Rechteck und bewegen die Maus bei gedrückter Maustaste, können wir eine Linie damit ziehen, als würden wir ein Kabel verlegen.

Noch haben wir jedoch kein Objekt, in welches das Kabel eingesteckt werden könnte und das Kabel verschwindet wieder, sobald wir die Maustaste loslassen. Setzen wir also ein weiteres Objekt wie vorhin beschrieben und nennen es „dac~“. Wir platzieren es unterhalb des „osc~“-Objekts, indem wir das Objekt einmal anklicken, so dass es blau wird, und bei gedrückter Maustaste durch Bewegung der

Maus das Objekt verschieben. Danach ziehen wir erneut ein Kabel aus dem Outlet von „osc~“ und führen es an den Inlet von „dac~“. Dort angekommen, wird der Cursor wieder zum Kreis.

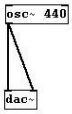


Lassen wir nun die Maustaste los, ist das Kabel von „osc~ 440“ sozusagen bei „dac~“ eingesteckt. Jetzt sollten wir einen Ton hören. Wenn nicht, muss geprüft werden, ob im Haupt-Pd-Fenster rechts oben bei „compute audio“ ein Häkchen gesetzt ist (in Linux: ob das Feld rot ist). Wenn nicht, das Häkchen per Mausklick setzen:

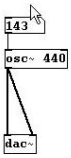


(Mit der Funktion „compute audio“ lässt sich einstellen, dass man programmieren kann, ohne dabei Klang zu generieren. Dies erspart dem Computer oft unnötige Rechenleistung – obwohl heutige Rechner damit kaum noch Probleme haben dürften.)

Wir hören nun also einen Ton, und zwar das eingestrichene a, den sogenannten Kammerton, der die Frequenz von 440 Hertz hat (die Bedeutung von „Frequenz“ und „Hertz“ wird später erklärt). Verbinden wir noch zusätzlich den Outlet von „osc~ 440“ mit dem rechten Inlet von „dac~“.



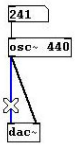
Damit sollte in beiden Lautsprechern des Computers etwas erklingen. Nun bringen wir eine Number-Box an (**Put # Number** oder Tastatur **Ctrl-3**) und schließen sie über ihren Outlet an den Inlet des Objekts „osc~“ an. Dann wechseln wir in den sogenannten „Execute-Mode“ (**Edit # Edit mode**, oder Tastatur **Ctrl-E**; der Cursor ändert sein Aussehen in einen Pfeil). Danach gehen wir mit der Maus auf die Number-Box, klicken diese an und bewegen die Maus bei gehaltener Taste auf und ab:



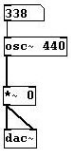
Dadurch ändern sich die Zahlen und somit die Tonhöhe. Es sollten allerdings Werte von mindestens 100 sein; der Werte-Bereich lässt sich feiner variieren, indem man die Shift-Taste gedrückt hält, gleichzeitig in die Box klickt und die Maus nach oben oder unten bewegt.

Eine andere Möglichkeit, Zahlen in die Number-Box einzugeben, wäre in die Box zu klicken, manuell über die Tastatur einen Wert einzugeben und ihn mit „Enter“ zu bestätigen.

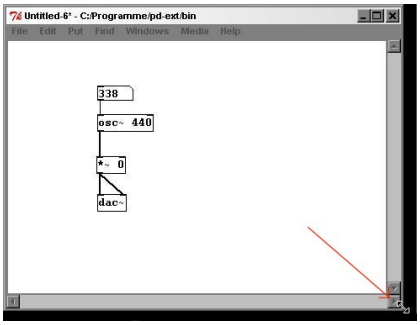
Nun wechseln wir wieder in den anderen Modus, den „Edit Mode“ (wieder **Edit # Edit mode**, oder Tastatur **Ctrl-E**). Mit dem Cursor, der nun wieder eine Hand ist, gehen wir über die Verbindung zwischen „osc~“ und „dac~“, wobei der Cursor zum X wird, und klicken darauf, so dass die Verbindungsschnur blau wird.



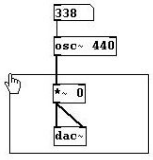
Dann gehen wir auf **Edit # Cut** oder drücken die Backspace-Taste. So wird eine Verbindung gelöscht. Außerdem löschen wir so auch die zweite Verbindung zum „dac~“. Nun bringen wir dazwischen ein weiteres Objekt an, nämlich „*~ 0“, und verbinden es folgendermaßen mit den anderen Objekten:



Schaffen wir uns nun etwas Platz: Wir vergrößern rechts unten das Fenster, in dem wir in die Ecke ganz rechts unten im Fenster klicken und bei gedrückter Maustaste eine neue Fenstergröße ziehen.



Dann klicken wir rechts unten in der Nähe des „dac~“-Objektes in den weißen Hintergrund, halten die Maustaste gedrückt und ziehen das erscheinende Rechteck so, dass darin das „dac~“- und das „*~“-Objekt eingeschlossen sind.

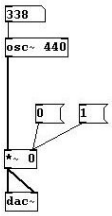


Auf diese Weise können wir immer einen Teil eines Patches selektieren. (So können auch Boxen gelöscht werden: Indem man sie selektiert und dann auf **Edit # Cut** geht oder einfach Backspace drückt.)

Wenn wir nun die Maustaste loslassen, sind die beiden markierten Objekte blau. Nun klickt man in eines der beiden markierten Objekte, hält geklickt und zieht sie zusammen nach unten, so dass wir mehr Platz darüber frei haben.

Um anschließend die Markierung wieder aufzuheben, klickt man irgendwo auf die weiße Hintergrundfläche.

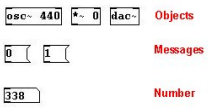
Dann bringen wir zwei „Message“-Boxen (**Put** # **Message** oder **Ctrl-2**) wie folgt an und versehen eine mit dem Inhalt „0“, die andere mit „1“.



Wir gehen wieder in den Execute-Mode (wieder **Edit** # **Edit mode** oder **Ctrl-E**) und klicken abwechselnd auf die beiden Message-Boxen: Bei 1 geht der Ton an, bei 0 aus.

2.1.2 Elemente der Oberfläche von Pd

In dem soeben erstellten Beispiel sind schon die meisten Elemente von Pd enthalten. Schauen wir genau hin – wir haben drei verschiedene Arten von Boxen verwendet: *Objekt*, *Message* und *Number*.



Die Objekt-Boxen sind viereckig, die Message-Boxen haben rechts eine Einbuchtung, die Number-Boxen rechts oben ein stumpfes Eck.

Alle diese Boxen haben Ein- und Ausgänge („Inlets“ und „Outlets“). Die Eingänge befinden sich immer oben, die Ausgänge unten. Man kann immer von einem Ausgang zu einem Eingang (in dieser Abfolge) Verbindungen ziehen. Es gibt einen Edit-Mode und einen Execute-Mode. Im Edit-Mode programmiert man, im Execute-Mode führt das Programm aus. Man kann die beiden Modi äußerlich durch die verschiedenen Cursor-Gestalten unterscheiden:

Schauen wir weiter genau hin: Es gibt zwei verschiedene Arten von „Kabeln“, dicke und dünne. Von der Number-Box zum „osc~“-Objekt verläuft ein dünnes Kabel, vom „osc~“-Objekt aus verläuft ein dickes Kabel. Das bedeutet, durch ein dickes Kabel strömen *Signale*, während durch ein dünnes nur *Kontrolldaten* gehen. Mit „compute audio“ im Pd-Hauptfenster stellen wir ein, ob die Signale tatsächlich geschickt werden sollen oder nur dann, wenn man das Häkchen aktiviert. Darüber hinaus haben alle Objekte, die Signale erzeugen oder mit Signalen als Input arbeiten (Input = das, was in ein Inlet geht; Output = was aus einem Outlet kommt), hinter ihrem Namen die Tilde: „~“ verzeichnet;

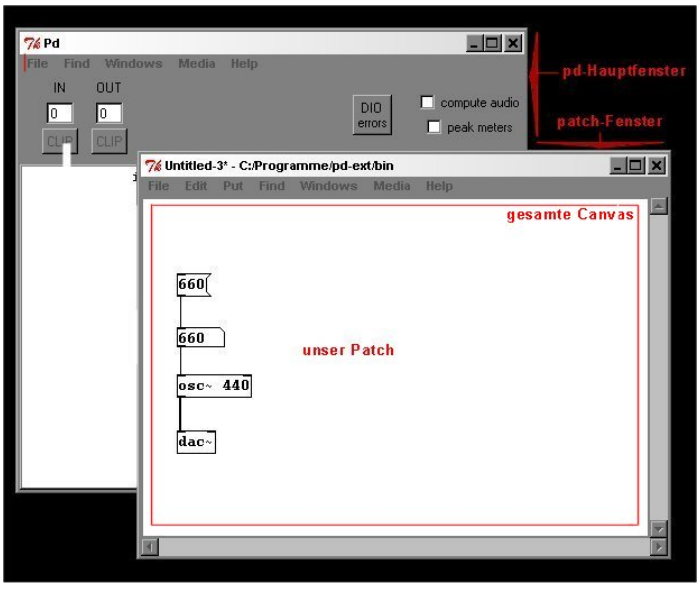
die haben alle anderen Objekte nicht! Diese beiden Ebenen nennen wir „Kontroll-Ebene“ (wo nur Kontrolldaten fließen) und „Signal-Ebene“ (wo auch Signale fließen).

Wir hatten zunächst das Objekt „osc~ 440“ geschrieben, den sogenannten „Oszillator“, und darauf hörten wir einen Ton mit 440 Hertz (was „Hertz“ ist, wird später erklärt). Dann hatten wir die Number-Box angebracht und darin neue Werte eingegeben, worauf der Ton nicht länger bei 440 Hertz. blieb, sondern die Frequenz des neu gegebenen Wertes annahm. Das ist eine grundlegende Struktur von Pd: Ein Objekt hat einen Namen (wenn es auch mit Signalen arbeitet, hat der Namen am Ende eine Tilde), dann kommt ein Leerzeichen und darauf folgen ein oder mehrere *Argumente* (hier war das Argument zunächst die „440“. Diese können bei den meisten Objekten ersetzt werden durch neue Werte, die in die Inlets eingegeben werden (in den meisten Fällen erfolgt die Änderung, anders als beim „osc~“-Objekt hier, dann durch den ganz rechten Inlet).

Ab dann gelten nur noch die Input-Werte (im Beispiel hier also 300 statt 440).

Als Informationen können wir in Number-Boxen oder in Message-Boxen Zahlen eingeben. In Message-Boxen können wir auch Buchstaben schreiben, das sind dann sogenannte *Symbole*. Alle diese Informationen nennen wir *Atoms*. Ein Atom steht also in einer Message-Box oder in einer Number-Box (mehr zu Atoms siehe Abschnitt 2.1.5).

Noch eine wichtige Bezeichnung: Das Programm, das wir schreiben, nennt man *Patch*. Ein Patch ist zunächst nur eine weiße Fläche, sozusagen eine weiße Leinwand, auf das wir dann unser Programm schreiben. Wir nennen diese weiße Fläche auch *Canvas* (englisch = Leinwand).



2.1.3 Zusammenfassung

- Es gibt den Edit Mode und den Execute Mode (man schaltet zwischen beiden hin und her mit **Ctrl-E** oder im Menü unter **Edit # Edit mode**). Im Edit Mode setzt man alle Teile eines Patches, im Execute Mode startet man alle Operationen sowie den Klang.

- Innerhalb eines Patches gibt es die Kontroll-Ebene und die Signal-Ebene (Kontroll-Objekte haben keine Tilde am Ende ihres Namens und sind mit dünnen Kabeln verbunden, Signal-Objekte haben eine Tilde am Ende und sind mit dicken Kabeln verbunden). Die Signal-Ebene ist nur aktiv, wenn im Pd-Hauptfenster rechts oben „compute audio“ aktiviert ist.

- Die Elemente des Patches sind *Objekte*, *Messages* und *Numbers*.

- Ein Objekt hat häufig ein oder mehrere Argumente (im Englischen „creation argument“ genannt), die aber durch einen Input geändert werden können.

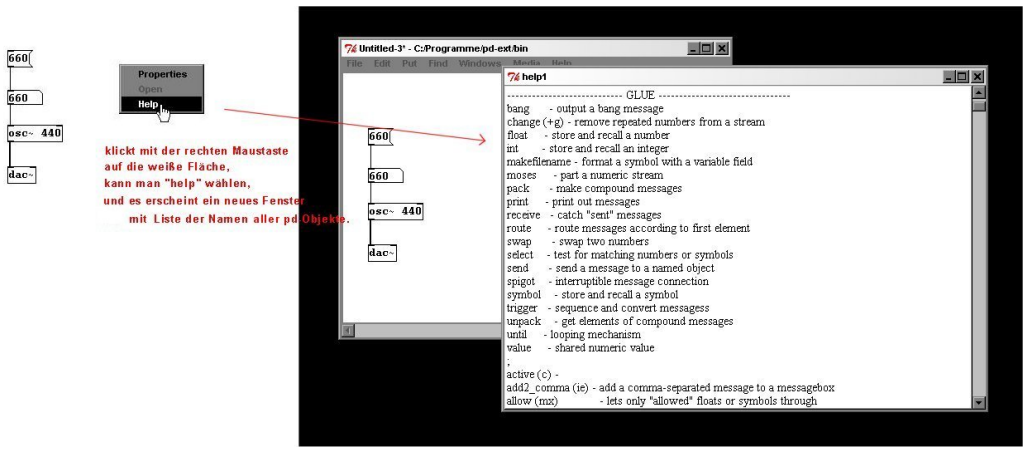
- Eine Message ist ein fester Wert im Execute mode und wird zusammen mit dem Patch gespeichert. Wenn in eine Message-Box geklickt wird, führt das dazu, dass die darin befindliche Message an alle Objekte verschickt wird, die mit ihrem Ausgang verbunden sind. In einer Number-Box hingegen kann man im Execute mode verschiedene Werte eingegeben, die dann jedoch nicht gespeichert werden.

2.1.4 Appendix

Noch ein paar zusätzliche Dinge, die die Arbeit mit Pd erleichtern:

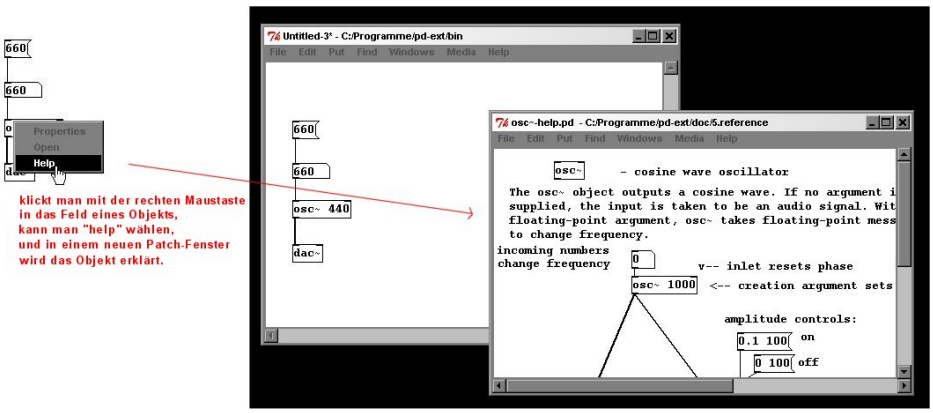
2.1.4.1 Liste aller Objekte

Klickt man mit der rechten Maustaste auf die weiße Fläche („Canvas“) des Patches, und öffnet das Menü **Help**, erscheint eine Liste mit allen Objekten von Pd.



2.1.4.2 help-Datei

Klickt man mit der rechten Maustaste in ein Objekt, öffnet sich ein Pulldown-Menü, über das sich die Help-Datei für das Objekt anwählen lässt, worin das entsprechende Objekt näher erklärt wird.



2.1.4.3 Duplizieren

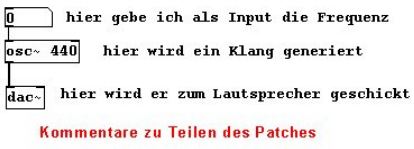
Sehr hilfreich wird es bald sein, Teile des Patches zu verdoppeln. Man selektiert einen Bereich (wie unter 2.1.1 beschrieben, als es darum ging, mehr Platz zu schaffen), so dass die selektierten Boxen blau sind, und geht auf **Edit** # **Duplicate** bzw. **Ctrl-D**. Damit ist der markierte Bereich verdoppelt und die Kopie erscheint als selektierter Bereich, der sich verschieben lässt (auf eine der blauen Boxen klicken, gedrückt halten, den Bereich verschieben und an der gewünschten Stelle die Maustaste wieder loslassen).

2.1.4.4 Short-Cuts

Grundsätzlich kann man wesentlich bequemer und schneller arbeiten, wenn man die Tastaturkürzel („Keyboard Shortcuts“) verwendet. Für viele der über die Menüs anwählbaren Funktionen gibt es Tastaturkürzel, die in den Menüs neben den entsprechenden Funktionen notiert sind.

2.1.4.5 Kommentare

Beim Programmieren kann es sehr schnell komplex werden. Um später noch die Bedeutung eines bestimmten Patches zu verstehen, empfiehlt es sich, seinen Patch zu *kommentieren*. Kommentare fügt man mit **Put** # **Comment** (bzw. **Ctrl-5**) ein. Damit lässt sich etwas Beliebiges schreiben, das den Patch erklärt.



Wenn wir das bislang Erläuterte begriffen haben, haben wir schon die wichtigsten Grundlagen der Struktur der Benutzeroberfläche von Pd erarbeitet. Nun kommen wir zur Struktur der Programmierung selbst.

2.1.5 Für besonders Interessierte: Atoms

Eine Message für ein Objekt hat zwei Teile: erstens den Hinweis auf eine Methode (Selector), zweitens kein, ein oder mehrere Werte (Argumente). Zum Beispiel ist die Message „5“ eigentlich die Message

„float 5“, mit den beiden Atomen „float“ und „5“. Die Message „bang“ besteht nur aus dem Selector „bang“ und enthält keine Argumente. Oder die Message „1 2 3 4 5“ ist eigentlich die Message „list 1 2 3 4 5“.

Ein Atom kann eines von drei Typen sein: eine Zahl (Programmiersprache = „float“) mit einem 32-Bit-Wert, ein Symbol, das eine Buchstabenfolge ist, oder ein Pointer, der eine Art Adresse ist (dazu kommen wir erst in Kapitel 5.2.3).

Die Message „float 5“ besteht aus den beiden Typenbezeichnern Symbol und Float, der Typ Symbol mit dem Wert „float“ (ein String) und der nachfolgende Typ Float mit dem Wert „5“.

Der Selector ist immer ein Symbol. Da Objekte auf veschiedene Messages unterschiedlich reagieren können, wird durch den Selector vorab eine nähere Bestimmung erzeugt..

2.2 Die Kontroll-Ebene

Zunächst müssen wir die Grundlagen der Kontroll-Ebene von Pd erarbeiten. Denn, wie schon erwähnt, arbeitet Pure Data zunächst nur mit Daten, d. h., mit Zahlen (und mit der Hilfe von Buchstaben). (In den Beispielen werden wir es aber möglichst gleich wieder auf den Klang anwenden.)

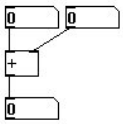
2.2.1 Rechenoperationen und Reihenfolgen

2.2.1.1 Theorie

2.2.1.1.1 Grundrechenarten

Wie schon gesagt wurde, arbeitet ein Computer nur mit Zahlen. Pd arbeitet hingegen mit Zahlen und mit sogenannten „Symbolen“, also Buchstaben. Grundlegender sind aber die Zahlen; wir haben ja im ersten Beispiel schon erfahren, dass so wichtige Parameter wie Tonhöhe oder Lautstärke eines Klanges in Pd nicht mit den herkömmlichen musikalischen Bezeichnungen, etwa *c1* für eine Tonhöhe oder *pianissimo* für eine Dynamik bestimmt werden, sondern eben nur mit Zahlen. Darum widmen wir uns nun zunächst den Grundlagen der Arbeit mit Zahlen auf der Kontroll-Ebene in Pd:

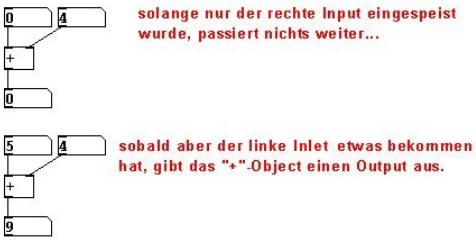
In Number-Boxen oder Message-Boxen können wir Zahlen eingeben. Bei bestimmten Objekten können wir nun Rechenoperationen mit diesen Zahlen durchführen. Setzen wir zum Beispiel das Objekt „+“ und schließen an seinen rechten und linken Inlet sowie an den Outlet Number-Boxen an:



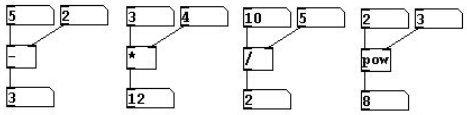
Geben wir nun in die rechte obere Number-Box die Zahl 4 ein (im Execute-Mode einmal in die Number-Box klicken, Zahl eintippen, Enter drücken) und in die linke obere Box die Zahl 5. Nun erscheint auf der unteren Box die Zahl 9, also die Summe von 4 und 5. Das „+“-Objekt hat also zwei Inlets, in die wir Zahlen eingeben, und einen Outlet, in dem das Ergebnis der Arbeit des Objekts, die in der Addition besteht, erscheint.

Hier sehen wir eine wichtige Regel von Pd: Bei Kontroll-Objekten mit mehreren Inputs müssen wir immer in der Reihenfolge von rechts nach links die Informationen eingeben. Oder anders gesagt, ein Objekt erhält Inputs. Es erzeugt aus diesen Inputs aber erst einen Output, wenn es den Input ganz links erhält (deshalb unterscheidet man auch „kalte“ Inlets, bei denen äußerlich erstmal nichts passiert, von

„heißen“, bei denen sofort nach Eingabe auch äußerlich etwas passiert.) Diesem Sachverhalt werden wir ständig wiederbegegnen.



Nach dem gleichen Prinzip funktionieren die anderen Grundrechenarten Subtraktion, Multiplikation, Division und Potenz:

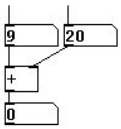


Möchte man nun mit dem selben Wert mehrere Rechnungen gleichzeitig durchführen, z. B. 3 * 3 und 3 * 4, dann kann man die Number-Box oder Message-Box einfach an mehrere Inlets anschließen (hier verwenden wir der Einfachheit halber als Multiplikator keinen Input, sondern ein Argument („* 3“ und „* 4“); in den vorherigen Beispielen haben wir hingegen gar keine Argumente vorgegeben, sondern nur Inputs. Wenn wir ein Objekt ohne Argument schreiben, geht Pd erst einmal von einem Wert „0“ als Argument aus.):

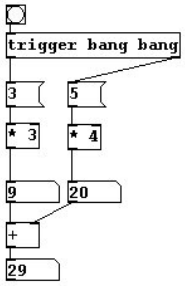
Möchte man zwei ganz verschiedene Rechnungen gleichzeitig ausführen, muss man seinen Mausklick mit einem „Bang“ vervielfältigen (**Put** # **Bang** oder **Shift-Ctrl-B**). Der Bang kann im Execute-Mode angeklickt werden.

2.2.1.1.2 Reihenfolgen

Wollen wir die beiden Ergebnisse wiederum miteinander addieren, müssen wir dafür sorgen, dass die beiden Ergebnisse in der richtigen Reihenfolge, nämlich von rechts nach links in das „+“-Objekt eingehen.

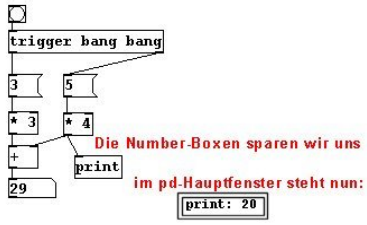


Um dies sicherzustellen, gibt es das „trigger“-Objekt:



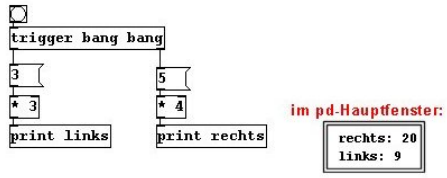
„trigger“ erhält als Input einen Bang, eine Zahl, ein Symbol, einen Pointer oder eine Liste (zu Pointer und Listen später mehr). Dadurch gestartet, vervielfacht „trigger“ diesen Input und gibt ihn oder ihn transfomiert in einen Bang *von rechts nach links* als Outputs heraus. Was herauskommen soll, bestimmt man durch Argumente (bang, float, symbol, pointer, list), hier ist es zweimal Bang, somit sind zwei Outlets entstanden (je mehr Argumente, desto mehr Outlets).

Die Number-Boxen der ersten Operationen, deren Ergebnisse dann wiederum addiert werden, können wir nun weglassen und die Outputs von oben gleich als Inputs weiterverwenden (Platz sparen). Wenn wir allerdings zwischendurch einmal wissen wollen, was an einem Ausgang herauskommt, können wir das Objekt „print“ verwenden.



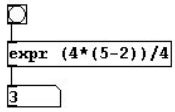
„print“ druckt quasi im Pd-Hauptfenster seinen Input aus. In diesem Fenster werden übrigens auch alle Fehler gemeldet. Schreiben wir z. B. das nicht existente Objekt „zzzgghhh“, wird es nicht erstellt und im Pd-Hauptfenster erscheint eine Fehlermeldung („zzzgghhh ... couldn’t create“).

So kann man beispielsweise sehen, in welcher Reihenfolge „trigger“ arbeitet, indem man „print“ noch verschiedene Argumente gibt (die Ergebnisse erscheinen im Pd-Hauptfenster untereinander, d. h., zeitlich hintereinander (zum Thema Reihenfolgen siehe noch 2.2.1.4)):



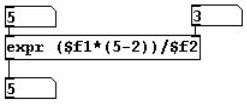
2.2.1.1.3 Expression

Größere mathematische Ausdrücke lassen sich zusammenfassen mit dem Objekt „expr“. Als Argument schreibt man die Rechnung (dabei eine korrekte Klammerung berücksichtigen, wie im Mathe-Unterricht gelernt!):

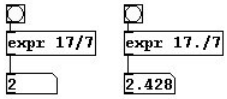


Um das Ergebnis zu erhalten, muss man einen Bang geben.

Man kann aber auch sogenannte „Variablen“ einsetzen; sie heißen hier: $f1, $f2, $f3 etc. (die Zählung beginnt bei 1). Damit entstehen Eingänge von links nach rechts, in die wir Zahlen eingeben können (wie immer kommt der Output, sobald ganz links ein Wert gegeben wurde. Daher sollten alle anderen Werte vorher eingegeben werden).



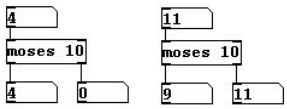
Zu beachten: Bei „expr“-Rechnungen (ohne Inputs), bei denen eine sogenannte float-Zahl, das heißt eine Komma-Zahl (und nicht eine ganze Zahl) herauskommen soll, muss man einer der beteiligten Zahlen einen Punkt beifügen (siehe zu floats noch 2.2.1.4).



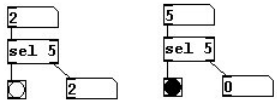
Potenzrechnungen folgen dieser Syntax: „expr pow ([Basis], [Exponent])“. Also im Fall von 2 hoch 3: „expr pow (2, 3)“.

2.2.1.1.4 Weitere Rechenoperationen

„moses“: Als Input kommt eine Zahl; diese wird von „moses“, je nach dem, ob sie im Verhältnis zum Argument kleiner oder größer/gleich ist, an verschiedene Outlets ausgegeben. Wenn man also „moses“ das Argument 10 gibt und als Input eine Zahl kleiner als 10, kommt diese im linken Outlet wieder heraus. Gibt man als Input 10 oder eine größere Zahl ein, kommt diese dann aus dem rechten Outlet heraus.

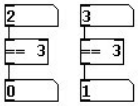


„select“ (meist abgekürzt: „sel“): Als Input eine Zahl, als Output links ein Bang, wenn die eingehende Zahl mit dem Argument übereinstimmt. Alle anderen eingehenden Zahlen werden rechts unten ausgegeben.



Relational Tests

„==“: Ist der linke Input gleich dem Argument bzw. dem rechten Input, erscheint als Output eine 1, ansonsten 0:



„>=“: Ist der linke Input größer als das Argument bzw. der rechte Input oder gleich, kommt als Output 1, ansonsten 0.

„>“: Ist der linke Input größer als das Argument bzw. der rechte Input, kommt als Output 1, ansonsten 0.

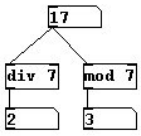
„!=“: Ist der linke Input ungleich dem Argument bzw. des rechten Input, kommt als Output 1, ansonsten 0.

„<“: Ist der linke Input kleiner als das Argument bzw. der rechte Input, kommt als Output 1, ansonsten 0.

„<=“: Ist der linke Input kleiner als das Argument bzw. der rechte Input oder gleich, kommt als Output 1, ansonsten 0.

Noch zwei weitere Rechenmodule:

Das Ergebnis einer Division kann man statt in einem Kommawert (17 / 7 = 2.428) auch differenziert ausdrücken: 17 / 7 = 2 Rest 3. Ein solches Ergebnis mit „Rest“ erhalten wir in Pd mit „div“ und „mod“:



Dann gibt es noch wichtige höhere mathematische Operatoren (zum Verständnis bitte im Mathebuch aus der schulischen Oberstufe nachschlagen):

„sin“ = Sinusfunktion

„cos“ = Cosinusfunktion

„tan“ = Tangensfunktion

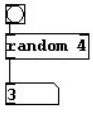
„log“ = (natürlicher) Logarithmus

„abs“ = Betrag

„sqrt“ = Quadratwurzel

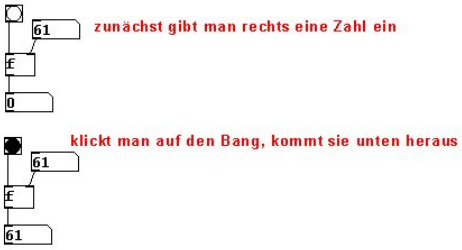
Zuletzt noch ein Algorithmus (Algorithmen sind Rechenoperationen, die der Rechner auf Basis eingegebener Werte selbständig durchführt):

„Random“ erzeugt eine Zufallszahl innerhalb eines gegebenen Rahmens. Die untere Grenze ist standardmäßig („by default“) 0; die obere Grenze gibt man als Argument ein (nur ganze Zahlen); die obere Grenze ist exklusiv, das heißt, wenn man „random 4“ eingibt, kommt bei jedem Bang als Input unten eine zufällige Zahl heraus, und zwar entweder 0, 1, 2 oder 3.



2.2.1.1.5 Float und Counter

Lernen wir noch im Zusammenhang mit Zahlenoperationen das „float“-Objekt kennen (Abk. „f“). Es ist ein Speicher für Zahlen. In den rechten Input gibt man eine Zahl ein. Damit wird sie im Objekt gespeichert und kann irgendwann später wieder abgerufen werden: Schickt man in den linken Inlet einen Bang, kommt die gespeicherte Zahl im Outlet wieder heraus (siehe zu „float“ auch 2.2.1.4).



Man kann auch eine Zahl direkt in den linken Eingang schicken, dann kommt sie gleich unten wieder heraus (und wird für den weiteren Abruf (Bang) gespeichert).

Oft wird bei Pd ein Zähler (engl. „counter“) gebraucht, der von einem Startwert aus in ganzen Zahlen hochzählt. Dies sieht dann so aus:

Zur Erklärung:

Wir geben dem „f“-Objekt zuerst den Startwert „0“. Wenn wir dann das erste Mal auf den Bang links klicken, kommt aus dem „f“-Objekt die 0 heraus und geht in das „+ 1“-Objekt hinein. Aus dem kommt unten dann 0 + 1 = 1 heraus. Diese 1 geht wieder rechts oben in das „f“-Objekt, so dass, wenn ich das nächste Mal auf den Bang klicke, diese 1 herauskommt, und dann im „+ 1“-Objekt zur 2 wird usw.

2.2.1.1.6 Zusammenfassung

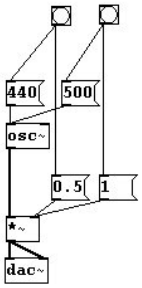
- An den Rechen-Objekten wird eine grundsätzliche Regel von Pd ersichtlich: Die Inputs für ein Kontroll-Objekt sollten immer von rechts nach links erfolgen. Um dies zu gewährleisten, benötigen wir häufig das Objekt „trigger“, das mehrere Outputs konsequent nacheinander von rechts nach links ausgibt.
- Ein „Bang“ ist wie ein Mausklick, der weitergegeben oder empfangen wird.
- Mit „print“ lassen sich im Hauptfenster Zwischenergebnisse eines Patches anzeigen. Was zeitlich nacheinander folgt, ist hier untereinander aufgeführt.

2.2.1.2 Anwendungen

Kommen wir zu Anwendungen des gerade Besprochenen (alles, was mit Klang zu tun hat wird erst später genauer erklärt):

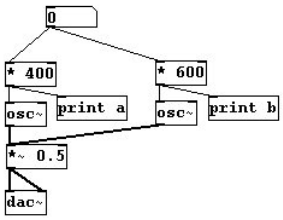
2.2.1.2.1 Zwei Töne – zwei Lautstärken

Wollen wir zum Beispiel zwischen zwei Tönen wechseln, einem tieferen, leiseren und einem höheren, lauteren, können wir das durch folgende Kombination erreichen, da wir nun auf die beiden Bangs für jeden Ton klicken können:



2.2.1.2.2 Ein Intervall

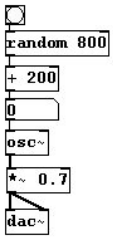
Für einen Zweiklang braucht man zwei „osc~“-Objekte. Wenn man wie folgt die Werte der Number-Box ändert, bewegt sich ein Intervall (Quint) auf und ab:



Im Pd-Hauptfenster werden hier, dank des „print“-Objekts, die Frequenzen der beiden Töne angezeigt.

2.2.1.2.3 Zufallsmelodie

Der Zufall kommt ins Spiel!



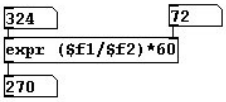
Bei jedem Klick auf den Bang erklingt ein anderer Ton zwischen 200 und 1000 Hertz – eine Zufallsmelodie.

Noch ein paar reine Rechenbeispiele:

2.2.1.2.4 Runden

2.2.1.2.5 Wie lange dauert eine Partitur?

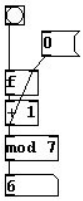
Eine Angabe, die Komponisten immer wieder benötigen: Man hat ein Stück im Tempo Viertel = 72 und einer gesamten Anzahl von 324 Vierteln geschrieben. Wie lange dauert das Stück in Sekunden?



Ergebnis: 270 Sekunden, also 4 Minuten 30 Sekunden.

2.2.1.2.6 Reihe hochzählen

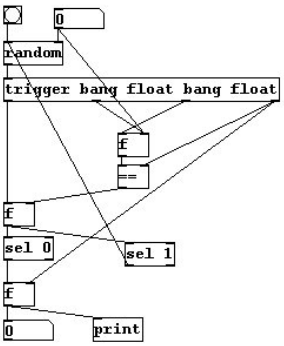
Dieser Counter zählt immer von 0 bis 6; nach der 6 fängt er wieder bei 0 an.



2.2.1.2.7 Random ohne Dopplungen

Wer bisher alles Vorgestellte verstanden hat, sollte folgende Aufgabenstellung meistern können, die allerdings nicht ganz trivial ist:

Erstellen Sie einen Random-Patch, in dem es keine Zahlenwiederholungen gibt, also nie zweimal hintereinander dieselbe Zahl erscheint (im Gegensatz zum normalen „random“-Objekt). Viel Erfolg beim Analysieren der Lösung!



2.2.1.2.8 Weitere Aufgabenstellungen

a) Erzeugen Sie zwei Zufallsmelodien gleichzeitig.

b) Wählen Sie mit zwei Bangs zwei verschiedene (beliebige) Intervalle an.

c) Berechnen Sie mit „expr“ Exponentialfunktionen, z. B. $y = x^2$ oder $y = x^{(2 + x)}$ oder $y = 1 - (2^x)$.

2.2.1.3 Appendix

2.2.1.3.1 Input für Bang

Ein Bang ist quasi ein Mausklick. Er kann also angeklickt werden und gibt diesen Klick weiter bzw. kann als Input einen Klick erhalten und darstellen (und wiederum weitergeben). Dieser Input muss aber nicht selbst ein Bang sein. Das Objekt „bang“ konvertiert jedes Kontroll-Ereignis, das es als Input erhält, in einen Bang, also zum Beispiel auch jede Zahl:

2.2.1.3.2 Zahlendarstellung

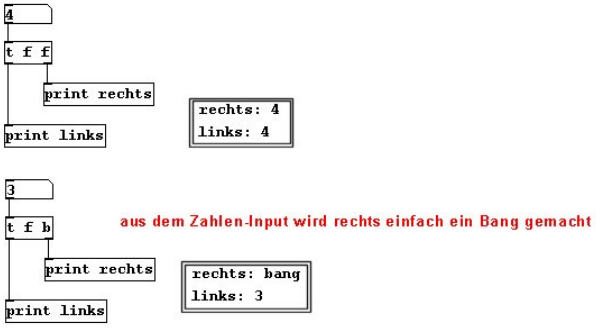
Zahlen mit vielen Kommastellen sind nicht vollständig mit der Standard-Number-Box zu lesen. Man kann die Number-Box aber vergrößern, indem man mit der rechten Maustaste in die Box klickt und auf „Properties“ geht, bei „width“ einen größeren Wert eingibt und abschließend auf „Ok“ klickt.

Ein weiterer Aspekt sind Zahlen größer als 999999. Sie werden vereinfacht dargestellt, nämlich als Produkt (mit max. zwei Kommastellen) von 1000000. Die 1000000 wird dargestellt als „e+006“.

Entsprechendes gilt für Zahlen kleiner als -999999 und für Zahlen zwischen 1 und -1 mit mehr als vier Stellen hinter dem Komma.

2.2.1.3.3 Mehr zu Trigger

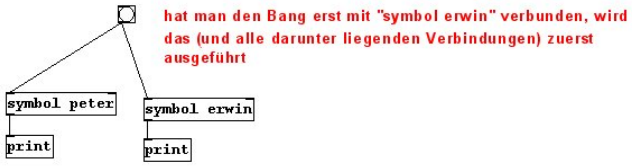
Das „trigger“-Objekt kann nicht nur Bangs verteilen, sondern auch Zahlen (später werden wir weitere Möglichkeiten kennenlernen). Es wird übrigens meist als „t“ abgekürzt, und statt der Argumente „bang“ und „float“ gebraucht man nur „b“ und „f“:



2.2.1.4 Für besonders Interessierte

2.2.1.4.1 Über Reihenfolgen

By default werden (derzeit) Objekte bzw. Verbindungen in der Reihenfolge ausgeführt, in der sie (zeitlich) gesetzt wurden:



Sichtbar ist dies natürlich nicht und darum grundsätzlich zu vermeiden!

2.2.1.4.2 Zu Float

„f“ steht für „floating point“, zu deutsch „Fließkomma-Zahl“. Genau genommen ist damit eine Zahl mit Werten hinter dem Kommabereich gemeint, im Gegensatz zu ganzen Zahlen. Wenn man nur mit ganzen Zahlen arbeiten will, kann man statt „float“ in Pd auch immer „int“ (Abk. „i“) einsetzen. Anders als Max/Msp, arbeitet Pd jedoch grundsätzlich mit float-Zahlen.

2.2.2 Verschiedenartige Daten

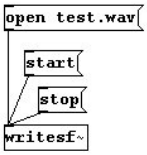
2.2.2.1 Theorie

2.2.2.1.1 Bang – ein GUI-Objekt

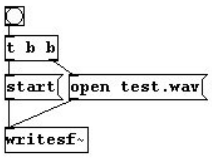
Der „Bang“, quasi ein Mausklick, steht genau genommen für die Buchstabenkombination b-a-n-g. Buchstabenkombinationen sind als sogenannte Symbole neben den Zahlen die zweite Datenform, mit der Pd arbeitet. Einige Objekte erkennen bestimmte Wörter und arbeiten auf deren Input hin. Viele Objekte reagieren auf das Symbol „Bang“. Da es so häufig vorkommt, gibt es speziell für „Bang“ eine grafische Repräsentation, den aufblinkenden Kreis (**Put # Bang**). Eine solche Repräsentation nennt man „GUI“-Objekt. (GUI = graphical user interface, also eine graphische Darstellung von etwas bzw. eine veränderbare Grafik, die dadurch neue Werte erzeugt und weitergeben kann).

2.2.2.1.2 Messages

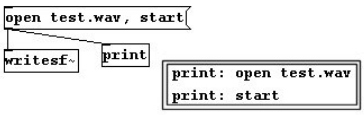
Betrachten wir in diesem Zusammenhang das Objekt „writesf~“ (ausnahmsweise sei hier noch ein Audio-Objekt eingeführt, da Symbole meist in diesen Zusammenhängen gebraucht werden; im Audio-Kapitel wird näher auf das Objekt selbst eingegangen), welches Klang als wav-Datei speichert. Es funktioniert folgendermaßen: Zunächst weisen wir ihm über die Message-Box den Namen zu, unter dem der Klang gespeichert werden soll, und zwar mit der Message: „open [Dateiname]“; wenn also die Datei zum Beispiel „test.wav“ heißen soll, dann lautet der Vorgang „open test.wav“. Danach geben wir mit den Messages „start“ und „stop“ Anfang und Ende der Aufnahme an.



Normalerweise wählen wir den Namen aus und starten dann auch gleich die Aufnahme. Die Reihenfolge bleibt natürlich wichtig – ehe das „writesf~“-Objekt mit der Aufnahme beginnen kann, muss es wissen, wie die zu speichernde Datei heißen soll. Dies könnte man beispielsweise mit „trigger“ lösen:



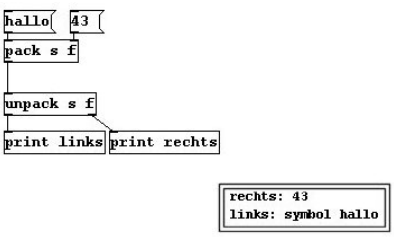
Messages können aber auch hintereinander geschickt werden, in dem man sie, getrennt durch ein Komma, in dieselbe Message-Box schreibt:



2.2.2.1.3 Listen

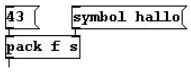
Die Message „open test.wav“ ist eine Verbindung aus zwei Symbolen (denn es sind zwei durch ein Leerzeichen getrennte Wörter). Eine solche Reihung von zwei oder mehreren Symbolen (oder Zahlen) nennt man „Liste“. Mit dem Objekt „pack“ kann man eine Liste aus mehreren „Elementen“ erstellen. Als Argumente gibt man Hinweise, welche Art von Elementen die Liste enthalten soll. Eine Zahl wird, wie bei „trigger“, mit „float“* (Abk. „f“) ausgedrückt, ein Symbol mit „symbol“ (Abk. „s“). Will man also aus den beiden Messages „hallo“ und „43“ eine Liste erstellen, verwende man das pack-Objekt wie dargestellt:

Auch hier gilt wieder: Erst wenn der Inlet ganz links eine Eingabe erhält, erfolgt der (richtige) Output. (Klickt man zuerst auf „hallo“, ohne vorher den rechten Input für „pack“ gegeben zu haben, erscheint stattdessen nur ein "list hello 0".) Den Output von „pack“ können wir vorerst nur mit dem „print“-Objekt sehen. Dieses zeigt dann an: „list hallo 43“. Diese Liste kann man nun aber auch wieder in ihre Elemente zerlegen, und zwar mit dem Umkehrungsobjekt (Umkehrungsobjekte gibt es in Pd viele) „unpack“, das nach dem gleichen Prinzip wie „pack“ funktioniert, nur erscheint hier als Ouput, was dort als Input erscheint.

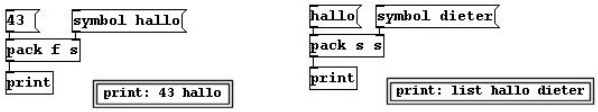


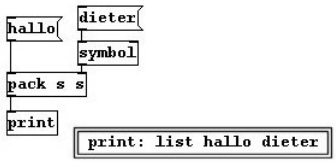
„print“ gibt nun „43“ und „symbol hallo“ aus. Alles, was nicht Zahl ist, wird also mit dem besonderen Hinweis („Selector“ genannt) seiner Datenart ausgewiesen.

Dies gilt auch zu beachten, wenn ein anderer Input von „pack“ als der ganz linke ein Symbol ist; dann muss Folgendes im Input stehen:



Ein Problem mit „pack s s“: Allein der erste Input braucht nicht extra als Symbol ausgewiesen zu sein. Im zweiten aber muss dann entweder „symbol“ vorangestellt sein oder die Message muss noch durch ein „symbol“-Objekt konvertiert werden:



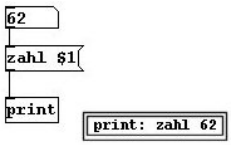


Hier sehen wir auch: Eine Liste, die mit einer Zahl beginnt, wird nicht extra als Liste bezeichnet; beginnt sie hingegen mit einem Symbol, steht ausdrücklich „list“ dabei.

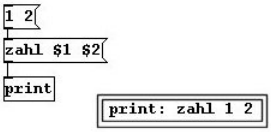
2.2.2.1.4 Messages mit Variablen

Werfen wir noch einen genaueren Blick auf die Message-Boxen:

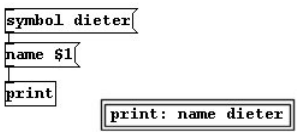
In den Inhalt einer Message-Box lassen sich Variablen integrieren. Dies erfolgt hier ähnlich, aber doch etwas anders als bei „expr“: Zunächst heißen hier die Variablen nur „$1, $2“ etc. Gibt man bei „zahl $1“ etwa als Input eine Zahl ein, erscheint als Output aus der Message-Box der vollständige Ausdruck mit dieser Zahl.



Sollen es aber mehrere Variablen sein, „zahl $1 $2“, entstehen nicht mehrere Eingänge (wie in „expr“), sondern es verbleibt der eine Eingang, dem wir hier mehrere Zahlen als Liste geben:



Symbole müssen als Symbol gekennzeichnet sein:

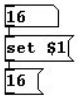


2.2.2.1.5 Messages: Set

Wir können den Inhalt einer Message-Box auch komplett neu bestimmen, und zwar mit einer vorgeschalteten Message-Box mit dem Symbol „set“:

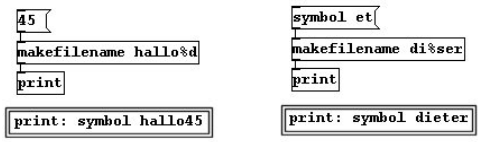
So kann der Inhalt einer Message-Box im Execute-Mode (vgl. den letzten Punkt unter 2.1.3) geändert werden.

In Kombination mit einer Variablen kann man dann beispielsweise aus dem Output einer Number-Box eine Message machen:



2.2.2.1.6 Makefilename

Nicht zulässig ist, eine Variable ohne trennendes Leerzeichen zu integrieren. Dafür benötigt man das Objekt „makefilename“, das Symbole mit variablem Feld ohne Leerzeichentrennung erstellt. Die Variablen, die man in ein Argument einfügt, lauten hier für Zahlen „%d“, für Symbole „%s“:



2.2.2.1.7 Openpanel

Das Objekt „readsf~”“ spielt ein schon bestehendes Soundfile ab, das zum Beispiel auf der Festplatte gespeichert ist. Es braucht eine Message „open [Name des Soundfiles]“. Mit „Name des Soundfiles“ ist dessen Ort auf einem Datenträger gemeint. Das heißt, wenn der Patch, in dem wir „readsf~“ verwenden, im Verzeichnis c:/Pd/Pd-patches/ gespeichert ist und das Soundfile „hallo.wav“ ebenfalls in diesem Verzeichnis liegt, müssen wir nur „open hallo.wav“ als Input geben. Ist „hallo.wav“ aber zum Beispiel im darüber liegenden Verzeichnis c:/Pd/, müssen wir schreiben: „../hallo.wav“ oder wenn es im darunter liegenden Verzeichnis c:/Pd/Pd-patches/soundfiles/ liegt, „/soundfiles/hallo.wav.“ Wenn es in c:/soundfiles/ liegt: „open ../../soundfiles/hallo.wav“. Oder wenn es auf einem anderen Datenträger liegt, zum Beispiel auf d:/soundfiles, muss es heißen „open d:/soundfiles/hallo.wav“.

Diese manchmal komplizierten Verzeichnispfad-Ausdrücke lassen sich mit dem Objekt „openpanel“ einfacher wiedergeben. Erhält es einen Bang, öffnet sich ein Fenster mit dem Inhalt der verfügbaren Datenträger des Computers. Klicken wir darin nun eine Datei doppelt an, gibt „openpanel“ in Pd den gesamten Pfad für diese Datei (als Symbol) aus:

(Ist ein Patch noch nicht gespeichert, geht Pd (unter Windows) vom Pfad pd/bin/ aus.)

2.2.2.1.8 Einfache Speicher für Daten

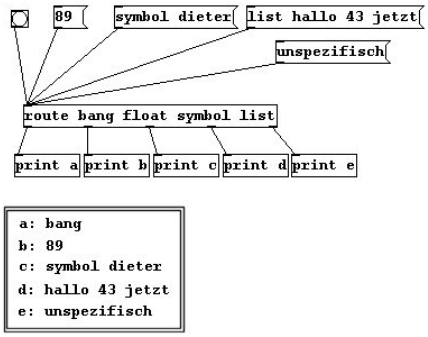
Wie schon mit dem „float“-Objekt erklärt (2.2.1.1.5), können Daten innerhalb des Patches mit den Objekten „float“, „symbol“ und „lister“ gespeichert werden (sind allerdings weg wenn der Patch geschlossen wird). „float“ und „lister“ werden meist abgekürzt zu „f“ und „l“.

Der rechte Inlet erhält eine Zahl, ein Symbol oder eine Liste, die damit im Objekt gespeichert werden und später wieder abgerufen werden können, indem man in den linken Inlet einen Bang schickt; das Gespeicherte erscheint dann im Outlet.

Zahl, Symbol oder Liste lassen sich auch direkt in den linken Eingang schicken – dann kommen sie unmittelbar unten wieder heraus (und werden im Objekt bis auf weiteres gespeichert).

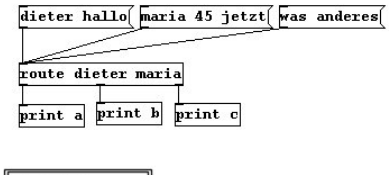
2.2.2.1.9 Route

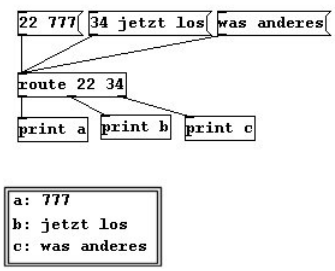
Ein Objekt, das verschiedene Datentypen sortieren kann, ist „route“. Es kann sowohl Typen (Zahl, Symbol, Liste, Bang) zuweisen ...



(Alles, was nicht zugewiesen werden kann, wird im ganz rechten Ausgang ausgeschieden.)

... als auch Listen bestimmten selbst gewählten Namen zuordnen:



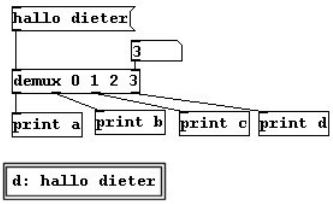


Zahlen und Symbole lassen sich hier allerdings nicht vermischen; „route 22 dieter“ funktioniert also nicht.

2.2.2.1.10 Demultiplex

„route“ verteilt einen Input auf verschiedene Ausgänge, je nach Präfix. „demultiplex“ (Abk. „demux“, beide in pd extended) verteilt einen Input je nach Angabe im anderen Inlet auf verschiedene Ausgänge. Zunächst erhält „demux“ als Argument die Nummern der Ausgänge, angefangen bei 0: „demux 0 1 2 3“.

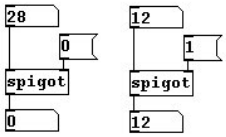
Es sind nun zwei Eingänge (bei „demux“ sind es immer nur zwei) und vier Ausgänge (entsprechend den vier Argumenten) entstanden. In den rechten Inlet geben wir nun eine Zahl ein, die die Nummer des Ausgangs bezeichnet, zum Beispiel 3. Danach geben wir irgendetwas (Zahl, Symbol oder Liste) in den linken Inlet ein und es kommt am dritten Outlet heraus.



Hier wird ersichtlich, dass Pd häufig bei 0, nicht bei 1 zu zählen beginnt.

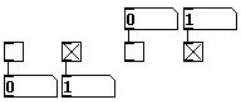
2.2.2.1.11 Spigot

Ein weiteres wichtiges Objekt ist „spigot“. Je nachdem, ob sein rechter Input 0 oder 1 ist, lässt „spigot“ einen Input entweder durch oder nicht – ähnlich wie ein Tor, das entweder offen oder geschlossen ist.



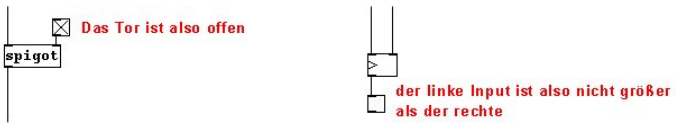
2.2.2.1.12 Toggle

Wie bei „spigot“, haben wir schon beim Objekt „==“ und den anderen Relational Tests gesehen, dass 0 und 1 häufig gebrauchte Informationen in Pd sind. Daher gibt es – ähnlich wie „bang“ für einen Mausklick – für den Wechsel zwischen 0 und 1 ein eigenes grafisches Objekt, den sogenannten „toggle“ (**Put** # **Toggle** oder **Shift-Ctrl-T**).



Toggle sieht aus wie ein Ein- und Ausschalter und kann in vielen Fällen auch so verstanden werden. Aber man sollte sich immer bewusst sein, dass in der Maschinensprache jeweils nur der Wechsel zwischen 0 und 1 dargestellt wird.

So können wir an „spigot“ einen Toggle anschließen und damit leicht erkennen, ob das „Tor“ gerade offen ist oder geschlossen. Oder ein Relational Test zeigt, ob etwas zutrifft oder nicht:

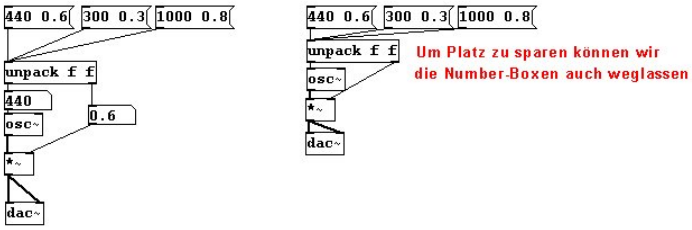


2.2.2.2 Anwendungen

Wenden wir nun einiges hiervon wieder an:

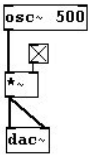
2.2.2.2.1 Eine Liste mit Tonhöhe und Lautstärke

Mit Listen wollen wir einem Oszillator eine Tonhöhe zuweisen und eine Lautstärke daran koppeln:



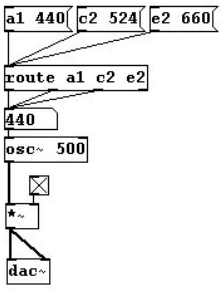
2.2.2.2.2 Ein-/Ausschalter

Im ersten Beispiel haben wir schon gesehen, dass wir den Ton mit den Messages „1“ und „0“ ein- und ausschalten konnten. Folglich können wir diesen Schalter auch mit einem Toggle bauen:



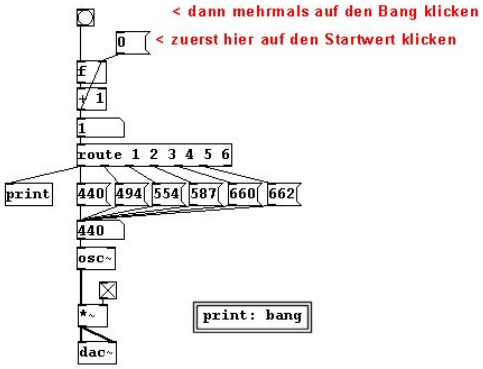
2.2.2.2.3 Tonhöhen mit Namen

Einem Oszillator wollen wir verschiedene Tonhöhen mit (frei gewählten) Namen zuordnen:



2.2.2.2.4 Eine einfache Sequenz

Wir bauen einen Zähler, der bei jedem Bang dem Oszillator eine bestimmte andere Tonhöhe zuweist:

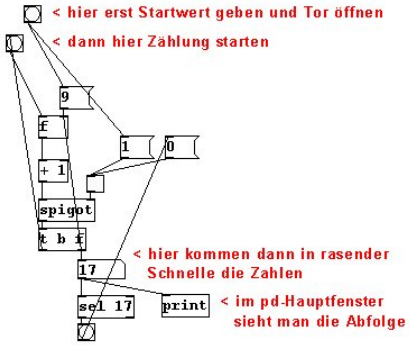


Erhält „route“ statt einer Liste nur den passenden Wert zu einem seiner Argumente, gibt es an dem entsprechenden Ausgang einen Bang aus. In diesem Beispiel ist „route“ also eine Zusammenfassung mehrerer Selektoren (man hätte stattdessen auch eine Reihe von „sel-“Objekten an den Zähler anschließen können: „sel 1“, „sel 2“ etc.).

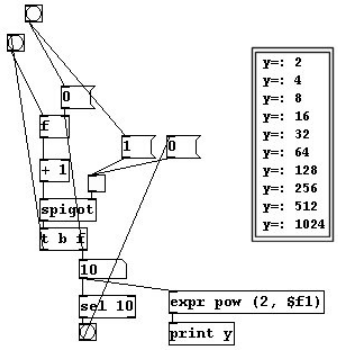
Den ganz rechten Ausgang von „route“ brauchen wir nicht zu ‚verkabeln’, solange der Input immer zu den Argumenten von „route“ passt.

2.2.2.2.5 Ein limitierter Counter

Wir bauen ein Zähler, der automatisch von 10 hochzählt und bei 17 stoppt:



So können wir etwa Funktionswerte für einen bestimmten Bereich schnell erhalten. Hier als Beispiel einfach die quadratische Funktion $y = 2^x$ für den Bereich von 1 bis 10:



Bei solchen Rekursionen (wo ein Output wieder zum Input wird) wie der Counter sie beinhaltet, muss man sehr aufpassen, nicht in eine sogenannte Endlosschleife zu geraten. Geben wir in diesem Beispiel hier nach erstmaliger Rechnung nicht wieder den Anfangswert, sondern öffnen das Tor und starten gleich die Rechnung, wird über 10 hinaus ohne Ende immer weiter hochgezählt.

2.2.2.2.6 Weitere Aufgabenstellungen

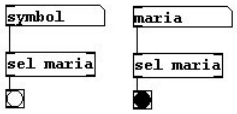
a) Erstellen Sie eine Sequenz von Listen mit Tonhöhe und Lautstärke.

b) Erstellen Sie eine Funktion, bei der wir mit einer Liste von zwei Zahlen, die den Start- und Endwert des x-Bereiches angeben, einen Ausschnitt berechnen können – also z. B. die Werte der Funktion $y = 3^x$ für den Bereich von $x = -2$ bis $x = 4$.

2.2.2.3 Appendix

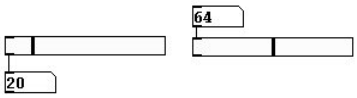
2.2.2.3.1 Symbol-Boxen

Analog zur Number-Box gibt es auch die Symbol-Box (wird aber nur selten in Pd verwendet). So kann z. B. „sel“ auch mit Symbolen verwendet werden:



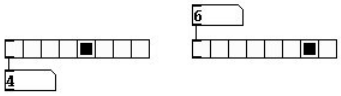
2.2.2.3.2 Slider

Auf der Kontroll-Ebene gibt es noch zwei weitere GUI-Objekte: den Slider und das Radio. Der Slider (**Put** # **HSlider** oder **VSlider** oder die Shortcuts davon) ist eine graphische Repräsentation einer Number-Box, allerdings auf einen Ausschnitt begrenzt, by default zwischen 0 und 127:



2.2.2.3.3 Radio

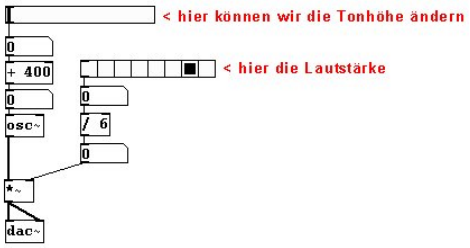
Das Radio (**Put** # **Hradio** oder **Vradio**) ist ebenfalls eine graphische Repräsentation einer Number-Box, aber stark vergröbert: Nur wenige Zahlen (by default von 0 bis 7) können durch Klick auf die verschiedenen Kästchen ausgeworfen werden.



Slider und Radio gibt es horizontal und vertikal; der Unterschied besteht allein im Aussehen.

2.2.2.3.4 Anwendung von Slider und Radio

Mit einem Slider sind verschiedene Tonhöhen anwählbar, mit einem Radio verschiedene Lautstärke-Stufen:



So hat man eine visuell gut verständliche Oberfläche, um Parameter eines Patches zu ändern. Das ist vor allem dann hilfreich, wenn man live auf der Bühne Elektronische Musik spielt.

2.2.2.4 Für besonders Interessierte: Alternative Typenbezeichnungen und mehr zu den Kästchen

Eine „float“-Spezifikation kann (z. B. bei „trigger“) in Pd häufig statt mit „f“ auch mit einer Zahl ausgedrückt werden (deren Wert manchmal eine Rolle spielen kann, jedoch nicht immer – z. B. gilt ihr Wert bei „f“ oder „pack“, aber nicht bei „t“):

Da jedoch darunter freilich die Klarheit leidet, ist die Verwendung einer Zahl in der Regel nicht ratsam.

Noch etwas Allgemeines zu den Kästchen: 1. Alle Kästchen sind strenggenommen Objekte, die Messages empfangen und senden können und auf diese Messages dann ihren Eigenschaften entsprechend reagieren. 2. Die Verbindungen geben an, welches Objekt an welche anderen Objekte Messages weitersendet. Dabei gilt, dass, wenn ein Objektausgang (Outlet) mit den Eingängen von mehreren anderen Objekten verbunden ist, alle diese Objekte diese Message erhalten. Die Reihenfolge ist dabei in pd (absichtsvoll) nicht definiert. 3. Es gibt GUI-Objekte, die durch Benutzerinteraktion diese Messages erzeugen und versenden. Beispiele für GUI Objekte: bang, toggle, slider und canvas.

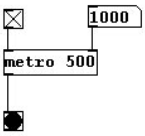
2.2.3 Zeitoperationen

Musik findet bekanntlich in der Zeit statt. Darum ist es für eine Audio-Programmiersprache natürlich wichtig, dass die zeitliche Abfolge gesteuert werden kann (also Dauern/Rhythmen und Abfolgen erzeugt werden können).

2.2.3.1 Theorie

2.2.3.1.1 Metro

Das erste elementare Objekt hierfür ist „metro“. Wie der Name andeutet, handelt es sich um ein Metronom. Wenn man es einschaltet (ein- aussschalten mit 1/0 im linken Eingang, also dem Toggle), erscheinen unten Bangs in dem zeitlichen Abstand, den man durch ein Argument bzw. im rechten Input angibt.



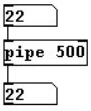
Das Tempo wird in **Millisekunden** (Abk. ms) angegeben, also in Tausendstelsekunden. Wenn also einmal pro Sekunde ein Bang kommen soll, geben wir an: „metro 1000“; soll alle zwei Sekunden ein Bang erfolgen, „metro 2000“, alle halbe Sekunde (also Viertel = 120) „metro 500“.

2.2.3.1.2 Delay

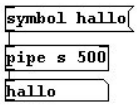
„delay“ (Abk. „del“) verzögert einen eingehenden Bang um so viele Millisekunden wie dem Argument bzw. rechten Input entsprechen:

2.2.3.1.3 Pipe

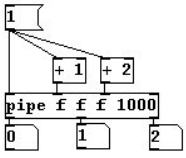
Dasselbe wie mit Zahlen und beliebigen Symbolen funktioniert auch mit „pipe“. Als Argument gibt man wiederum die Verzögerungsdauer an. By default erwartet „pipe“ als Input Zahlen.



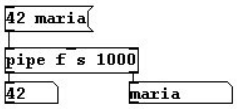
Will man ein Symbol durchschicken, muss dies als erstes Argument angegeben werden (mit „s“, wie bei Route). Als zweites (oder als rechten Input) benennt man die Dauer:



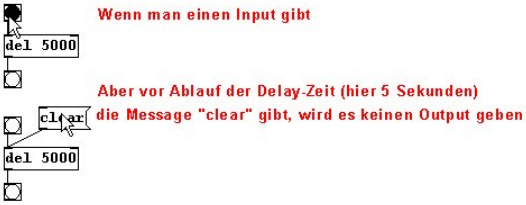
Außerdem kann „pipe“ wie „pack“/„unpack“ mehrere Eingänge und Ausgänge haben:



Listen handhabt „pipe“ wie „route“:

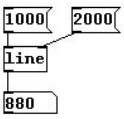


‚Wartet’ ein Input gerade in einem „del“ oder „pipe“, kann er mit der Message „clear“ oder „stop“auch gelöscht werden, ehe er wieder herauskommt:

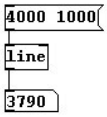


2.2.3.1.4 Line

Mit „line“ erstellen wir eine Zahlenreihe in der Zeit. Das heißt, wir können dem Programm anweisen, innerhalb einer bestimmten Zeitspanne von einer Startzahl zu einer Zielzahl zu zählen. „line“ erhält normalerweise kein Argument. Im rechten Input geben wir die Dauer für die Zahlenreihe an (by default 0). Links geben wir den Zielwert an (by default 0, das kann als Argument anders angegeben werden).

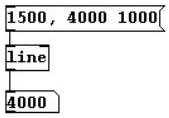


Gibt man nun links einen neuen Zielwert an, springt er direkt dorthin, das heißt, der Wert rechts wurde wieder auf 0 gestellt und muss wieder neu vergeben werden (das ist eine Ausnahme in Pd; normalerweise speichern Pd-Objekte ihre „kalten“ Eingänge bis sie neu gesetzt werden). Man kann aber auch beide Werte (Zielwert und Dauerwert) als Liste geben:



Klickt man nun wieder auf die Message-Box, passiert nichts, denn „line“ ist jetzt bei 4000 angekommen und verbleibt dort. Gibt man in der Liste einen neuen Zielwert an, z. B. 50, macht „line“ nun eine Zählung von 4000 zu 50 (in 1000 Millisekunden).

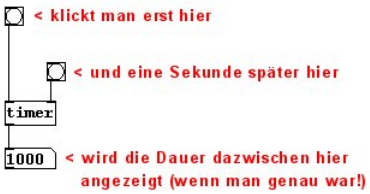
Will man bei einer bestimmten Zahl beginnen und in einer bestimmen Zeitspanne zu einer anderen Zahl gehen, muss man also zuerst links (ohne zuvor rechts einen Wert gegeben zu haben) zum Anfangswert springen (mit einer einzelnen Message-Box) und dann die Liste geben. Wie am Ende von Abschnitt 2.2.2.1.2 schon gesagt wurde, kann man aber mehrere Messages in einer Message-Box unterbringen, getrennt jeweils durch ein Komma. Dann sieht es so aus:



In diesem Beispiel erfolgt also jedes Mal, wenn man auf die Message-Box klickt, eine Zählung von 1500 bis 4000 in 1000 Millisekunden.

2.2.3.1.5 Timer

„timer“ ist eine Art Stoppuhr. Bei beiden Eingängen gibt man Bangs. Es wird immer der zeitliche Abstand zwischen dem linken Bang (den man daher zuerst gibt) und dem rechten gemessen (in Millisekunden):



Hier kann man auch sehen, dass Trigger-Operationen für den Computer keine zeitliche Ausdehnung haben, auch wenn sie klar nacheinander abfolgen:

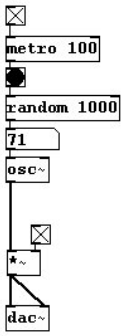
(so auch unter 2.2.2.2.5)

„timer“ ist eine (etwas unnötige) Ausnahme von der Pd-Regel, dass Inputs immer von rechts nach links erfolgen sollen.

2.2.3.2 Anwendungen

2.2.3.2.1 Automatische Zufallsmelodie

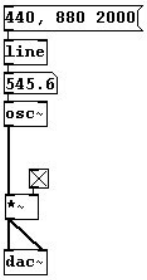
Nun können wir schon komplexere musikalische Anordnungen realisieren. Zum Beispiel eine schnelle Zufallsmelodie, die automatisch abläuft:



Probieren Sie im obigen Beispiel einmal verschiedene Geschwindigkeiten des Metronoms aus (als rechter Input in „metro“ eine Number-Box)!

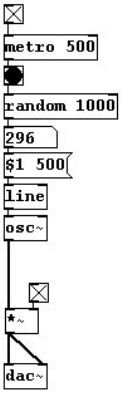
2.2.3.2.2 Glissando

Wir können auch ein Glissando erstellen:



2.2.3.2.3 Glissando-Melodie

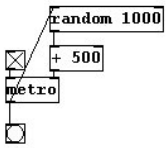
Oder beides kombinieren und eine zufällige Glissando-Melodie erstellen:



2.2.3.2.4 Unregelmäßige Zufallsrhythmen

Wir können auch unregelmäßige Rhythmen erzeugen, ebenfalls auf Zufallsbasis:

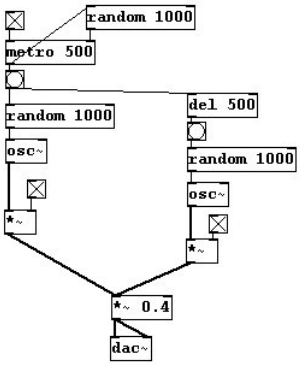
In diesem Beispiel kommen aus dem „metro“ Bangs im Abstand zwischen 0 und 999 Millisekunden (per Zufall). Will man z. B. Dauern im Abstand zwischen 500 und 1500 Millisekunden haben, muss man eine grundsätzliche Addition hinzufügen:



Einen solchen grundsätzlichen Zusatz (hier „+ 500“) zu einer Rechnung nennt man „Offset“.

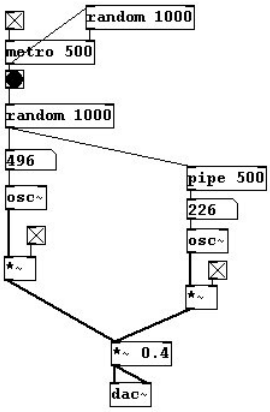
2.2.3.2.5 Kanons

Diese Rhythmen können wir wieder an den Zugallsgenerator anschließen und kanonisch auf einen zweiten Oszillator übertragen:



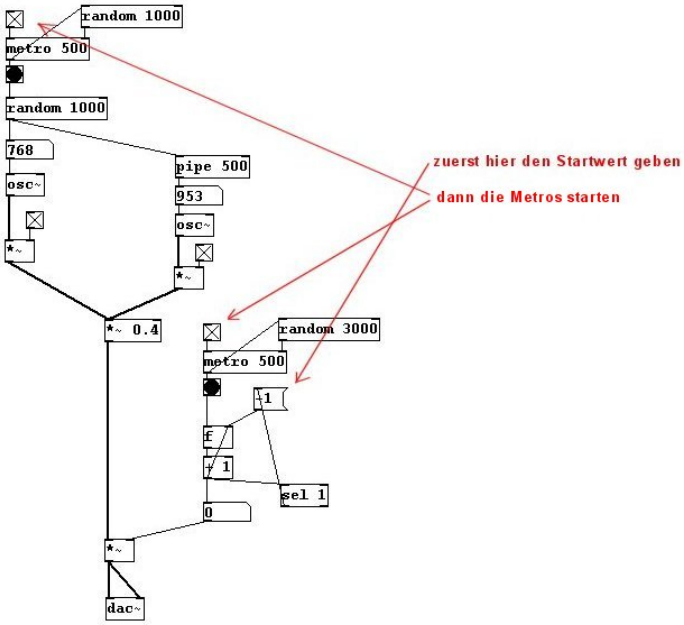
(Die „*~ 0.4“ wird später erklärt.)

Oder gleich einen richtigen Kanon erstellen:



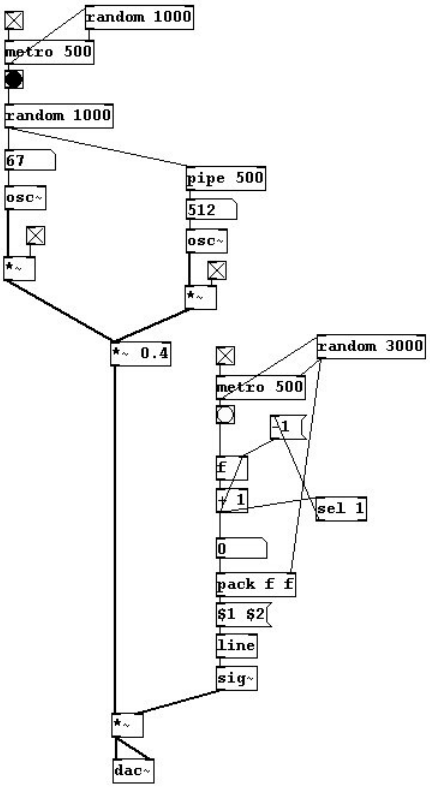
2.2.3.2.6 Pausen

Wir können auch automatisierte Pausen einfügen:



2.2.3.2.7 Crescendo/Decrescendo

Oder als Crescendi und Decrescendi („sig~“ wird ebenfalls später erklärt):



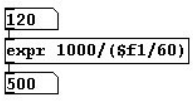
Hier wird wieder deutlich: Für Pd gibt es zunächst nur Rechenoperationen mit Zahlen. Ein Crescendo ist ebenso eine Zahlenreihe wie ein Glissando. Man könnte auch sagen: Ein Crescendo ist ein Glissando der Lautstärke.

2.2.3.2.8 Metronom

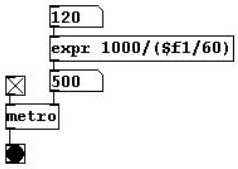
So können wir uns schon ein Metronom bauen:

Bauen wir es zuerst nur optisch, so dass als Metronomsignal ein Bang ausgegeben wird. Metronomangaben erfolgen als Schläge pro Minute, so wie in Partituren verzeichnet ist: Viertel = 60, Viertel = 100 etc.

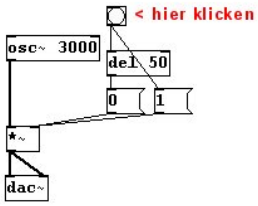
Anfangs müssen wir also bpm (engl. Abk. für beats per minute = Schläge pro Minute) in Millisekunden umrechnen:



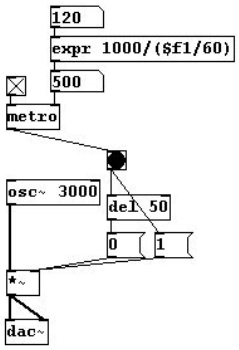
Das Ergebnis setzen wir als Zahlen für ein Metronom.



Nun wollen wir aber die Impulse nicht nur am Bang sehen, sondern auch hören. Nehmen wir wieder den Klang-Patch von vorhin und stellen einfach ein, dass ein kurzer Ton bei jedem Bang ausgegeben werden soll. Den kurzen Ton erstellen wir folgendermaßen:



Komplett:



In Verbindung haben wir nun ein Metronom gebaut. Später werden wir noch ein alternatives Klangsignal integrieren.

2.2.3.2.9 Weitere Aufgabenstellungen

a) Erstellen Sie eine Zufallsmelodie, die alle halbe Sekunde zum nächsten Ton springt (alternativ: glissandiert).

b) Erstellen Sie ein Metronom mit unregelmäßigen Zufallsrhythmen (durchschnittliches Tempo verstellbar).

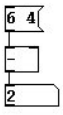
c) Erstellen Sie ein Metronom, das immer abwechselnd fünf Schläge im Tempo Viertel = 60 und fünf Schläge im Tempo Viertel = 100 macht.

d) Erstellen Sie eine Zufallsmelodie, die im Wechsel von zwei Sekunden eher hoch oder eher tief ist.

2.2.3.3 Appendix

2.2.3.3.1 Verteilung von Listen

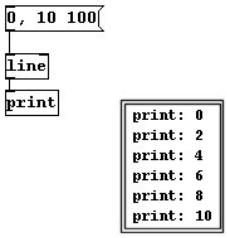
Wie bei „line“ gesehen, kann man in Pd grundsätzlich einem Objekt mit mehreren Eingängen ganz links eine Liste geben, statt in jeden Eingang etwas einzugeben (allerdings gibt es Objekte, die davon ausgenommen sind). Die Elemente der Liste werden dann von rechts nach links auf die Eingänge verteilt:



2.2.3.3.2 Die zeitliche Auflösung von Kontrolldaten

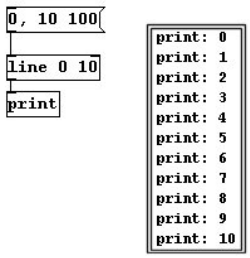
Die zeitliche Auflösung von Arbeiten in der Kontroll-Ebene ist in Millisekunden.

Dies ist allerdings häufig nicht so voreingestellt; wie man sich denken kann, erfordert eine Rechnung in Millisekunden schon eine sehr hohe Leistung vom Prozessor (auch CPU* genannt). Bei „line“ beispielsweise ist voreingestellt, dass die Schritte im 20-Millisekundenabstand erfolgen:



Wenn man also in 100 Millisekunden von 0 bis 10 zählen will, macht der Computer alle 20 Millisekunden einen Schritt; im Output erscheinen daher nur die Zahlen im Abstand von zwei Schritten.

Dieses Intervall (in Millisekunden) lässt sich aber bei „line“ verstellen, als zweites Argument (das erste gibt den primären Zielwert der Zählung an, der dann ja allerdings durch den Input aufgehoben ist):



Zu bedenken ist allerdings, dass das Ergebnis nur solange „sauber“ ist, solange der Prozessor des Computers leistungsstark genug ist. Andernfalls kommt es zu Fehlern.

*CPU = central processing unit, zu deutsch „Hauptprozessor“ oder „Zentraleinheit“. Daneben gibt es häufig noch weitere Prozessoren, z. B. in der Grafikkarte, wo speziell grafische Operationen berechnet werden.

2.2.4 Sonstiges

Um den Umgang mit Pd noch zu verbessern, gibt es einige weitere Optionen.

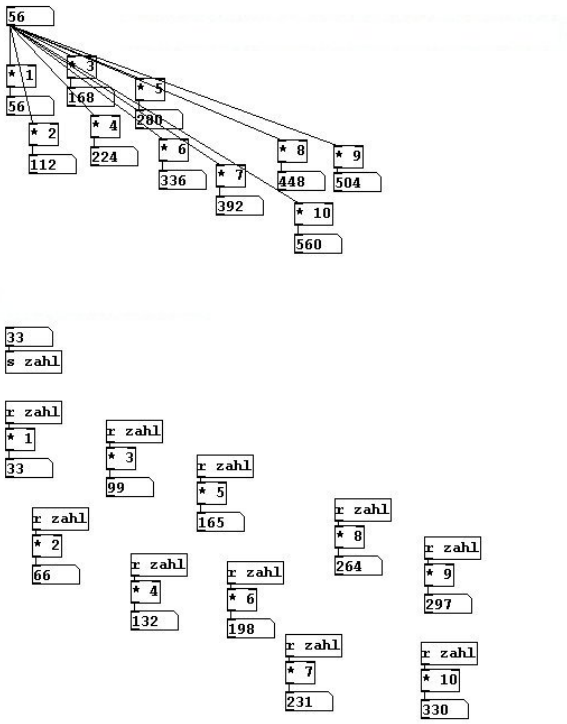
2.2.4.1 Senden und Empfangen

2.2.4.1.1 Send/Receive

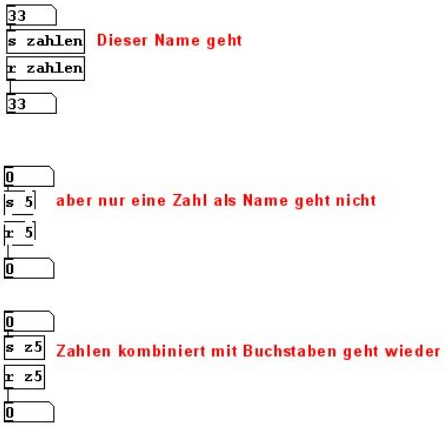
Um nicht alle Boxen immer mit ‚Kabeln’ verbinden zu müssen, gibt es die Möglichkeit, etwas zu „senden“ und zu „empfangen“. Dies erfolgt mit den Objekten „send“ und „receive“.

„send“ erhält als Argument einen beliebigen Namen. Ein „receive“, das dann als Argument denselben Namen hat, erhält den Input des „Send“ und gibt ihn aus.

„send“/„receive“ (Abk. „s“ und „r“) sind praktisch, wenn man z. B. eine Zahleninformation an vielen verschiedenen Orten braucht (worunter allerdings die Klarheit des Patches leidet).

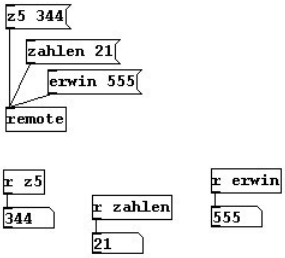


Solche frei wählbaren Namen (wir kommen darauf später zurück) müssen immer zusammenhängende Buchstabenfolgen ohne Leerzeichen sein; einzelne Zahlen sind nicht zulässig.

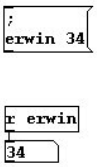


2.2.4.1.2 Versand mit Listen

Hat man verschiedene Receiver (mit verschiedenen Namen), kann man von einer zentralen „Verteilungsstelle“ Nachrichten mit „remote“ zuweisen (vergleichbar „route“). Dem erteilt man eine Liste, deren erstes Element der Name des Empfängers ist und deren zweites Element die Nachricht selbst. Remote ist Teil von pd extended.

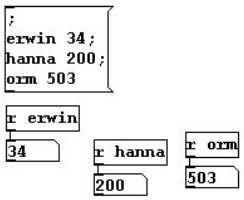


Eine ähnliche Möglichkeit ist, einer solchen Liste in einer Messagebox ein Semikolon voranzustellen; dann braucht man nur auf die Message-Box zu klicken, und die Nachricht wird verschickt.



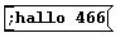
2.2.4.1.3 Eine Reihe von Versand-Listen

So kann man auch viele verschiedene Nachrichten in einer Messagebox (mit einem einzigen Klick) verschicken:

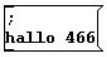


Zwei Interpunktionszeichen innerhalb von Messages haben also eine besondere Bedeutung: Kommata (eine Reihe von mehreren Messages) und Semikola (die den Sender bedeuten).

Man beachte: Pd macht in Message-Boxen automatisch nach einem Semikolon einen Zeilenwechsel. Wenn wir also Folgendes schreiben:



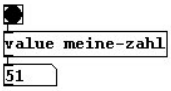
und die Eingabe kopieren (**Ctrl-D**) oder den Patch schließen und neu öffnen, sieht das Ergebnis dann so aus:



Ebenso erfolgt in einem Comment (2.1.4.5.) nach einem Semikolon automatisch ein Zeilenumbruch.

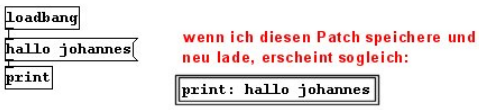
2.2.4.1.4 Value

Eine weitere Möglichkeit, einen Wert zu verschicken, ist, ihn global, das heißt für den ganzen Patch festzulegen. Dies funktioniert mit „value“. Man gibt als Argument einen beliebigen Namen und als Input den Wert ein. An anderen Stellen im Patch kann man dann den Wert mit demselben Objekt und demselben Argument durch einen Bang erhalten:



2.2.4.2 Loadbang

Manche Werte möchten wir vielleicht gerne erhalten, bis wir den Patch das nächste Mal öffnen oder ein Wert soll gleich zu Beginn ge-bangt werden. Dafür gibt es „loadbang“ (schickt gleich beim Öffnen des Patches einen Bang) und „init“ (Abk. „ii“), das als Argument eine Zahl oder ein Symbol (oder eine Liste aus Zahlen, Symbolen oder Zahlen und Symbolen) ausgibt (pd extended).



2.2.4.3 GUI-Optionen

GUI steht für „graphical user interface“, also alle besonderen grafischen Objekte in Pd. GUI-Objekte sind Number- und Symbol-Box, Bang, Toggle, Slider, Radio, Canvas, sowie – zu denen kommen wir später – Array und VU. Alle GUI-Objekte haben noch erweiterte Funktionen. Zu denen gelangt man, wenn man mit der rechten Maustaste auf das Objekt klickt und im so erscheinenden Pulldown-Menü mit der linken Maustaste „Properties“ anwählt.

Hier lässt sich weiterhin Folgendes einstellen:

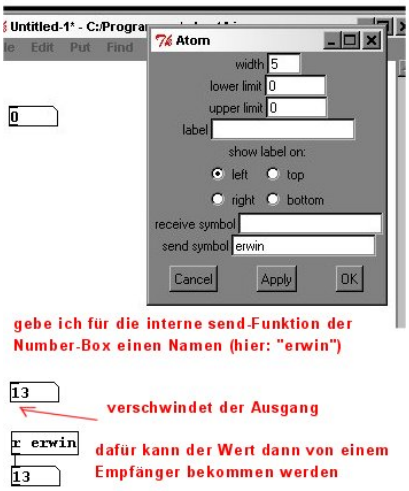
2.2.4.3.1 Number- und Symbol-Box

width entspricht der Größe der Box. Diese Angabe ist z. B. bei großen Zahlen bzw. Zahlen mit vielen Kommastellen erforderlich.

Mit *lower und upper limit* lässt sich der Bereich einstellen, der in die Box eingegeben werden kann (bei Zahlen z. B. ein Rahmen zwischen 0 und 1000).

Mit *label* kann man der Box einen angezeigten Namen geben (darunter stellen wir ein, wo der Name an der Box angezeigt werden soll).

Mit *receive/send* kann man in die Box eine send- und receive-Funktion einbauen. Wenn man z. B. in send „post1“ eingibt, und irgendwo einen Receiver „post1“ hat, erhält dieser alle Eingaben in die Number-Box. Entsprechendes gilt für die „receive“-Funktion.



Wie man in der Grafik sehen kann, verschwindet das Input- bzw. Output-Kästchen, wenn man die interne send- bzw. receive-Funktion aktiviert.

Die Änderungen werden wirksam, wenn wir unten auf „apply“ oder auf „ok“ klicken.

2.2.4.3.2 Bang

size bezeichnet die veränderbare Größe (in Pixeln).

intrrpt/hold besagt, wie lange der Bang aufleuchtet (in Millisekunden).

init bedeutet, dass der Wert (hier Bang) beim Öffnen des Patches gleich ausgegeben (wie bei Loadbang) wird.

Das *send-Symbol / receive-Symbol* ist wie in Number-Box eine interne „send-“ / „receive“-Funktion.

Mit *name* kann man wie in Number-Box ein „label“ erstellen. Hier wird die Position mit x- und y-Werten bestimmt. Zusätzlich kann man auch die Schriftart und die Schriftgröße einstellen, außerdem die Farben für Hintergrund, Vordergrund und Name ändern. Zuerst klicken wir in folgener Reihe an, was geändert werden soll, ...

... und anschließend auf eine vorgegebene Farbe ...

Unter „compose color“ lassen sich die Farben auch individuell generieren.

Mit „backgd“ ist der Hintergrund gemeint, also die ganze Fläche des Bangs, mit „front“ der Vordergrund, also die Farbe, in der bei Aktivierung (durch einen Input oder durch Anklicken) der Bang kurz aufleuchtet.

Alle Änderungen werden wiederum mit „Apply“ oder „Ok“ aktiv.

2.2.4.3.3 Toggle

Hier funktioniert alles analog zum Bang, bis auf *value*: Der Toggle wechselt ja standardmäßig immer zwischen 0 und 1. Hier kann man aber einstellen, dass statt der 0 (die 1 bleibt immer eine 1) ein anderer Wert kommt.

2.2.4.3.4 Slider

width: Breite (in Pixeln)

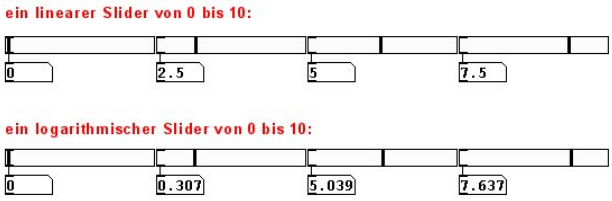
height: Höhe (in Pixeln)

bottom: Wert des Sliders, wenn der Zeiger ganz unten steht

top: Wert des Sliders, wenn der Zeiger ganz oben steht

lin: Entweder lineare oder logarithmische Zunahme /Abnahme innerhalb des Rahmens. Wenn man draufklickt, erscheint auf dem Feld „log“. Es gilt immer das, was gerade zu lesen ist.

Man beachte: Bei „log“ kann nicht „0“ als Eckwert stehen.



init: Der untere Wert des Rahmens wird gleich beim Öffnen des Patches ausgegeben.

steady on click: Der Zeiger wird durch Bewegung bei gehaltener Maustaste verschoben. Klickt man auf „steady on click“, erscheint „jump on click“, darauf springt der Zeiger direkt dorthin, wo man innerhalb des Sliders hinklickt.

Der Rest verhält sich wieder analog zum Bang.

2.2.4.3.5 Radio

Bei Radio funktioniert alles wie bei den anderen, bis auf *number*, die Anzahl der Kästchen.

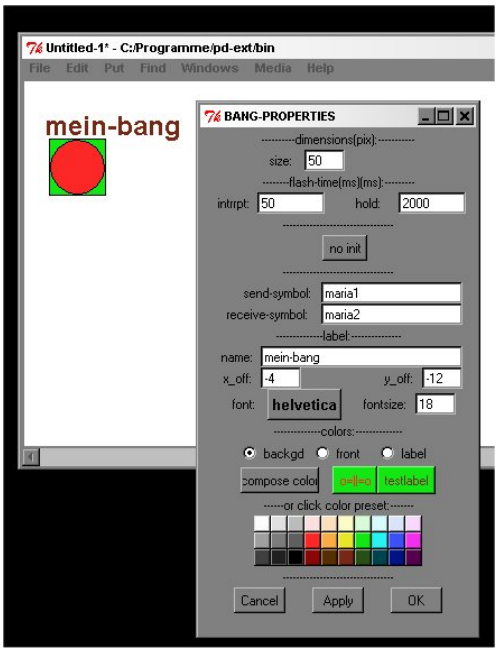
2.2.4.3.6 Canvas

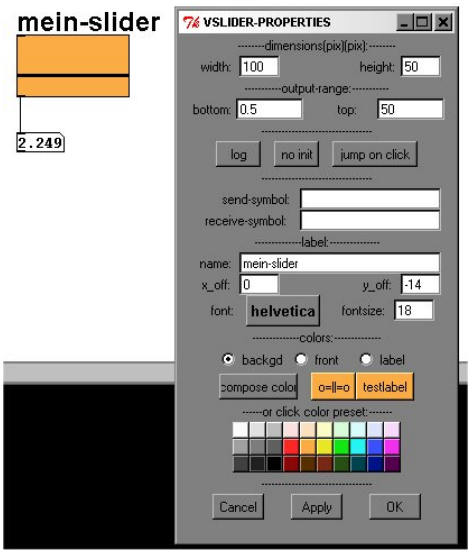
Am Ende von 2.1.2. hieß es, die weiße Fläche, auf die wir Objekte setzen, heißt „canvas“. Wir können aber noch weitere solcher Farbflächen hinzufügen (**Put # canvas**). Sie haben keine Funktion außer der, farbige Flächen zu sein.

Die Fläche entält links oben ein blaues Quadrat – das ist das eigentliche Objekt. Die ganze Fläche ist das Produkt dieses Objekts, sozusagen dessen Output. Sie bedeckt alle davor erstellen Elemente des Patches und liegt unter allen danach erstellen.

Die Properties sind gleicher Art wie die der anderen GUI-Objekte.

2.2.4.3.7 Beispiele für umgestaltete GUI-Objekte



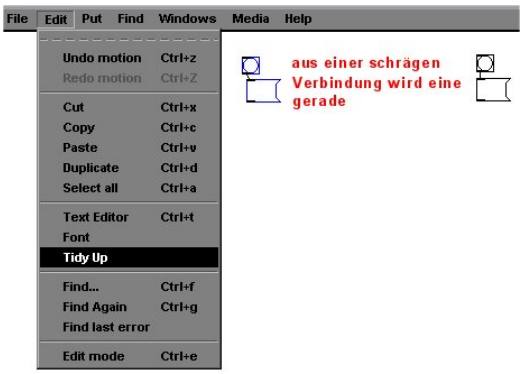


2.2.4.3.8 Fontgröße ändern

Unter **Edit # Font** kann man die allgemeine Größe der Boxen ändern.

2.2.4.3.9 Tidy up

Schräge Verbindungen können selektiert und mit **Edit # Tidy up** gerade gezogen werden. Das funktioniert allerdings häufig nicht.

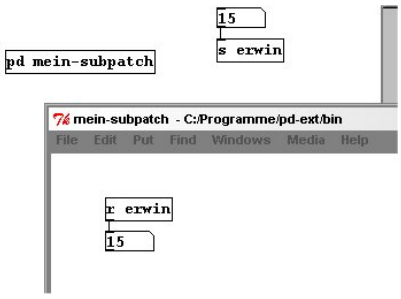


2.2.4.4 Subpatches

2.2.4.4.1 Platz

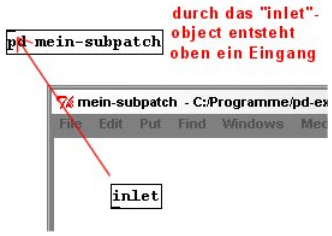
Im Laufe des Programmierens wird der Platz eines Fensters irgendwann zu klein. Daher kann man Teile eines Patches in sogenannte „Sub-Patches“ auslagern. Wenn wir ein Objekt „pd“ und als Argument einen beliebigen Namen schreiben, z. B. „pd mein-subpatch“ (der Name darf kein Leerzeichen enthalten), öffnet sich ein neues Fenster. (Wenn dieses geschlossen wird, kann es später wieder geöffnet werden, indem man im Execute-Mode einmal auf das Objekt „pd mein-subpatch“ klickt). Hier hat man nun Platz für neue Teile des Patches.

Es gibt zwei Verbindungsmöglichkeiten zum darüberliegenden ersten Fenster des Patches: „send“ und „receive“ funktionieren über verschiedene Fenster hinweg genauso wie innerhalb eines Fensters:

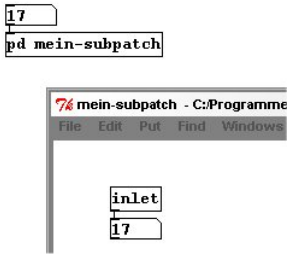


Hieran sieht man auch, dass man unabhängig voneinander in verschiedenen Fenstern eines Patches zwischen Execute-Mode und Edit-Mode hin und her wechseln kann.

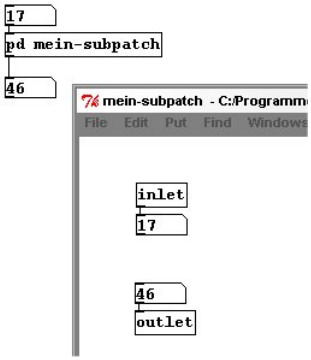
Die andere Möglichkeit der Verbindung besteht über die Objekte Inlet und Outlet. Platziert man im Unterfenster einen Inlet, erscheint im Objekt für das Unterfenster („pd mein-subpatch“) ein Eingang.



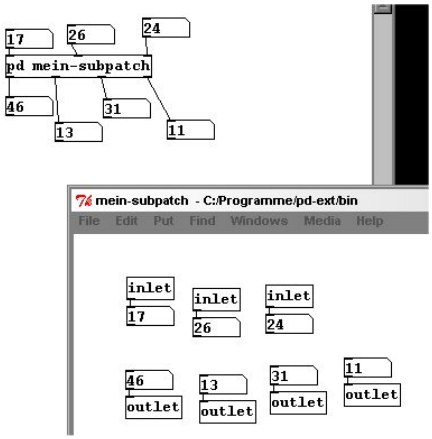
Schickt man nun im Hauptfenster in diesen Eingang z. B. eine Zahl, erscheint diese im Unter-Patch.



Umgekehrt verfährt man mit Outlet:



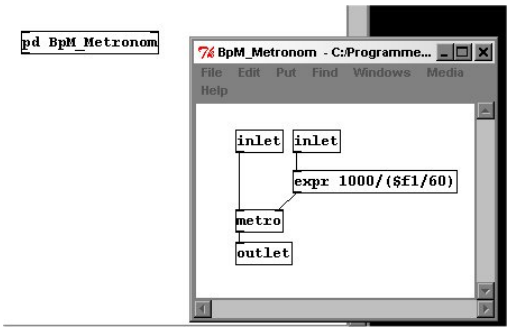
Mehrere Inlets bzw. Outlets werden nebeneinander gesetzt und so erscheinen die Inlet- / Outlet-Kästchen dann gleichfalls nebeneinander angeordnet:

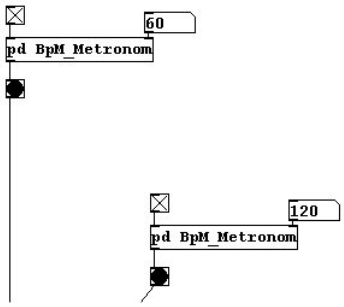


Das Fenster eines Subpatches kann auch geschlossen werden und arbeitet trotzdem weiter, solange das Fenster des Hauptpatches nicht geschlossen wird.

2.2.4.4.2 Modularisierung

Subpatches beheben nicht nur Platzmangel, sondern dienen auch der Strukturierung des Patches. Das heißt, dass man Teile, die eine bestimmte Aufgabe erledigen, in einen eigenen Subpatch auslagert, so dass diese kleine 'Maschine' immer verfügbar ist. So einen Teil nennt man auch „Modul“. Als Beispiel sei hier ein Metronom angeführt, in das man statt Millisekunden-Werte BpM-Werte eingeben kann:





Selbstgewählte Namen können keine Leerzeichen enthalten, man ersetzt sie daher meist durch Bindestriche oder (wie in diesem Beispiel) mit Unterstrichen.

Kapitel 3. Audio

3.1 Grundlagen

Von nun an sind die Beispielpatches nicht mehr mit nachträglichen roten Erklärungen kommentiert, sondern mit den regulären Pd-internen Comments (**Put # Comment**), so wie Sie es selbst später auch praktizieren sollten.

Viele der Funktionen, die in Kapitel 2 gezeigt wurden, werden im Folgenden nicht verwendet; das betrifft alle grafischen Elemente und Funktionen wie „send“ und „receive“, obwohl sie freilich in der Praxis immer gebraucht werden. Die hier vorgestellten Patches sind jedoch immer auf das Wesentliche reduziert. Sobald die hier vorgestellten Techniken aber Bestand-TEIL eines Stückes sind, ist es erforderlich, Teile in Subpatches zu packen, sie mit „Inlets“/„Outlets“ zu verbinden etc. Unter Abschnitt 3.4.2.4 wird anhand eines Beispiels gezeigt, wie das aussehen kann.

Fortan sind größere Patches bereits fertig gebaut als extra Dateien vorhanden (http://www.kreidler-net.de/pd/patches/patches.zip).

3.1.1 Tonhöhe

Erinnern wir uns wieder an unser allererstes Beispiel. Gehen wir dem nach, was wir gehört haben: einen Ton mit 440 Hertz (später auch mit anderen Frequenzen, also anderen Tonhöhen), den wir ein- und ausschalten konnten, das heißt, laut oder unhörbar leise stellen.

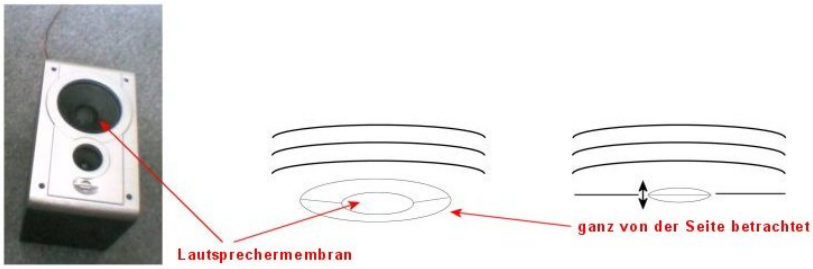
Zunächst zur Tonhöhe: In Pd arbeiten wir mit zwei Arten von Tonhöhen – entweder in Hertz oder in MIDI-Zahlen. Die traditionelle Bezeichnung a, b, gis etc. gibt es in Pd gar nicht. Stattdessen haben alle chromatischen Töne MIDI-Nummern, z. B. entspricht a' der Nummer 69, b' 70 etc. Die andere Bestimmung der Tonhöhe erfolgt in Hertz. Um das zu verstehen, müssen wir nun in die Lehre der Akustik eintauchen.

3.1.1.1 Theorie

3.1.1.1.1 Digitale Steuerung des Lautsprechers

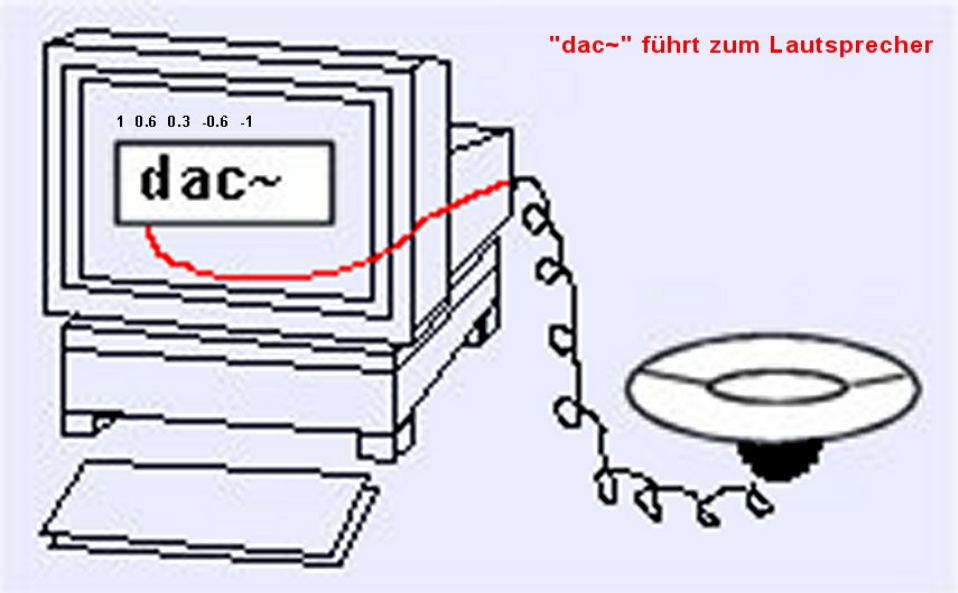
Klang ist Luft, die schwingt. Traditionelle Instrumente sind dazu da, die Luft in bestimmte Schwingungen zu versetzen. Das geht zum Beispiel mit Saiten (Geige), Lippen (Trompete) oder Membranen (Pauke). Auch in unseren Ohren haben wir Membrane, nämlich das Trommelfell, das mit den Schwingungen der Luft mitschwingt. Unser Gehirn transformiert diese Schwingungen dann zu dem, was wir als Klang wahrnehmen.

Bei Elektronischer Musik verwenden wir zur Klangerzeugung Lautsprecher. Diese haben auch eine Membran (oder mehrere), die vor und zurück schwingt und dadurch die Luft zum Schwingen anregt.

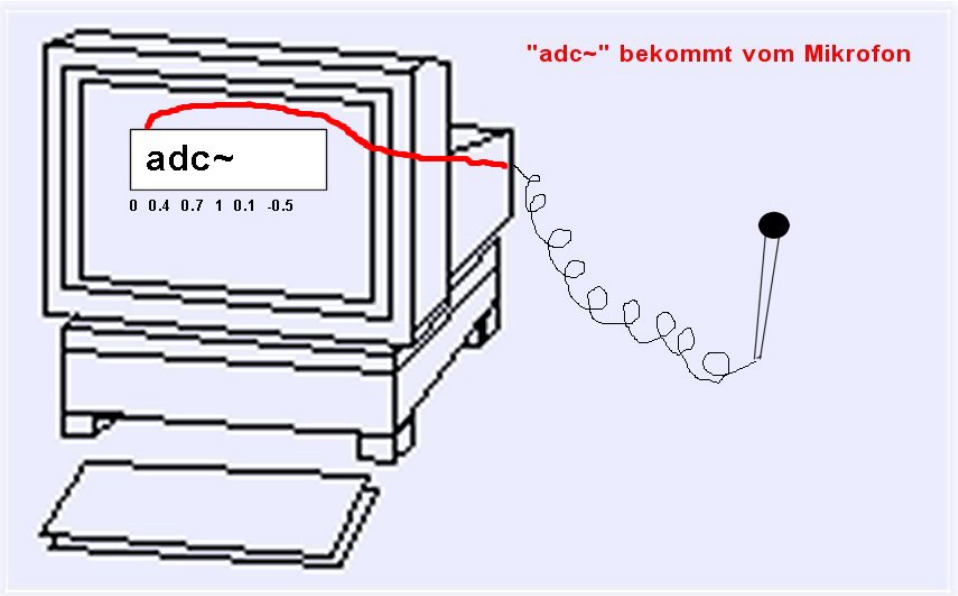


Diese Membran wird jetzt vom Computer aus gesteuert. In Pd übernimmt dies das Objekt „dac~“ (digital audio converter). Damit ist Folgendes gemeint: Klang ist als physikalisches Phänomen,

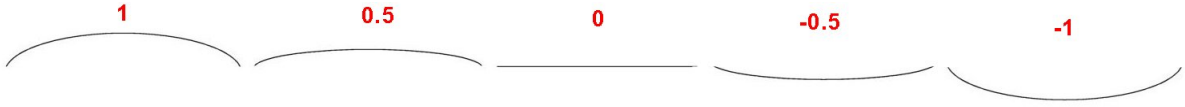
nämlich als Schwingung der Luft, analog. **Der Computer selbst arbeitet aber nur mit Zahlen**, also digital. Das Objekt „dac~“ macht Zahlen zu Klang, indem es die Zahlen in Stromschwankungen konvertiert, die - entsprechend verstärkt - im Lautsprecher zu Bewegungen der Lautsprechermembran führen.



Umgekehrt kann man ein Mikrofon an den Computer anschließen. Ein Mikrofon besitzt ebenfalls eine Membran, die durch Luftschwingungen angeregt wird und diese Schwingungen in Form von Stromschwankungen an den Computer weitergibt, der diese Stromschwankungen in Zahlenfolgen umwandelt. In Pd kann dieser Input mit dem Objekt „adc~“ empfangen werden.



Zunächst aber zurück zum Lautsprecher. Dessen Membran kann sich hin- und herbewegen. Die äußerste Position ist für den Computer die Position 1. Die hinterste Position ist für den Computer die Position -1. Wenn die Membran genau in der Mitte, also in Ruhe ist, ist sie in der Position 0. Alle anderen Positionen dazwischen sind Werte zwischen -1 und 1.



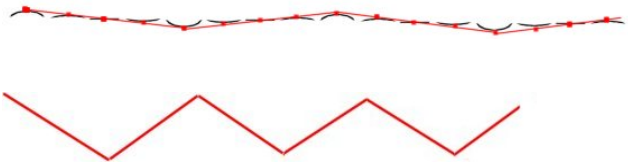
In der Realität sind diese Bewegungen so klein und schnell, dass wir sie mit bloßem Auge kaum nachvollziehen können.

3.1.1.1.2 Wellen

Stellen wir uns nun vor, dass sich die Membran immer im gleichen Tempo zwischen den äußeren Grenzen hin- und herbewegt:

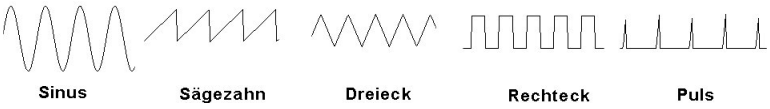
Markieren wir nun die einzelnen Stadien:

Abstrahiert, als Schaubild mit y-Achse (Stand der Membran) und x-Achse (Zeit) sieht das dann so aus:



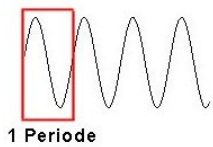
Auf dieser Darstellung sieht man sehr gut die *Schwingungsform*, ein Dreieck.

So gibt es verschiedene Schwingungsformen, je nach dem, wie die Bewegung der Membran verläuft. Ihre Namen erhält sie von ihrem Aussehen:

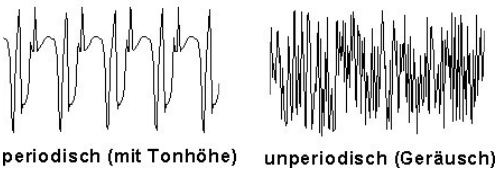


Das Pd-Objekt „osc~“ erzeugt eine Sinus-Schwingung.

Das Wichtige bei diesen Schwingungen ist: Sie wiederholen sich dauernd, ohne Änderung ihrer Bewegungscharakteristik. Eine solche Schwingung nennen wir *periodisch*, da sie periodisch immer wiederkehrt. Eine Periode ist ein vollständiger Verlauf einer Schwingung, der ständig wiederholt wird. Physikalisch spricht man von einer *Welle*.

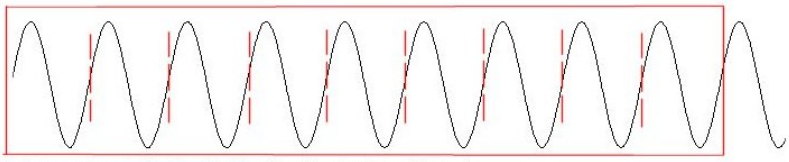


Das für uns Besondere an periodischen Schwingungen ist, dass wir solche Schwingungen als Töne mit bestimmten Tonhöhen hören. Dahingegegen sind Geräusche Schwingungen, die unperiodisch sind.



3.1.1.1.3 Messung

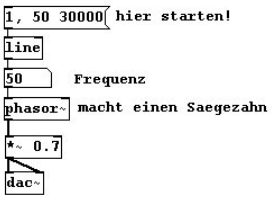
Befassen wir uns aber zunächst nur mit periodischen Schwingungen. Bei ihnen kann man nun zählen, wie oft die Periode in einer Sekunde auftritt. Diese Zahl ist die Frequenz einer Schwingung, ihre Einheit heißt „Hertz“ (Abk. Hz); Frequenz bedeutet immer, wie oft etwas pro Sekunde wiederholt wird (mathematisch ausgedrückt: 1/Sekunde).



1 Sekunde: darin 9 vollständige Perioden = 9 Hertz

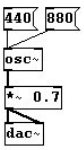
Von der Frequenz hängt die Tonhöhe ab. Der Kammerton a' bedeutet, dass die Luft periodisch 440 Mal pro Sekunde schwingt, c'' hingegen 523 Mal pro Sekunde und das tiefe G des Cellos ungefähr 100 Mal pro Sekunde.

Daran sehen wir schon: Je langsamer die Frequenz, desto tiefer der Tonhöheneindruck für unser Ohr. Tatsächlich hört ein Mensch – je nach Alter – Töne zwischen 20 Hz und 15000 Hz. Kinder können bis zu 20000 Hz hören, alte Menschen hingegen oft nur noch bis 10000 Hz. Hunde oder Fledermäuse hören noch weit über 20000 Hz hinaus. Diesen Bereich nennt man Ultraschall. Im Gegensatz dazu ist Infraschall unterhalb der menschlichen Hörgrenze angesiedelt, also zwischen 0 und 20 Hz, Infraschall. Diesen Bereich nehmen wir als Rhythmus wahr. Das kann an diesem Experiment in Pd schön beobachtet werden:

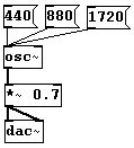


Anfangs hört man einen Rhythmus von Klicks (das ist der Klang eines Sägezahns), der allmählich schneller wird. Ab einer bestimmten Geschwindigkeit (über 20 Klicks pro Sekunde) schlägt die Wahrnehmung in einen Tonhöheneindruck um. Für die Luft (und für den Computer) ist das alles weiterhin immer nur ein „Rhythmus“. Aber für das menschliche Ohr ist es ab einer bestimmten Geschwindigkeit (ca. 20 Hz) eine Tonhöhe! Je schneller dieser Rhythmus wird, desto höher die Tonhöhe für uns.

Das Charakteristische für unser menschliches Ohr ist weiterhin, dass wir Tonhöhen *logarithmisch* hören. Gemeint ist damit: Einer Verdopplung der Frequenz entspricht für unser Ohr der Eindruck eines Oktavsprungs. Wechseln wir von einem Ton mit der Frequenz 440 zu einem mit der Frequenz 880, hören wir zunächst a' und dann a'' :

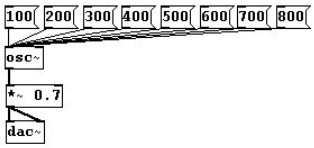


Wollen wir nun aber von 880 Hz aus wieder eine Oktav hören, müssen wir + 880 nehmen = 1760:

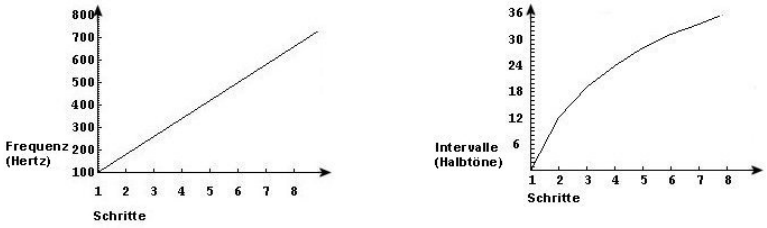


So wird klar: Von 30 Hz zu 60 Hz hören wir eine Oktav, aber von 1030 Hz zu 1060 Hz nur einen sehr kleinen Schritt. Tatsächlich ist der Sprung von 10000 Hz zu 20000 Hz nur der einer Oktav!

Anders, wenn wir zu einer Grundfrequenz immer denselben Betrag hinzuaddieren, z. B zu 100 Hz, was ungefähr der G-Saite des Cellos entspricht, zu der wir immer weiter 100 Hz addieren:

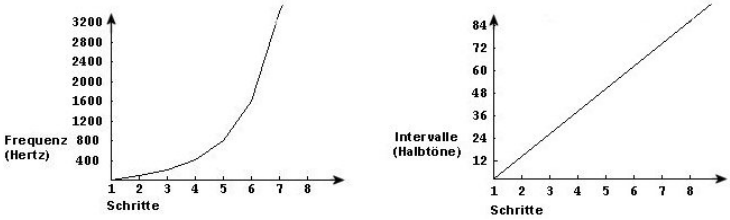


Was wir dann hören, ist von 100 zu 200 Hz eine Oktav, von 200 zu 300 Hz eine Quint, von 300 zu 400 Hz eine Quart etc. Was in Mathematik eine additive Reihe ist, also die Zunahme um immer das gleiche Intervall, ist für unser Ohr eine sukzessive Zunahme um ein immer kleiner werdendes Intervall:



Die linke Darstellung zeigt einen linearen Verlauf – den mathematischen. Die rechte Seite zeigt, was wir hören – einen logarithmischem Verlauf.

Wollen wir umgekehrt einen linearen Verlauf hören – also immer die Zunahme um das selbe Intervall, zum Beispiel der Oktav – müssen wir mathematisch einen exponentiellen Verlauf erstellen:



Die Umrechnung von linearem zu logarithmischem Verlauf findet in Pd zwischen MIDI-Nummern und Frequenz statt. MIDI-Nummern entsprechen unserer Hörweise, indem sie für gleich große Schritte immer gleich große Zahlenschritte einsetzen – das ist pro Halbton eine ganze Zahl. Man kann in Pd zwischen Frequenz- und MIDI-Angabe umrechnen:

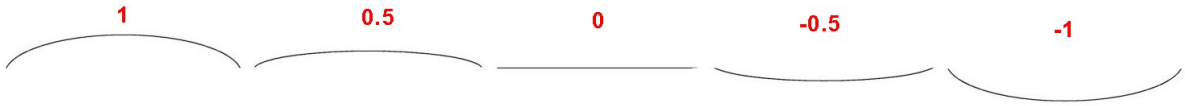
Eine kleine Tabelle von MIDI-Zahlen, Frequenzzahlen und ihre traditionelle Bezeichnung:

Frequenz (Hz)	MIDI	Tonname
65.4	36	C
100	43	G
130.8	48	c
261.6	60	c'
277.1	61	cis'
293.6	62	d'
311.1	63	dis'
329.6	64	e'
349.2	65	f'
370	66	fis'
392	67	g'
415.3	68	gis'
440	69	a'
466.1	70	b'
493.8	71	h'
523.2	72	c"
1000	83	h"
4186	108	c"''

Die regulären Oszillatoren wie „osc~“ oder „phasor~“ müssen als Input jedoch immer Angaben in Hertz erhalten.

3.1.1.1.4 Samplerate

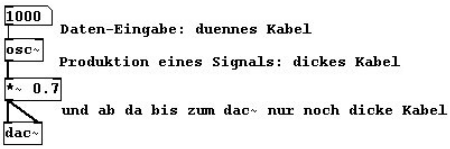
Wir müssen uns wieder bewusst machen: Für Pd ist Klang nur Zahlen. D. h., die Membran-Positionen des Lautsprechers sind Zahlen zwischen -1 und 1.



Objekte wie „osc~“ erzeugen eine sehr schnelle Abfolge von Zahlen zwischen -1 und 1, was „dac~“ dann zum Lautsprecher schickt. Genau gesagt werden 44100 Zahlen pro Sekunde erzeugt. Wenn der Lautsprecher sich also in einer Sekunde von -1 zu 1 bewegt, macht er genau genommen 44100 sehr kleine Schritte von -1 hin zu 1. Diese Zahl, 44100, heißt Samplerate.

Jeder Klang in Pd wird durch Zahlen zwischen -1 und 1 erzeugt, und zwar immer in der Geschwindigkeit von 44100 Zahlen pro Sekunde (Samplerate). Eine einzelne dieser Zahlen nennt man *Sample.*

Alles, was in dieser Geschwindigkeit erzeugt oder bearbeitet werden muss, ist in Pd durch Objekte mit einer Tilde „~“ gekennzeichnet. Untereinander sind diese Objekte mit dicken Kabeln verbunden. Wir nennen diese Zahlenreihen *Signale*.

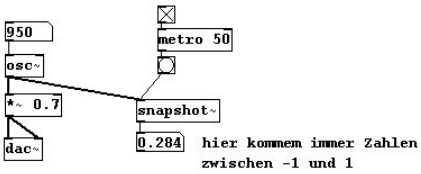


Dem Objekt „osc~“ beispielsweise können wir, wann immer wir wollen, eine neue Frequenz-Zahl als Input zuweisen; wir arbeiten damit nicht ständig. Daher eine dünne Verbindung. Aus dem Outlet des „osc~“-Objekts dagegen kommt *ununterbrochen* Klang, das heißt, Zahlen zwischen -1 und 1, 44100 pro Sekunde (*pro Sekunde* bedeutet: *Hertz*).

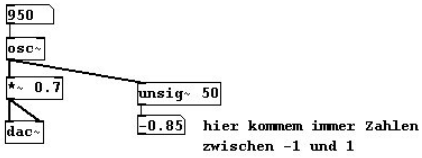
An den Outlet von „osc~“ können wir keine Number-Box anschließen, um diese Nummern zu sehen. Number-Boxen können nur für Kontroll-Verbindungen verwendet werden, nicht für Signal-Verbindungen. Signal-Verbindungen sind zu schnell, wir könnten 44100 verschiedene Zahlen pro Sekunde gar nicht visuell wahrnehmen. Ausschnitte jedoch können mit dem Objekt „snapshot~“

angezeigt werden. Als Input erhält es den Klang und einen Bang, der, angeklickt, dann den aktuellen Stand ausgibt, zum Beispiel dann zu einer Number-Box oder einem print-Objekt:

Wollen wir diese dauerhaft sehen, könnten wir ein (schnelles) Metronom anschließen:



In Pd-extended gibt es aber auch „unsig~“, was automatisch ein Metronom anschließt. Als Argument geben wir den Metronom-Wert:



Umgekehrt kann man mit „sig~“ auch Zahlen auf Kontroll-Ebene in Zahlen der Signal-Ebene umwandeln. Oben gibt man nur einmal einen Wert ein, der dann unten 44100 Mal pro Sekunde ausgegeben wird.

3.1.1.1.5 Samples – Millisekunden

Ähnlich wie für die Frequenz (und wie bei der Amplitude im folgenden Kapitel) gibt es also für die Zeitmessung in Pd *zwei* unterschiedliche Einheiten: Samples und Millisekunden. Samples sind in der Regel die Zählweise bei Signalen, Millisekunden bei Kontrolldaten.

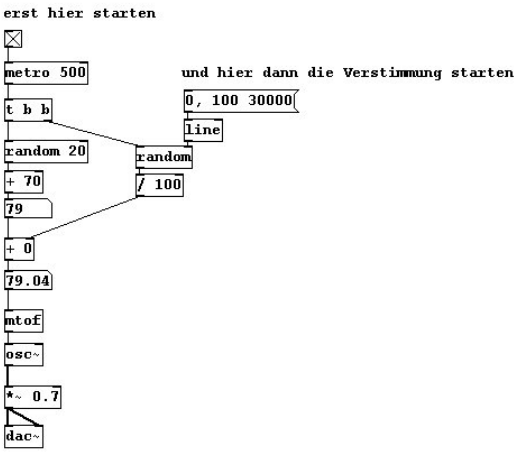
Umrechnung von Millisekunden-Dauern zu Sample-Dauern:

3.1.1.2 Anwendungen

3.1.1.2.1 Temperiert – Zufall

Random-Midi-Werte werden langsam ver-offsettet. (Also: Übergang von temperierter Stimmung zu random-Stimmung):

patches/3-1-1-2-1-random-offset.pd



3.1.1.2.2 Weitere Aufgabenstellungen

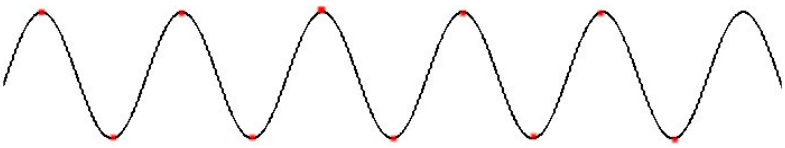
a) Erstellen Sie ein (für unser Ohr) lineares und ein logarithmisches Glissando von c bis c'".

b) Erstellen Sie eine Tonleiter in Vierteltönen.

3.1.1.3 Appendix

3.1.1.3.1 Nyquist-Theorem

Die Zahl 44100 ist mit gutem Grund gewählt. Wie gesagt wurde, hört der Mensch (idealiter) Frequenzen bis zu 20000 Hertz. Der US-amerikanische Physiker Harry Nyquist (1889-1976) stellte 1928 eine Theorie auf, derzufolge man zur digitalen Darstellung eines Signals mindestens die doppelte Signalfrequenz benötigt („Nyquist-Abtasttheorem“). Konkret heißt das, man braucht mindestens die Eckpunkte jeder Periode, um eine Welle in ihrem Grundgerüst darzustellen, also zwei Punkte pro Periode:

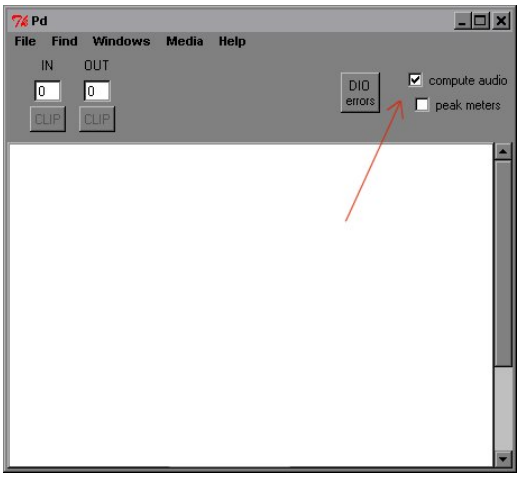


Bei einer Welle mit 20000 Hz, also mit 20000 Perioden pro Sekunde, brauchen wir mindestens 40000 Punkte pro Sekunde, um sie zu erfassen. Um sicherzustellen, dass dies für alle Wellen innerhalb des menschlichen Wahrnehmungsbereiches möglich ist, hat man entschieden, für Audio-CDs eine Samplerate von 44100 Hz zu veranschlagen. Damit kann man also Wellen bis 22050 Hz erzeugen. Für Computer existieren jedoch eine große Bandbreite von Frequenzen, die bis 8000 Herz für Systemklänge hinunterreichen. Für hochwertige Audioaufnahmen werden jedoch auch Sampleraten von 48000 Hz (bzw. 48 kHz = kiloHertz, für kilo = Tausend) arbeiten, mit 96 kHz oder gar mit 192 kHz verwendet.

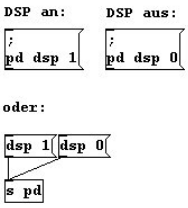
3.1.1.3.2 DSP

Es ist deutlich geworden, dass die Arbeit mit (vielen) Signalen für den Computer eine große Rechenarbeit darstellt. Man stelle sich nur vor, 100 „osc~“-Objekte zu haben. Jedes einzelne erzeugt 44100 Zahlen pro Sekunde, die miteinander synchronisiert sind. Daher gibt es in Pd die Option, DSP

(digital signal processing) im Hauptfenster auszuschalten, wenn es gerade nicht gebraucht wird, um den Prozessor von unnötiger Arbeit zu entlasten.



Man kann dies aber auch als Befehl verschicken; der Empfänger „pd“ ist in dem Fall das Programm selbst:



Grundsätzlich gilt für Computermusik, je schneller der Prozessor, desto höher die Leistung.

Pd erleichtert sich die Arbeit, indem es nicht jedes Sample einzeln berechnet und weitergibt, sondern immer eine Menge von Samples abarbeitet und dann als ganzen Block weitergibt. Das optimiert die Leistung wesentlich. Die standard-Blockgröße beträgt 64 Samples, kann aber neu eingestellt werden. Mehr dazu unter 3.8.1.1

3.1.1.4 Für besonders Interessierte

3.1.1.4.1 da- / ad-Wandlung

Wenn gesagt wurde, der „dac~“ schickt die in Pd erzeugten Zahlen zur Lautsprechermembran (3.1.1.1.1), ist das natürlich verkürzt ausgedrückt. Genau genommen wird in der Soundkarte des Computer aus den Zahlen ein elektrischer Strom mit variabler Spannung erzeugt („digital-analog-Wandlung“, Abk. „da-Wandlung“); je nach Höhe des Spannungswerts liegt dann die Membranposition. Genauso wird umgekehrt im Mikrofon zunächst aus dessen Membranstimmung eine variable Stromspannung gewonnen, die dann in der Soundkarte des Computers digitalisiert wird, d. h., in Zahlen transfomiert.

3.1.1.4.2 Schallwellen

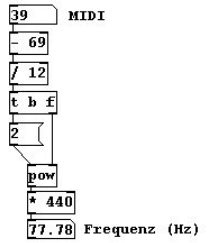
Schallwellen sind, anders als beispielsweise Wasserwellen, Longitudinalwellen. Longitudinalwellen sind dadurch charakterisiert, dass sie in Bewegunsrichtung des Schalls schwingen und nicht senkrecht

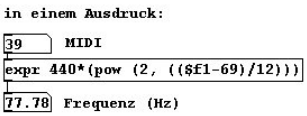
dazu, wie Transversalwellen). Für nähere Erläuterungen sei auf das Physikbuch der schulischen Oberstufe verwiesen.

3.1.1.4.3 Umrechnung MIDI Nummer – Frequenz

Die Umrechnung von MIDI-Nummer zu Frequenz erfolgt durch „mtof“. Die Formel für die Frequenz f und die MIDI-Nummer m lautet:

$$f = 440 \cdot 2^{(m-69)/12}$$





Wollen wir einfach die Frequenz eines temperierten Tons wissen, der in einem bestimmten Abstand zu einer bekannten Frequenz steht, benötigen wir diese Formel:

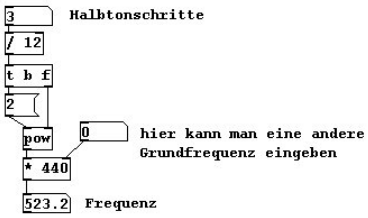
$$f = g \cdot 2^{a/12}$$

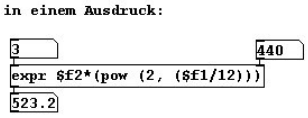
f bezeichnet die gesuchte Frequenz, g die Frequenz des Bezugstons, a den Abstand in Halbtönen.

Möchten wir zum Beispiel die Frequenz von c'' ermitteln und wissen, dass a' die Frequenz 440 Hz hat, setzen wir ein:

$$f = 440 \cdot 2^{3/12} = 523.2$$

In Pd:





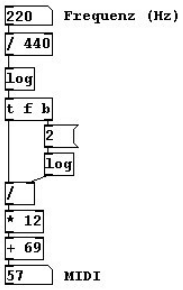
Für die umgekehrte Berechnung von Frequenz in MIDI lautet die Formel:

$$m = 69 + 12 \cdot \log_2(f/440)$$

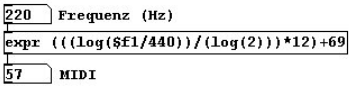
Wobei wir in Pd nur den natürlich Logarithmus auf Basis der Euler'schen Konstante haben; daher brauchen wir noch diese Formel:

$$\log_a(b) = \frac{\log_e(b)}{\log_e(a)}$$

Somit in Pd:



in einem Ausdruck:



3.1.1.4.4 Geräuschperiodizität

Es wurde gesagt, dass Geräusche keine Periode haben. Nun könnte man sich aber vorstellen, dass ein Rauschen zehn Sekunden läuft und dann in einer exakt gleichen Schleife wiederholt wird – dann hätte das Rauschen theoretisch doch eine periodische Frequenz, nämlich von 0.1 Hz. Darum lautet die Definition für ein Geräusch genauer, dass es unperiodisch ist oder eine Periode von weniger als 20 Hz hat. Entsprechend kann auch gesagt werden: Die Frequenzen von Geräuschen haben evtl. einen gemeinsamen Grundton, nur eben unterhalb von 20 Hz.

Das Gebiet der Akustik birgt lauter spannende Experimente, etwa in Form der Längenberechnung von Schallwellen oder des Dopplereffekts. An dieser Stelle sei ausdrücklich auf die Lektüre einschlägiger Akustiklehrbücher verwiesen.

3.1.2 Lautstärke

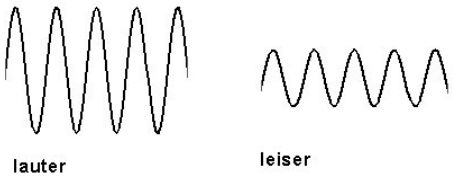
3.1.2.1 Theorie

3.1.2.1.1 Messung

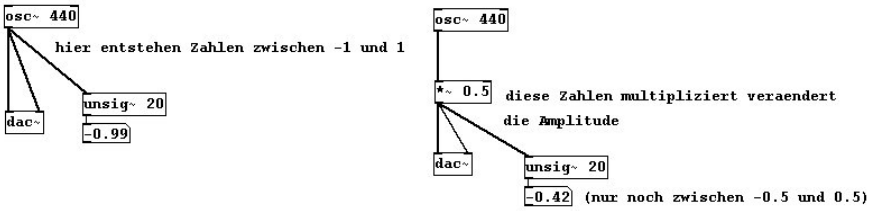
Die nächste Eigenschaft eines Tons, die wir betrachten wollen, ist seine Lautstärke. Traditionell wird Lautstärke in Musik mit Anweisungen wie pianissimo, piano, mezzoforte etc. notiert. Ihre Verwendung ist allerdings subjektiv und je nach Instrument mitunter sehr verschieden. In der Physik, und danach richtet sich Pd, wird Lautstärke objektiviert mit deziBel- oder root-mean-square-Werten. Die beiden sind vergleichbar mit MIDI-Nummern und der Frequenz-Angabe für Tonhöhe. deziBel

(Abk. dB) richtet sich linear nach unserem persönlichen Höreindruck, wobei quasi eine Oktav in Lautstärke dem Intervall von 6 dB entspricht. Insgesamt rangiert die Skala von 0 bis 130 dB – wobei absolute Stille immer noch einen Wert von zwischen 15 und 20 dB hat und Schall von über 120 dB Lautstärke gesundheitsschädlich ist. Ab 130 dB nehmen wir Schall praktisch nur noch als Schmerz wahr. root-mean-square-Werte (Abk. rms) entsprechen wie bei Frequenz-Werten nicht unserem Höreindruck, sondern verlaufen logarithmisch von 0 bis 1. 0 entspricht dabei 0 dB, 1 entspricht 100 dB. Unter rms versteht man den geometrischen Mittelwert, der aus einer Folge von Amplitudenwerten errechnet wird. Hierzu werden die Zahlen zunächst quadriert, dann der Mittelwert gebildet (duch Summation und Division durch die Anzahl der Elemente) und anschließend aus dem Mittelwert die Wurzel gezogen. Bei einem Audiosignal wird der rms-Wert über einen zeitlich begrenzten Ausschnit des Signals ermittelt und gibt bei einem stark schwankenden Signal, wie einer Schwingung, einen Anhaltspunkt über die ungefähre durchschnittliche Amplitude des Signals in diesem Ausschnitt. Für die Umrechnung vom einen ins andere gibt es folgende Objekte in Pd:

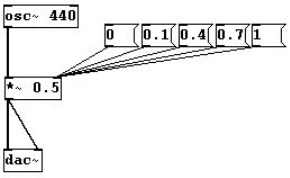
Die Lautstärke einer Schwingung bestimmt sich aus ihrer Amplitude. Das ist der Grad der Auslenkung der Membran über den Nullpunkt hinaus nach vorne und hinten. Je stärker sich die Membran bewegt, desto lauter nehmen wir den Klang wahr. In unserer Achsendarstellung sieht das so aus:



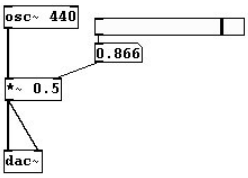
Es muss immer wieder betont werden: Bis dies mit dem „dac~“-Objekt an den Lautsprecher geschickt wird, arbeitet Pd nur mit Zahlen. Wenn also ein Klang leiser ist, heißt das, die Zahlen werden nicht mehr zwischen -1 und 1 produziert, sondern in kleinerem Ambitus um den Nullpunkt herum, zum Beispiel nur zwischen -0.5 und 0.5. Wir bewerkstelligen dies im Patch, indem wir die vom „osc~“-Objekt produzierten Zahlen mit einem Faktor versehen:



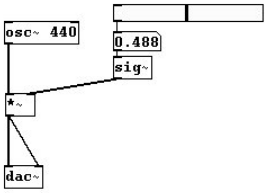
So können wir alle Grade von Lautstärke einstellen, von absoluter Stille bis zur größten Lautstärke (letztere hängt natürlich noch von den vorhandenen Lautsprechern und dem Verstärker ab).



Nun könnte man auch eine Art Schieberegler anbringen, und zwar mit dem HSlider (**Put # HSlider**), den man auf einen Bereich zwischen 0 und 1 einstellt (vgl. Kapitel 2.2.2.3.2 und 2.2.4.3.4):

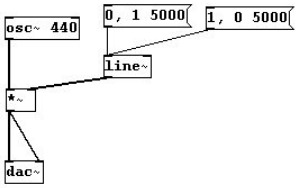


Spätestens wenn man diesen Slider schnell bewegt, kommen nun aber Störgeräusche auf. Das liegt daran, dass ein Signal, das in Samples gerechnet wird, in Konfrontation mit einer Kontrolldaten-Bearbeitung (die in Millisekunden abläuft) gerät. Wenn es nur um vereinzelte Zahlen geht wie zuvor bei den Faktoren 0, 0.1, 0.4, 0.7, 1, ist das irrelevant; ab einer bestimmten Geschwindigkeit spielt es allerdings sehr wohl eine Rolle. Darum muss man die Kontrolldatenverbindung dann durch ein Signal-Pendant ersetzen. Wir konvertieren mit dem „sig~“-Objekt:

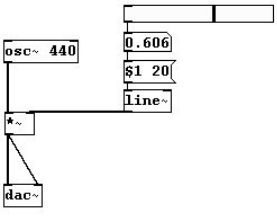


Damit ist nun Synchronität zwischen der Zahlenproduktion des Oszillators (44100 Zahlen pro Sekunde) und der Eingabe des Faktors („sig~“ macht daraus ebenfalls genau 44100 Zahlen pro Sekunde) hergestellt. Es gilt noch zu beachten: Kommt in den rechten Inlet von „*~“ ein Signal, darf als Argument nichts in dem Objekt stehen; stünde darin ein Argument (wie vorhin 0.5), ginge das Objekt davon aus, dass es rechts einen Kontrolldaten-Input erhielte.

Um ein Crescendo oder Decrescendo zu erstellen, müssen wir entsprechend „line~“ verwenden:

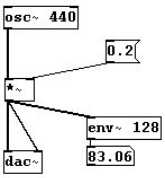


Eine elegantere Verwendung des Sliders als Lautstärke-Schieberegler („Potentiometer“) wäre folgende:

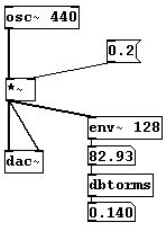


So wird noch zwischen jedem Schritt ein kleines Crescendo / Decrescendo durchgeführt. Diese Auffüllung von Schritten mit Zwischenwerten nennt sich „Interpolation“ (wie schon mit Tonhöhen in Kapitel 2.2.3.2.3 geschehen).

Wir können die Lautstärke eines Tons mit dem Objekt „env~“ ermitteln, das die Lautstärke in dB ausgibt. Es muss immer ein Zeitraum definiert werden, innerhalb dessen ein Durchschnittswert gebildet wird; wir geben ihn als Argument in Samples an (üblicherweise als Sample-Zahl eine 2er-Potenz):

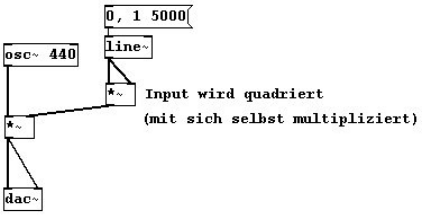


Mit der Umrechnung in rms ...



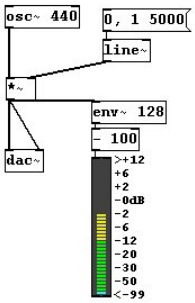
... sieht man, dass die Faktoren zwischen 0 und 1 nicht zu verwechseln sind mit rms-Werten zwischen 0 und 1.

Wie schon angedeutet, entspricht das menschliche Hören bei der Lautstärke wie bei Tonhöhe nicht den Ergebnissen der physikalischen Messung (wie in der Gegenüberstellung der Schaubilder für Tonhöhen in Frequenzen und Intervallen). Ein einfacher Trick, um einen linearen Eindruck einer Lautstärkezunahme oder -abnahme zu erzielen, ist es, die Werte zu quadrieren:



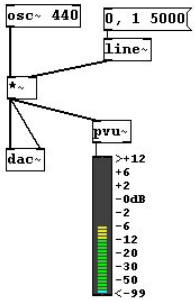
Hier sind aber alle möglichen Verläufe auszuprobieren; letztlich ist es eine kompositorische Entscheidung, wie die Lautstärke zu- oder abnehmen soll. Was denn eine „Lautstärken-Oktav“ sei, kann nicht so objektiviert werden wie bei Tonhöhen.

Zur Visualisierung der Amplitude gibt es in Pd ein eigenes GUI-Objekt: Den VU-Meter (**Put # VU**). Als Input erhält er eine dB-Angabe. Allerdings ist er so konstruiert, wie Anzeigen auf herkömmlichen Mischpulten eingestellt sind: 100 dB werden als 0 dB angezeigt und Abweichungen davon gehen dann in den Minus- oder Plus-Bereich. Dies muss so bereits als Input gegeben werden. Also einfach vom Output von „env~“ 100 subtrahieren:



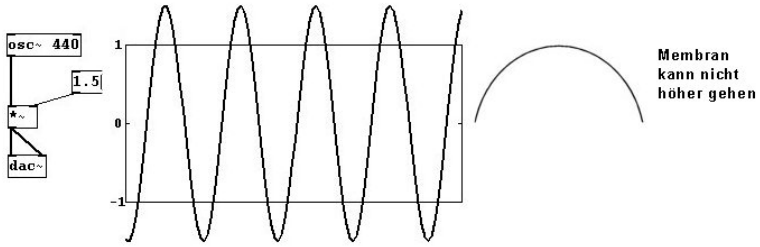
Dann zeigt der VU Lautstärkeänderungen grafisch an. (VU ist die Abk. für „Volume“ = engl. für Lautstärke).

In Pd-extended gibt es zur Konversion für den VU-Meter auch das Objekt „pvu~“:

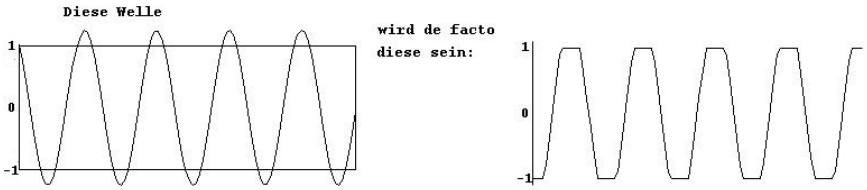


3.1.2.1.2 Probleme

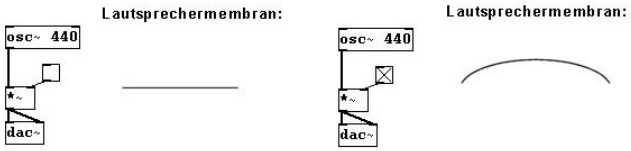
Wichtig ist noch: Amplituden über 1 und unter -1 werden 'abgeschnitten'. Erhält der Lautsprecher vom „dac~“ einen Wert jenseits von 1 oder -1, bleibt die Membran an der äußersten Stelle einfach stehen.



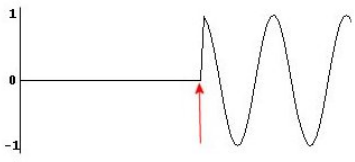
Erhöht man die Lautstärke eines Klanges derart, dass der Verlauf seiner Schwingung durch das Stehenbleiben der Membran 'abgeschnitten' wird, entsteht der Effekt, den man *Übersteuerung* nennt.



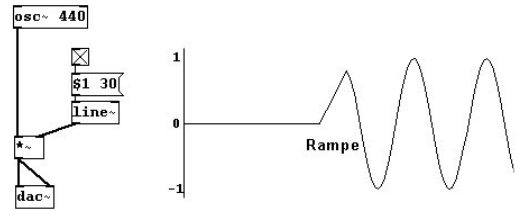
Eine weitere Widrigkeit ist folgende: Wird die Position der Lautsprechermembran ruckartig um ein größeres Intervall versetzt (wenn man zum Beispiel den Klang einschaltet), erklingt ein sogenannter „Klick“:



Dies tritt jedoch nur dann stark in Erscheinung, wenn als eigentlicher Klang eine Schwingung verwendet wird, die selbst eine sehr geschmeidige Membranbewegung erzeugt, also z. B. der Sinuston. Im Schaubild ist der „Ruck“ gut zu erkennen:

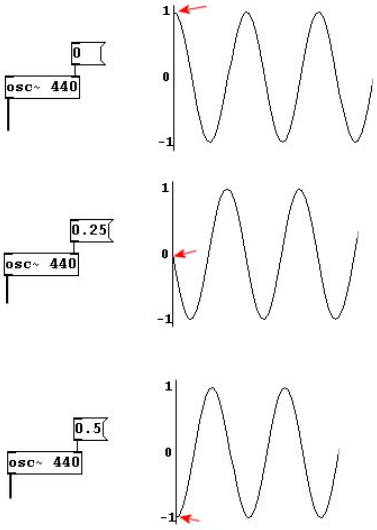


Ein „Ruck“ bedeutet in der Regel eine Versetzung, die schneller als 30 ms ist. Um solche Klänge ohne Klick einzuschalten, sollte man daher eine sogenannte „Rampe“ bauen, d. h. ein sehr schnelles Crescendo am Anfang und am Ende erzeugen:



3.1.2.1.3 Phase

Außerdem lässt sich an der Welle eines Klangs in Pd einstellen, bei welcher Membranposition er beginnen soll (oder wo er hinspringen soll). Dies nennt man die *Phase* einer Welle. Man kann die Phase in Pd im rechten Inlet des „osc~“-Objekts einstellen, mit Zahlen zwischen 0 und 1:

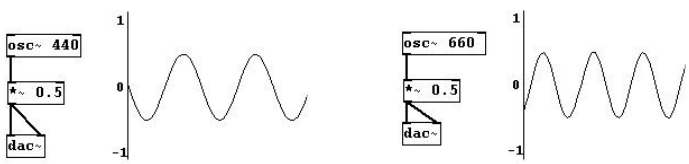


Die gesamte Periode einer Welle wird also wieder eingefasst in den Bereich zwischen 0 und 1. Häufig wird sie aber auch in Grad benannt, wobei die komplette Periode 360 Grad hat. Zum Beispiel ist häufig die Rede von einer „Phasenverschiebung um 90 Grad“. Dem entspricht in Pd ein Input für die Phase von 0.25.

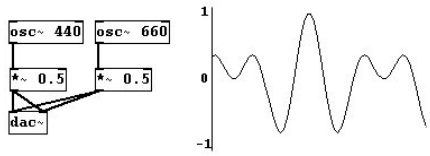
Die Änderung der Phase einer Welle hat zunächst für unser Ohr keine besondere Auswirkung. In anderen Zusammenhängen werden auf die Phasenlage allerdings zurückkommen.

3.1.2.1.4 Schallwellen addieren sich

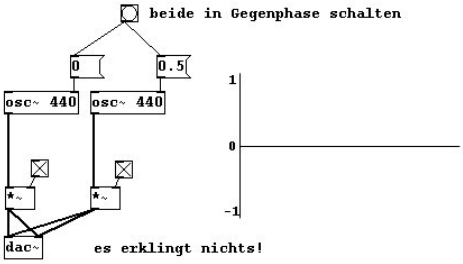
Hat man diese beiden Oszillatoren:



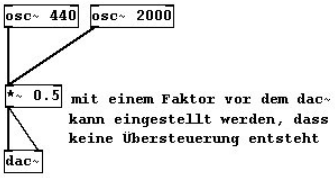
... und schließt sie zusammen an den „dac~“, ergibt sich Folgendes:



Die beiden einzelnen Wellen gingen (wegen des Faktors) nur von -0.5 bis 0.5. Zusammen aber reichen sie nun von -1 bis 1 und bilden eine etwas komplexere Form. Das liegt daran, dass sich Schallwellen grundsätzlich *addieren*. Einfach gesagt: Wie haben nur eine Luft, die alle Schwingungen, die ausgelöst werden, gleichzeitig vollziehen muss. Addition heißt aber auch, dass es zu Auslöschungen kommen kann. Gegenläufige Wellen, bei denen die eine immer gerade „nach hinten“ geht, während die andere „nach vorne“ geht, gleichen einander aus. Das ist der Fall, wenn gleich schnelle Schwingungen in Gegenphase sind:



Normalerweise müssen wir aber bei Beteiligung mehrerer Klangproduzenten an einem Gesamtklang die Summe so heruntermultiplizieren, dass sie einen Wert von 1 bzw. -1 nicht über- bzw. unterschreitet:



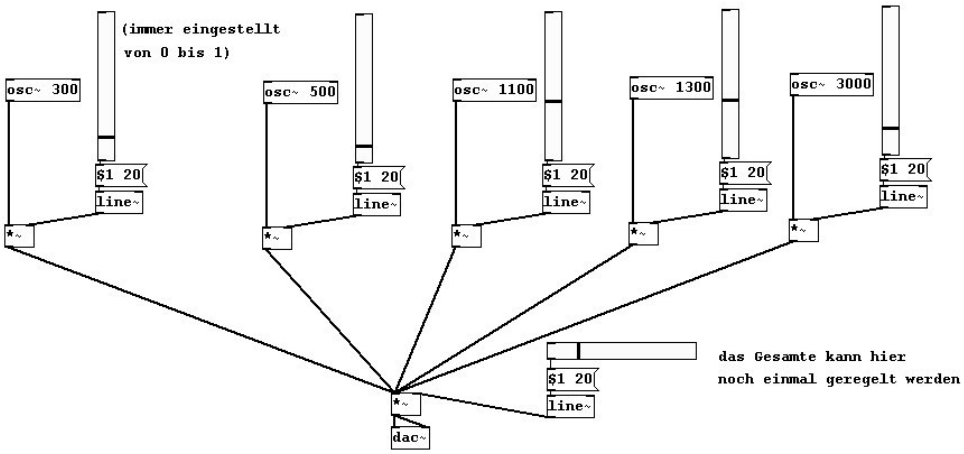
Hier werden einfach beide Oszillatoren an ein Multiplikationsobjekt angeschlossen. Dadurch werden sie zunächst automatisch addiert (wenn immer mehrere Signale als Input eines einzelnen Objekts gegeben werden, werden sie zunächst addiert und dann gemäß dem Objekt bearbeitet), und dann der Multiplikation unterzogen.

3.1.2.2 Anwendungen

3.1.2.2.1 Akkord

Wir erstellen einen Akkord mit variabler Lautstärke für jeden einzelnen Ton:

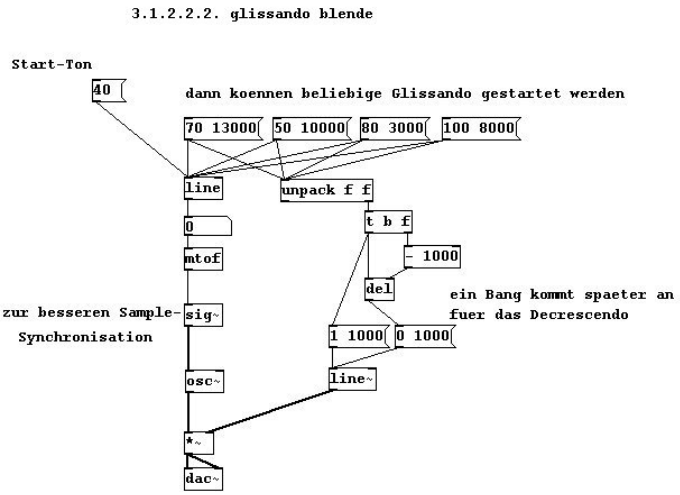
3-1-2-2-1-akkord.pd [patches/3-1-2-2-1-akkord.pd]



3.1.2.2.2 Glissandi

Glissandi, die am Anfang und Ende geschmeidig ein- bzw. ausblenden:

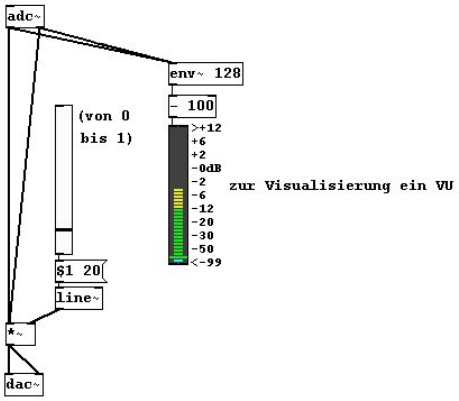
patches/3-1-2-2-2-glissandi-blende.pd



3.1.2.2.3 adc-Input bearbeiten

Nehmen wir ein Mikrofon, sprechen etwas hinein und geben dies mit veränderter Lautstärke wieder aus:

patches/3-1-2-2-3-input-bearbeiten.pd

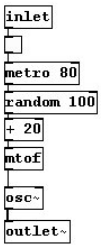


3.1.2.2.4 Oszillatorenkonzert

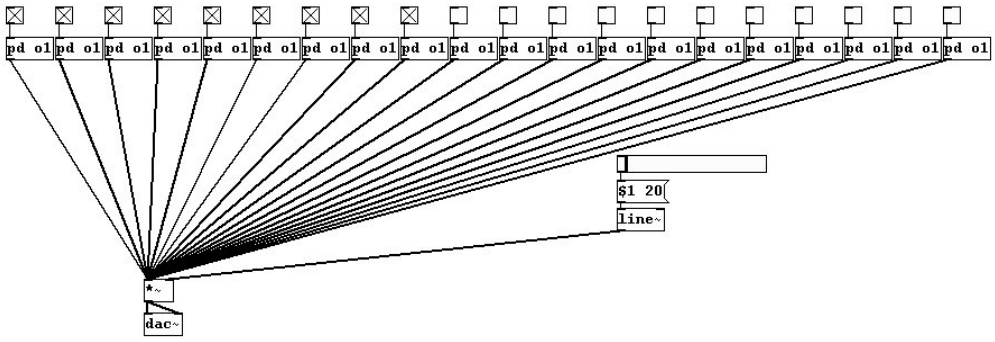
Werden wir 'symphonisch': Warum nicht gleich zwanzig verschiedene Oszillatoren etwas spielen lassen?

patches/3-1-2-2-4-oszillatorenkonzert1.pd

Erstellen wir zunächst diesen Subpatch „o1“:



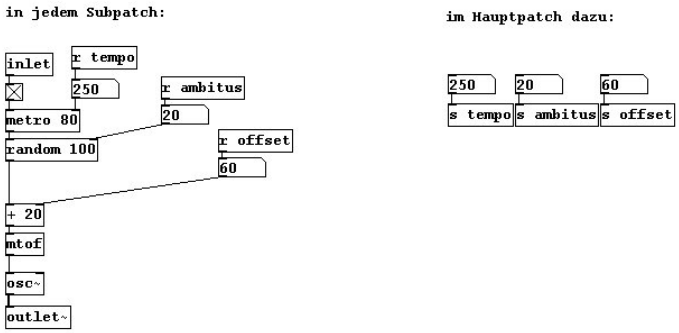
... und vervielfältigen ihn dann einige Male



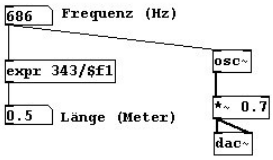
Nun schalten wir alle nacheinander ein!

Hier sei nur angedeutet, dass sich nun natürlich alle Parameter ändern lassen – und schon hat man etwas zum Spielen:

patches/3-1-2-2-4-oszillatorenkonzert2.pd



Schall bewegt sich in der Luft bei 20 Grad Celsius mit ca. 343 Metern pro Sekunde. Wir können nun die räumliche Länge einer Periode berechnen und das Ergebnis auch gleich überprüfen ...



... indem wir zum Beispiel bei einer Frequenz von 686 Hertz unseren Kopf einen halben Meter weit hin und her bewegen, hören wir deutlich Wellenberg und Wellental.

3.1.2.2.5 Weitere Aufgabenstellungen

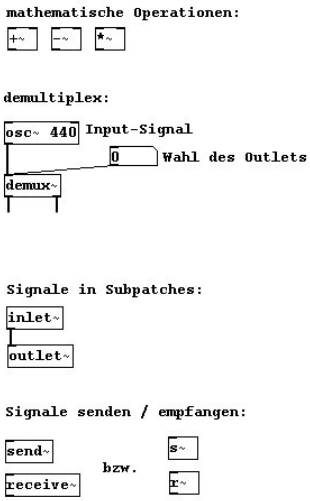
a) Erstellen Sie (random-)Glissando-Akkorde, zusätzlich noch mit einer random-Lautstärkeänderung für jeden einzelnen Ton.

b) Die Lautstärke des Mikrofoneingangs soll die Tonhöhe eines Oszillators steuern (dann auch mehrere mit jeweils verschiedenem Frequenz-Offset)!

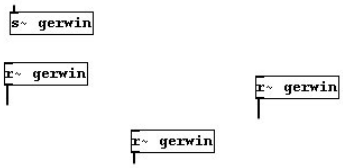
3.1.2.3 Appendix

3.1.2.3.1 Weitere Tilde-Objekte

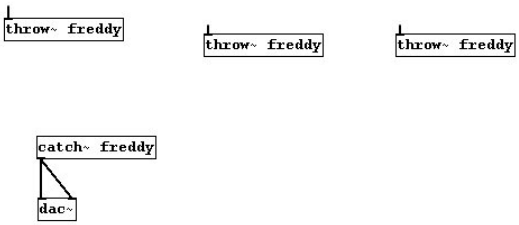
Von einigen Objekten, die wir in Kapitel 2 kennengelernt haben, gibt es auch eine Version mit Tilde. Sie funktionieren genau gleich, nur dass sie eben Signale statt Kontrolldaten verarbeiten:



Mit „send~“ kann etwas an beliebig viele „receive~“-Objekte geschickt werden; umgekehrt darf es aber nur ein „send~“-Objekt sein:



Umgekehrt kann man viele verschiedene Signale an eine zentrale Stelle leiten (eben zum Beispiel zum „dac~“), mit „throw~“ und „catch~“:



3.1.2.3.2 Bit-Tiefe

Relevant für Pd ist ferner die Bit-Tiefe. Der Prozessor eines Computers arbeitet nur mit dem binären Zahlencode, d. h. mit 0 und 1. Die Bit-Anzahl besagt, wieviele Plätze für Nullen und Einsen verwendet werden. Hätte man z. B. nur zwei Stellen, könnte man nur 2^2, also vier verschiedene Kombinationen erstellen:

```
0 0
0 1
1 1
1 0
```

Je mehr Stellen man hat, desto mehr bzw. detaillierter kann etwas verarbeitet werden. Für Pd, wo ja Erscheinungen wie Frequenzen, Amplitude etc. mit Zahlen berechnet werden, bedeutet dies, dass diese Zahlen je nach Bit-Tiefe genauer berechnet werden können – die Anzahl der Kommastellen

hängt also davon ab. Pd arbeitet normalerweise mit 16 Bit, das entspricht der Qualität einer Audio-CD. 16 Bit entspricht 2^{16} = 65,536 möglichen Werten für ein Sample.

3.1.2.4 Für besonders Interessierte

3.1.2.4.1Schalldruck vs. Schallintensität

Lautstärke und vor allem Lautstärke-Intervalle sind objektiv wie subjektiv sehr abhängig, z. B. von den Charakteristika des Raumes, vom Lebensalter des Hörers etc. Es gibt auch nicht eine einzige präzise Messung für Lautstärke; bekannt sind die Theorien von Schalldruck und Schallintensität. Ergänzend hierzu sei ausdrücklich der Blick in ein Buch über Akustik empfohlen.

3.1.2.4.2 Kontrolldaten vs. Signale

Wir haben nun gesehen, dass es für die wesentlichen Bereiche der Klangerzeugung in Pd jeweils zwei verschiedene Einheiten gibt: für die Tonhöhe Frequenz und MIDI-Nummern, für die Amplitude root-mean-square und deziBel und für die Zeit Millisekunden und Samples.

Für Letzteres soll noch das weiter oben angeführte Beispiel mit Crescendo/Decrescendo von „line~“ erläutert werden:

Würde man „line“ (ohne Tilde) hierzu einsetzen, kann es zu unerwünschten Knacksern oder Ähnlichem kommen, da die beiden Zeitmaße kombiniert werden; das Problem ist, dass sie nicht synchronisiert sind. Es kann passieren, dass die Zahlenschritte nicht zusammentreffen und dadurch Unregelmäßigkeiten verursachen, die sich durch kurze Verzögerungen oder eben Knackser äußern.

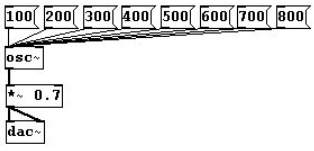
Wie in Kapitel 2.2.3.3.2 beschrieben wurde, gibt „line“ alle 20 Millisekunden einen Wert. Das heißt, es trifft möglicherweise nicht mit dem Sample-Aufkommen zusammen. Zwar kommt ca. alle 0.02 Millisekunden ein neues Sample, dennoch kann ein „line“-Wert gerade nicht mit einem mehr oder weniger gleichzeitigen Sample zusammentreffen und dann kommt es zu Komplikationen. „line~“ (mit Tilde) hingegen erzeugt ein Signal mit 44100 Werten pro Sekunde; diese 44100 Werte werden exakt zeitgleich generiert wie sie irgendein anderes Tilde-Objekt erzeugt; sie sind immer synchron. Der Computer arbeitet grundsätzlich 44100 Samples pro Sekunde ab und an verschiedenen Orten im Patch immer genau zeitgleich.

3.2 Additive Synthese

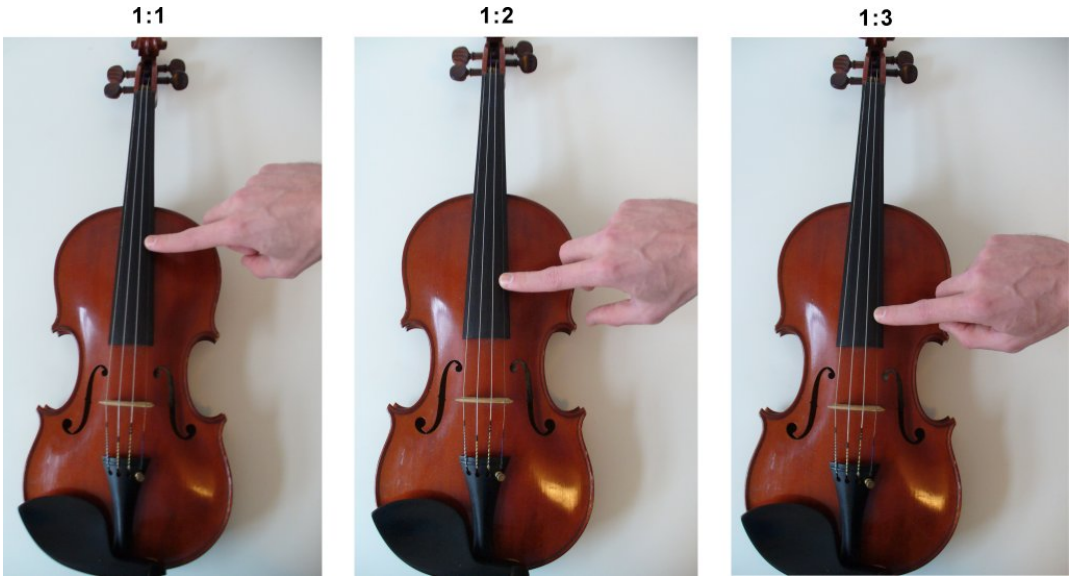
3.2.1 Theorie

3.2.1.1 Die Teiltonreihe

Die additive Reihe von Frequenzen (wenn also immer derselbe Hertz-Betrag hinzuaddiert wird), aus der eine Abfolge immer kleinerer Intervalle entsteht, heißt Teiltonreihe:

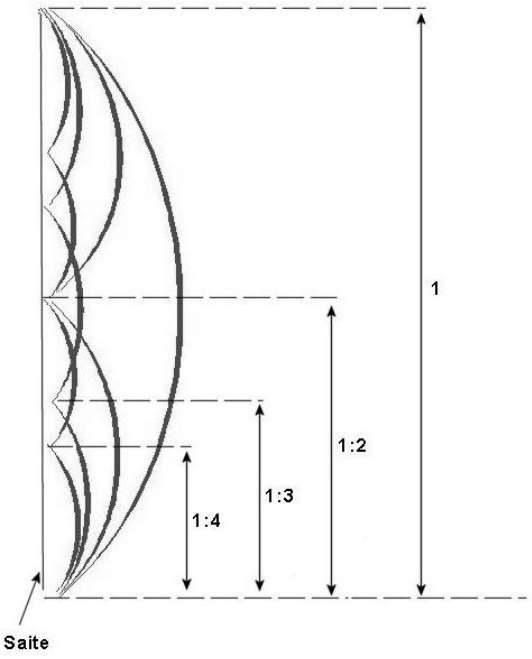


Sie entsteht ebenso, wenn man dem Experiment von Pythagoras (ca. 570 - ca. 510 v. Chr.) folgt, durch Teilung einer Saite in verschiedene Proportionen:



Die Zahlenverhältnisse beschreiben die Längen der beiden Saitenteile zueinander.

Wird eine Saite als Ganze gestrichen, schwingt sie nicht nur im Ganzen, sondern zugleich in all den weiteren Teilen:

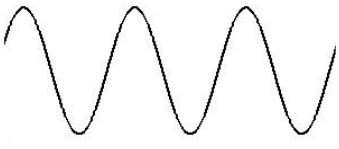


Hier beschreiben die Zahlenverhältnisse die Länge des Teils zur Gesamtlänge der Saite.

Somit klingen auch alle diese Teilschwingungen mit; jeder Klang einer Saite ist also bereits ein Akkord!

Das Besondere an gerade diesem Akkord ist aber, dass alle seine Töne sehr stark miteinander verschmelzen, zumindest wenn ihre Lautstärken nach oben hin abnehmen. Jeder in der Natur vorkommende Ton hat Obertöne, dennoch nehmen wir einer Eigenheit des menschlichen Ohrs zufolge das Ganze als nur einen Ton wahr.

Andererseits besitzen all diese Schwingungen selbst als Teil einer komplexeren Schwingung keine Obertöne. Einen isolierten Klang ohne Obertöne gibt es in der Natur nicht – mit elektronischen Mitteln kann man sie aber erzeugen. Sie heißen Sinustöne, da ihre Welle diese sinusartige Form hat:

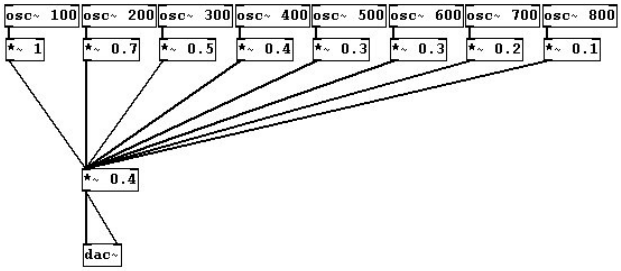


Der Physiker Jean Baptiste Joseph Fourier (1768-1830) fand heraus, dass sich jeder periodische Klang volständig in eine Reihe von Sinustönen (unterschiedlicher Frequenz, Amplitude und Phase) zerlegen lässt, deren Summe mit dem Original identisch ist. Eine solche Analyse und das zugehörige mathematische Verfahren nennt man die Fourier-Analyse bzw. Fourier-Transformation.

Folglich kann man umgekehrt jeden periodischen Klang erstellen, also „synthetisieren“, wenn man mehrere Sinustöne schichtet.

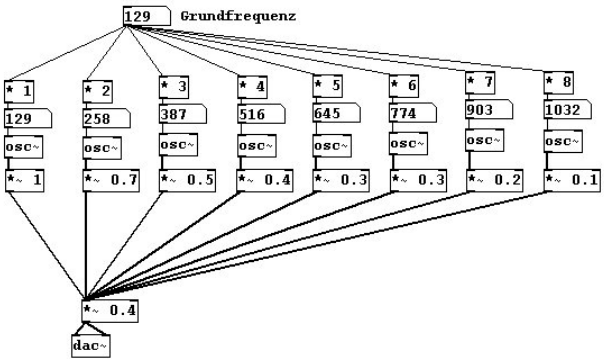
In Pd kann, wie schon gesagt, die Welle eines Sinustons mit dem Objekt „osc~“ erzeugt werden. Sinustöne sind für die Elektronische Musik insofern sehr charakteristisch, weil sie nur mit elektronischen Mitteln erzeugt sind.

Mit mehreren „osc~“-Objekten, deren Frequenzen also eine additive Reihe darstellen, kann man einen Teiltonakkord erstellen:



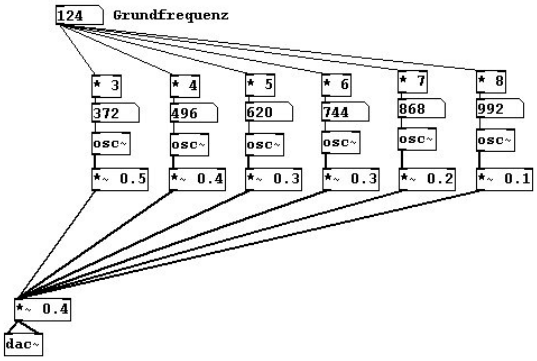
Grundsätzlich werden die Amplituden nach oben hin zur besseren Verschmelzung meist leiser (jedoch gibt es auch Instrumente, für die es gerade charakteristisch ist, dass manche Obertöne lauter sind als ihre unteren und oberen Nachbarn, zum Beispiel die Klarinette). Die Art der Zusammensetzung von Obertönen und ihren Lautstärken macht die *Farbe* eines Klanges aus. Als Oberbegriff redet man von seinem *Spektrum*. (Hinweis: Der Begriff "Teiltöne" beinhaltet die Grundfrequenz, im Gegensatz zu "Obertöne". Das heißt, der erste Teilton ist die Grundfrequenz, der zweite Teilton der erste Oberton, usw.).

Dass die Obertöne im Ohr verschmelzen, realisiert man in dem Moment, in dem die Grundfrequenz bewegt wird:



Wir beschränken uns hier auf die ersten acht Teiltöne.

Selbst wenn wir untere Teiltöne weggelassen, empfinden wir die Grundfrequenz als Grundton, wenn wir sie ändern:



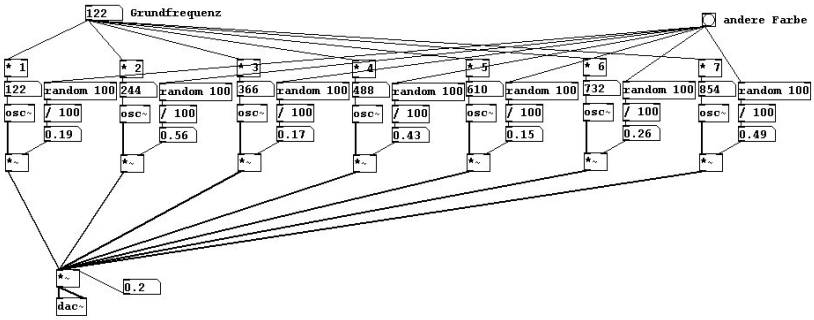
Unser Gehirn schließt eben aus dem übrigen Spektrum auf den Grundton. Diesen nicht vorhandenen Ton nennt man *Residualton*.

3.2.2 Anwendungen

3.2.2.1 Eine Random-Klangfarbe

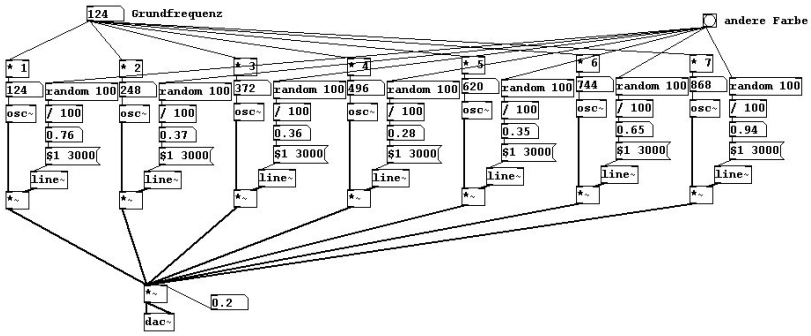
patches/3-2-2-1-random-klangfarbe.pd

Aus Platzgründen ist dieses Beispiel auf die ersten sieben Teiltöne beschränkt:



3.2.2.2 Verwandlung einer Klangfarbe in eine andere

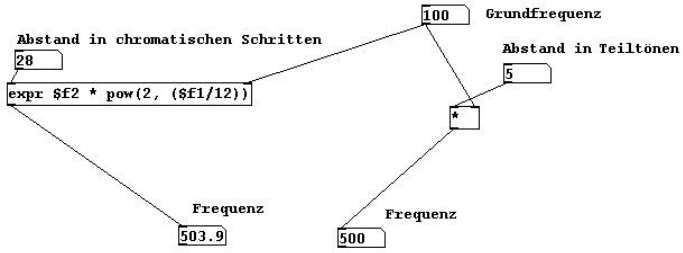
patches/3-2-2-2-klangfarbenverwandlung.pd



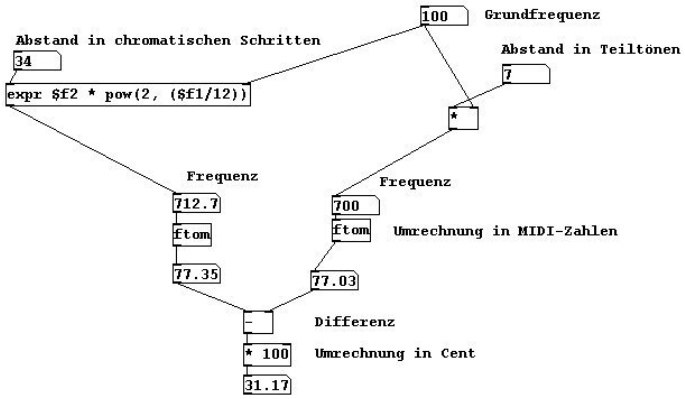
3.2.2.3 Natürlich vs. temperiert

Betrachten wir noch den Unterschied zwischen natürlichen und temperierten Intervallen (erst die Grundfrequenz eingeben!):

patches/3-2-2-3-natuerlich-temperiert.pd



Die Differenz zwischen natürlicher und temperierter Stimmung in Cent (Hundertstel eines Halbtones) angezeigt:



Wir sehen hier: Der 7. Teilton ist ca. 31 Cent tiefer als die temperierte Sept.

3.2.2.4 Weitere Aufgabenstellungen

Erstellen Sie einen Teiltonakkord mit manipulierten Obertönen, das heißt nicht mehr genauen Obertönen.

3.2.3 Appendix

3.2.3.1 Grenzen von Pd

Das vorige Beispiel der Random-Klangfarbe offenbart auch schon eine Grenze von Pd: Wir können nicht per Zufall die Anzahl der Oszillatoren bestimmen – wir müssen zumindest das Maximum vorab platzieren.

3.2.4 Für besonders Interessierte

3.2.4.1 Studie II

Ein Pionierstück der Elektronischen Musik ist die Studie II von Stockhausen, komponiert 1954. Darin werden nur Sinustöne und deren Gemische in nicht-temperierten Intervallen verwendet. Eine Analyse dieses Stückes sei an dieser Stelle empfohlen!

3.2.4.2 Spektren-Komposition

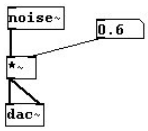
Der Komponist und Theoretiker Trevor Wishart beschreibt im vierten Kapitel seines Buches „Audible Design“ viele Möglichkeiten der Komposition von Spektren.

3.3 Subtraktive Synthese

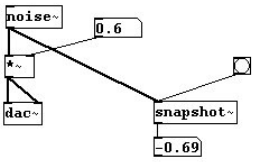
3.3.1 Theorie

3.3.1.1 Weißes Rauschen

Claude Debussy entgegnete einmal auf die Frage, wie er komponiere, dass er einfach alle Töne nehme und dann die, die er nicht möge, wieder weglasse. Er nahm die Idee der Filterung vorweg. Anders als die additive Synthese, die quasi vom Atom des Klanges, dem Sinuston, ausgeht, verwendet die subtraktive Synthese zunächst allen Klang und reduziert ihn dann. Allen Klang zu erzeugen ist tatsächlich möglich. Sämtliche Frequenzen auf einmal erhält man, wenn die Membran des Lautsprechers völlig zufällig, chaotisch schwingt. Dafür ist in Pd das Objekt „noise~“ verantwortlich:



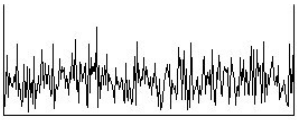
„noise“ müsste korrekt eigentlich „random~“ lauten, denn es erzeugt 44100 Zufallszahlen pro Sekunde, und zwar Zahlen zwischen -1 und 1, also für Positionen der Membran.



3.3.1.2 Filter

Das Rauschen, das alle Frequenzen enthält, nennt man analog zum Licht „weißes Rauschen“: Das normale, weiße Licht enthält alle Lichtfrequenzen, während zum Beispiel rotes oder blaues Licht durch Filtervorsätze entsteht.

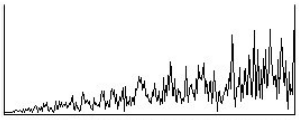
Solche Filter gibt es in Pd auch, etwa den „Lowpass“, der nur die tiefen Frequenzen durchlässt und die hohen unterdrückt. Zur Darstellung verwenden wir nun ein neues Schaubild, das auf der x-Achse die Frequenzen und auf der y-Achse die Amplituden zeigt:



weißes Rauschen

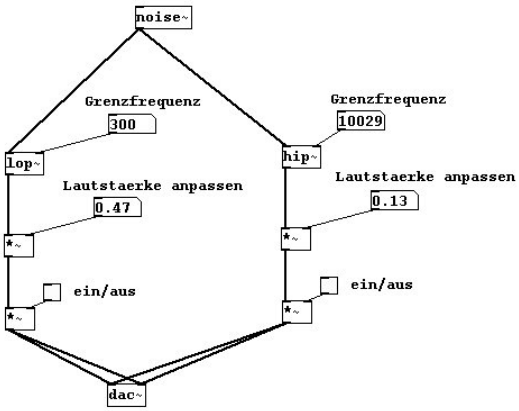
weißes Rauschen mit Tiefpassfilterung

Umgekehrt gibt es den „Highpass“-Filter, der nur hohe Frequenzen durchlässt:

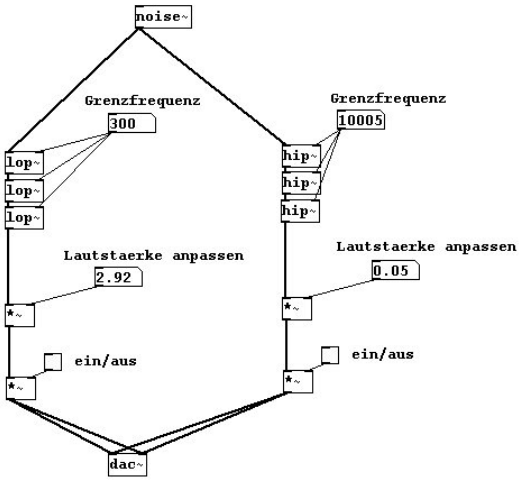


weißes Rauschen mit Hochpassfilterung

In Pd heißen die Objekte „hip~“ und „lop~“. Als Argument bzw. rechten Input erhalten sie die Frequenz, ab der gefiltert werden soll.

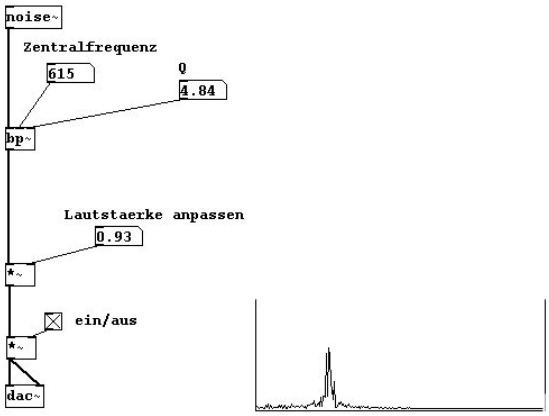


Wie aus den Grafiken zuvor schon ersichtlich war, arbeiten die beiden Filter nicht sonderlich 'steil'. Man kann ihre Wirkung aber verstärken, indem man mehrere hintereinanderschaltet (Kaskade):

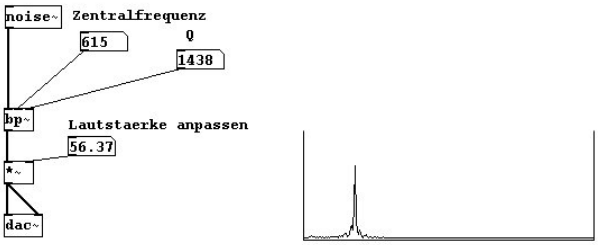


Wir müssen die Lautstärke immer wieder neu anpassen, denn Filter reduzieren die eingehende Energie. (Dafür verstärken sie manchmal anderes.)

Eine weitere Filterart ist der „Bandpass“. Hier wird nur ein kleiner Teil um eine Zentralfrequenz herum durchgelassen, eine Art 'Band' von Frequenzen. Als Argumente / Inlets gibt man die Zentralfrequenz und die Stärke des Bandes, genannt „q“.



Theoretisch sollte man mit einem immer stärkeren Band irgendwann beim Sinuston herauskommen:



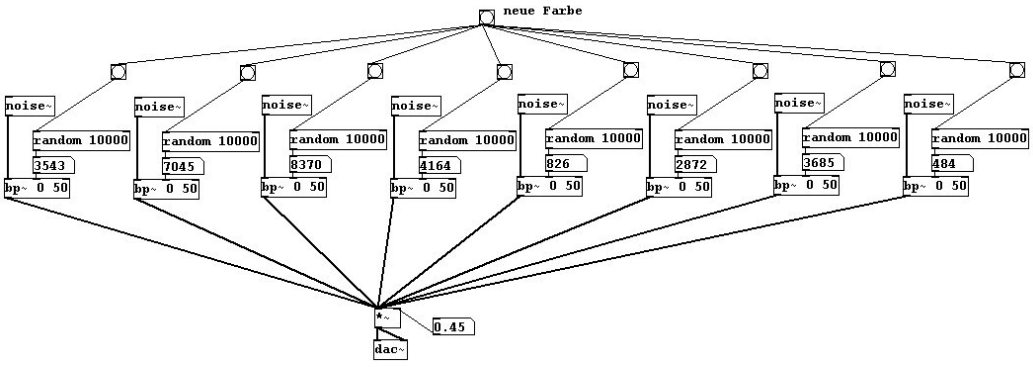
Wie man allerdings unschwer hören kann, gelingt dies nicht. Ein gewisser Rauschanteil bleibt bei dem Bandpassfilter immer.

3.3.2 Anwendungen

3.3.2.1 Filter-Farben

Als nur ein Beispiel der freien Anwendung von Filtern hier eine zufällige Verteilung von Bandpassfiltern:

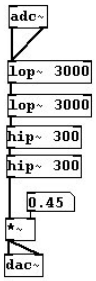
patches/3-3-2-1-filterfarben.pd



3.3.2.2 Telefon-Filter

Für die Übertragung von Telefongesprächen hat man ermittelt, dass zur Verständigung ein Ausschnitt zwischen 300 und 3000 Hz genügt. Wir können das also nachbauen:

patches/3-3-2-2-telefonfilter.pd



3.3.2.3 Weitere Aufgabenstellungen

Experimentieren Sie mit den Filterungen des „Glissandoorchesters“ (3.1.2.2.4).

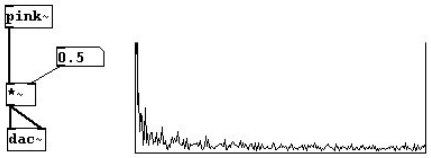
3.3.3 Appendix

3.3.3.1 Weißes Rauschen und Klicks

Bei „noise~“ bewegt sich die Membran per Zufall hin und her; beim Ein- und Ausschalten hören wir keinen Klick, denn Noise besteht nur aus mehr oder minder starken Klicks. Wir benötigen hierbei keine „Rampe“ (vgl. 3.1.2.1.2).

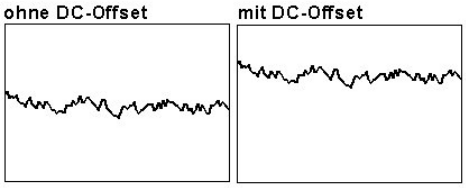
3.3.3.2 Rosa Rauschen

Neben dem weißen Rauschen gibt es auch das „rosa“ Rauschen. Das menschliche Ohr hört nicht alle Frequenzbereiche gleich laut. Am besten hört es um 2000 Hz herum, darum klingt weißes Rauschen auch eher hoch. In der Tiefe und in der Höhe hören wir wesentlich schlechter. Will man ein Rauschen erzeugen, das der Mensch tatsächlich als gleichmäßiges Summe aller Frequenzen hört, muss dies dem Ohr angepasst werden, müssen vor allem die tiefen Frequenzen also maßgeblich lauter sein als die mittleren. Diese Verteilung nennt man rosa Rauschen und kann in Pd mit „pink~“ erzeugt werden:

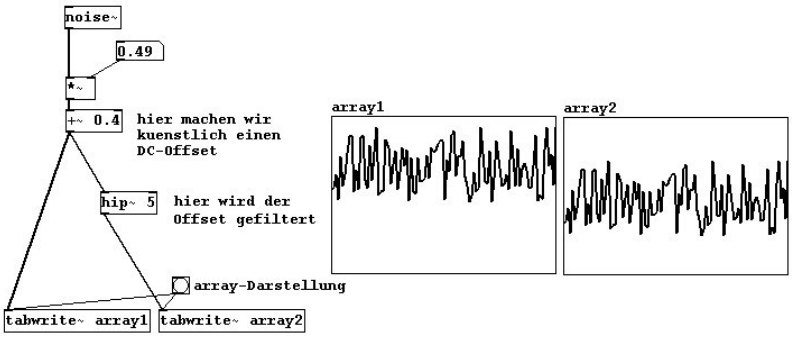


3.3.3.3 DC Offset

Bei einer Mikrofoneingabe passiert es häufig, dass eine Grundspannung zum Signal hinzukommt. Man nennt dies „DC-Offset“. Das Resultat ist diese Wellenform:



Dieser Offset entspricht quasi einer unendlich langsamen Welle mit einer gegen 0 gehenden Frequenz, die daher einfach mit einem sehr tief eingestellten Highpassfilter ausgemerzt werden kann:

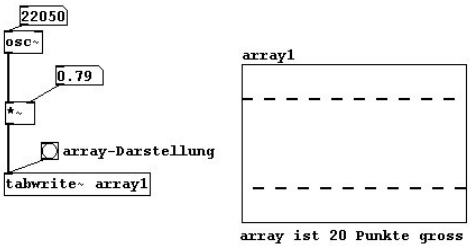


3.3.4 Für besonders Interessierte

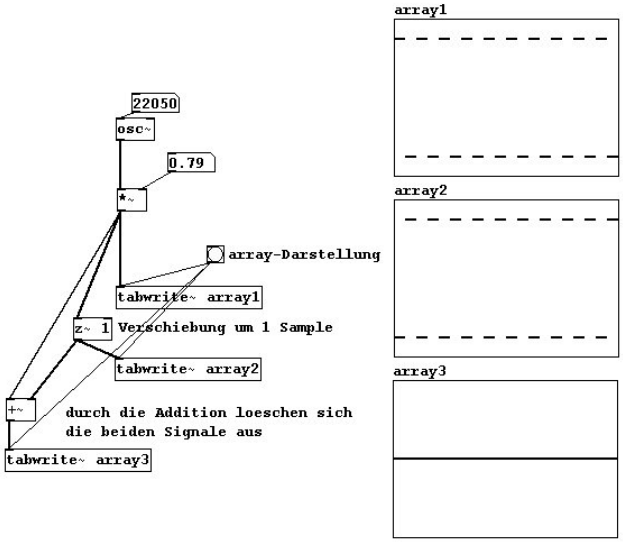
3.3.4.1 Arbeitsweise digitaler Filter

Das 'Innenleben' der digitalen Filter ist kompliziert. Eine Ahnung ihrer Technik soll dennoch gegeben werden: Wie unter 3.1.1.3.1 beschrieben wurde, kann bei einer Samplerate von 44100 Hz maximal die Welle von 22050 Hz erstellt werden. Bei dieser Welle gibt es nur noch zwei Punkte pro Periode:

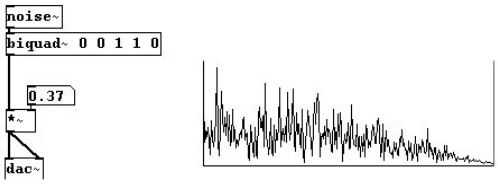
patches/3-3-4-1-filterarbeit.pd



Wenn man nun diese Welle einfach um eine Position, also um ein Sample nach vorne oder hinten verschiebt und zu der ursprünglichen addiert, löschen sie einander komplett aus. Diese Verschiebung funktioniert mit „z~“ (Pd-extended).

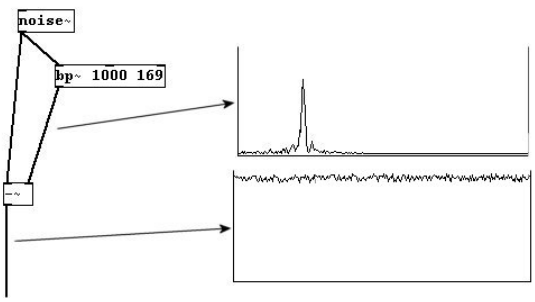


Nach dieser Idee, der sample-weisen Verschiebung (Delay) und Addition, verfahren digitale Filter. Bei dem „biquad~“-Filter kann man dies von Hand einstellen. Er führt folgende Differenzfunktion durch: y(n) = ff1 * w(n) + ff2 * w(n - 1) + ff3 * w(n - 2) mit w[n] = x[n] + fb1 * x[n - 1] + fb2 * x[n - 2]. n ist hierbei die Sampleposition und ff1, ff2, ff3, fb1 und fb2 frei zu wählende Faktoren. In Pd erhält „biquad~“ entsprechend fünf Argumente für ff1, ff2, ff3, fb1 und fb2. Seine Syntax lautet: „biquad~“ [fb1] [fb2] [ff1] [ff2] [ff3]. Für den eingangs aufgeführten Fall der Welle von 22050 Hertz könnte wir also „biquad~ 0 0 1 1 0“ schreiben. Damit werden hohe Frequenzen unterdrückt, am stärksten die Welle mit 22050 Hertz, die komplett ausgelöscht wird. Folglich also ein Lowpassfilter:



Mit der Biquad-Formel können viele weitere Filterformen erstellt werden. Zum Beispiel haben wir mit den Argumenten 1.41407 -0.9998 1 -1.41421 1 eine „Bandsperre“, die Umkehrung eines Bandpassfilters, die an einer bestimmten Stelle keine Frequenzen durchlässt, in diesem Fall bei 5512.5 Hz. Die Erklärung für diese Rechenwege würden aber ein eigenes Buch füllen. In Pd-extended gibt es Objekte („bandpass“, „equalizer“, „highpass“, „highshelf“, „hlshelf“, „lowpass“, „lowshelf“, „notch“), die diese Rechnungen durchführen. Der Vorteil des Biquad-Filters ist, dass wesentlich steilere Filterungen möglich sind als mit „lop~“, „hip~“ und „bp~“. Der Nachteil ist, dass nicht nur manche Frequenzen unterdrückt werden, sondern andere dafür mitunter auch extrem verstärkt werden, bis hin zur „Explosion“ (an der Formel sieht man ja, dass der Filter rekursiv arbeitet).

Beim Biquad-Verfahren konnte man auch sehen, dass Filter Phasenverschiebungen vornehmen. Darum kann z. B. ein „bp~“ nicht einfach zu einem Kerbfilter umgekehrt werden:



3.4 Sampling

3.4.1 Theorie

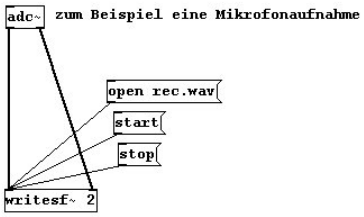
Zunächst muss eine unselige Vermischung besprochen werden: Wir schon gesagt ist ein Sample die kleinste Einheit der Messung und Generierung von Klang im Computer. Es gibt jedoch noch einen anderen Gebrauch des Wortes Sample, der ein kleineres Stück (von ein paar Sekunden Länge) aufgenommenen Klangs meint. Um die Bearbeitung kurzer aufgenommener Klänge geht es in diesem Kapitel. Dafür muss jedoch zuerst der „array“ in Pd erklärt werden, was einigen Platz beansprucht.

3.4.1.1 Speicherung von Klang

3.4.1.1.1 Klangdateien

Im Computer gibt es zwei verschiedene Orte, an denen etwas gespeichert werden kann: der Hauptspeicher und die Festplatte(n). Der Zugriff auf den Hauptspeicher ist sehr schnell im Vergleich zur Festplatte. Dafür ist dort nicht so viel Platz.

In Pd kann man an beiden Orten Klang speichern. Wird auf der Festplatte gespeichert, heißt das, dass man eine fixierte Klangdatei speichert. Übliche Formate sind wav oder aiff. Man speichert mit „writesf~“. Als Argument gibt man die Anzahl der Kanäle, worauf entsprechende Inlets entstehen, an die man den aufzunehmenden Klang anschließt. Zunächst nennt man mit der Message „open [name]“ den Namen der zu erstellenden Datei. Die Aufnahme wird dann mit „start“ gestartet und mit „stop“ gestoppt.

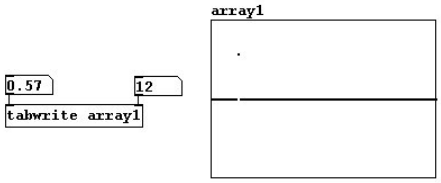


3.4.1.1.2 Buffers

Der erwähnte andere Ort ist der Hauptspeicher. Einen Platz für einen Klang erstellen wir mit dem „array“ (**Put # Array**, dann auf „ok“ klicken). Er ist zugleich eine Visualisierung von Klang.

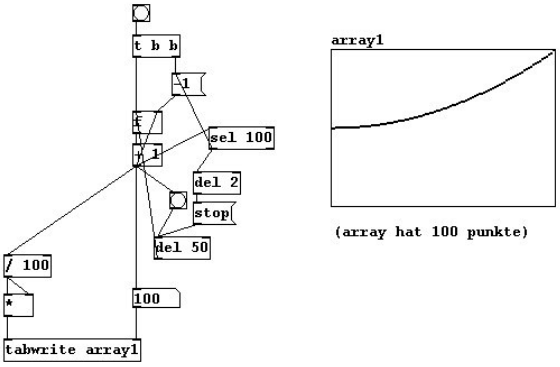
Zunächst betrachten wir den Array aber ganz allgemein als Speicher von Zahlen. Ein Array hat eine begrenzte Zahl von Speicherplätzen. Die Anzahl kann man einstellen, wenn man mit der rechten Maustaste auf den Array klickt und in die „Properties“ geht. Darauf öffnen sich zwei Fenster, eines mit „array“ überschrieben und eines mit „canvas“. Im „array“-Fenster können wir die „size“, also Größe einstellen. Diese Zahl meint die Anzahl der Speicherplätze. In jedem Speicherplatz kann eine Zahl deponiert werden.

Diese Plätze belegen wir nun mit „tabwrite“. Das rechte Argument bestimmt den Platz, das linke dann (wie immer: von rechts nach links) den zu speichernden Wert. In dem Array wird das auf der x- (Platz) und y- (Wert) Achse angezeigt:



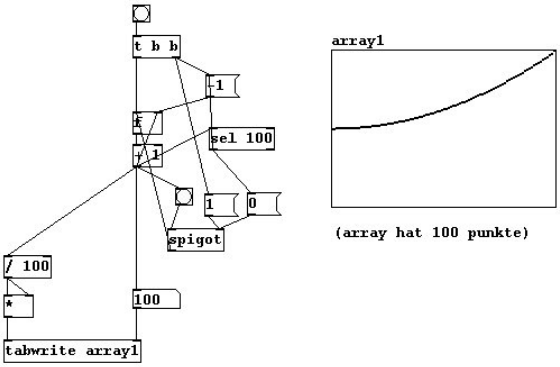
So können wir zum Beispiel Funktionen darstellen:

patches/3-4-1-1-2-funktion1.pd

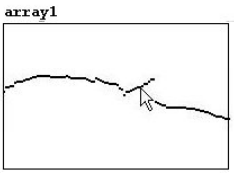


Oder ohne zeitliche Ausdehnung (falls das Beispiel einen stack overflow erzeugen sollte, hilft ein „del 0“ zwischen „spigot“ und „f“):

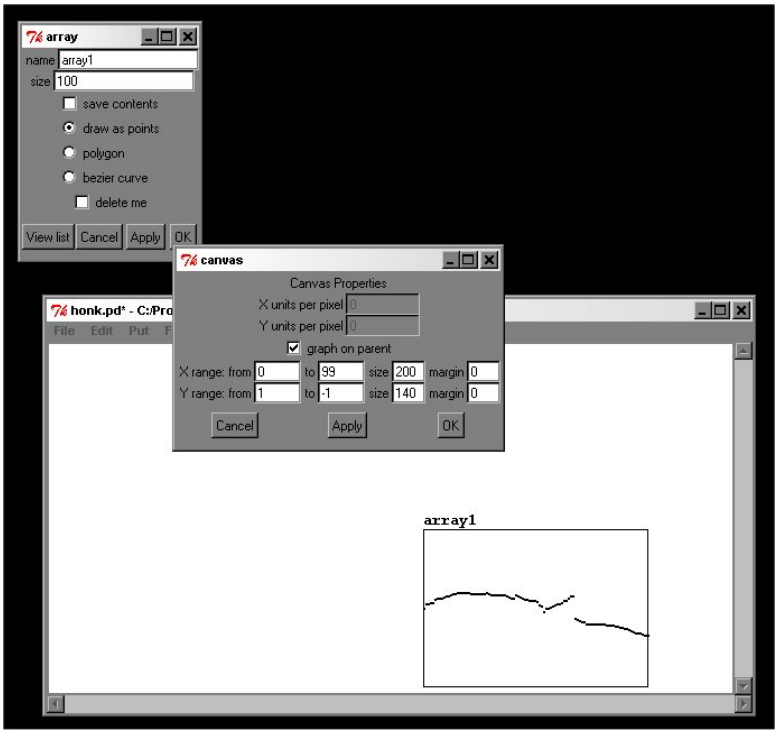
patches/3-4-1-1-2-funktion2.pd



Mann kann aber auch quasi von Hand etwas mit der Maus in den Array zeichnen. Bewegt man die Maus auf einen Wert innerhalb des Arrays, ändert sich die Pfeilrichtung des Cursors, so dass man bei gedrückter Maustaste eigene Linien ziehen kann.



In den Properties (Rechtsklick auf den array) lässt sich Folgendes einstellen:



Im „array“-Fenster:

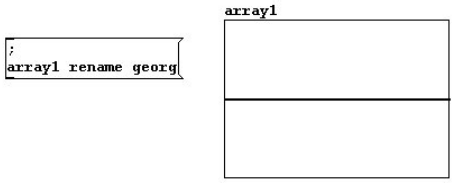
- Name: Wie bei allen Namen in Pd ohne Leerzeichen und als alphanumerische Zeichenfolge.
- Größe: Wie schon beschrieben.
- „save contents“: Mit Setzung eines Hakens werden alle Werte des Arrays im Patch gespeichert. Dies kann aber später, bei vielen / großen Arrays, sehr viel Platz auf der Festplatte beanspruchen und verlängert die Ladezeit des Patches enorm.
- „draw as points“ / „polygon“ / „bezier curve“: Verschiedene Arten der Visualisierung – ob bloße Punkte oder zu Linien zusammengefasst.
- „delete me“: Mit Setzung eines Hakens ist der Array daraufhin gelöscht! Allerdings wird der Kasten unnötigerweise noch angezeigt und muss noch eigens gelöscht werden.
- „view list“: Hier werden alle Werte in einer Liste dargestellt.

Im „canvas“-Fenster:

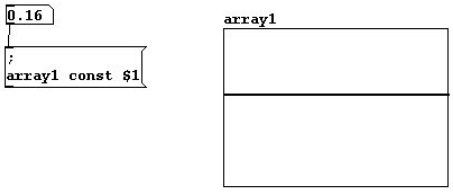
- „graph on parent“: Wird erst unter 3.1.1.2 besprochen.
- Xrange: Hier können die Anfangs- und Schlusswerte der x-Achse bestimmt werden.
- YRange: Hier wird der Darstellungsbereich für die y-Achse eingestellt. Werte außerhalb dieses Rahmens werden aber auch gespeichert, nur platzausweitend dargestellt.
- „size“: Visuelle Größe im Patch.

Weiterhin kann ein Array Messages bzw. „sends“ empfangen.

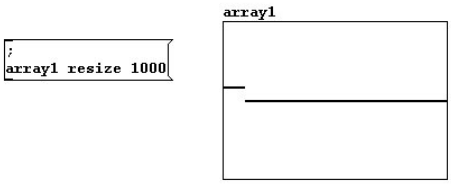
Einen neuen Namen geben:



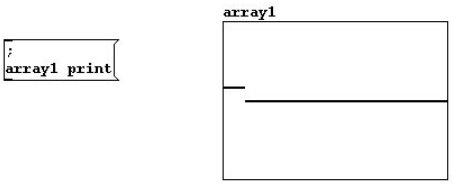
Alle Werte gleichsetzen:



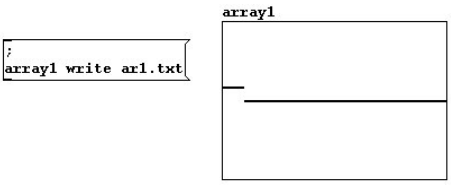
Die Größe ändern:



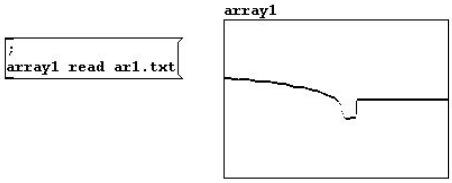
Die Größe kann auch ausgedruckt werden:



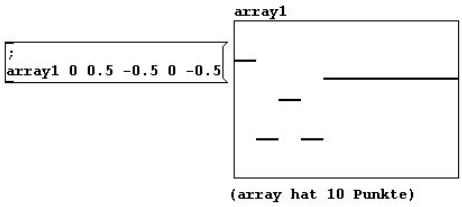
Den Inhalt in eine Textdatei schreiben:



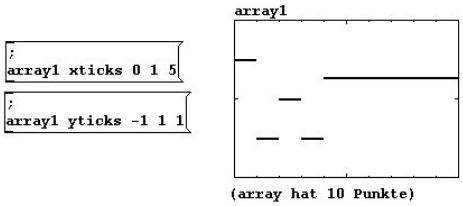
Eine Textdatei lesen:



Wir können so auch Werte eingeben; die erste Zahl bestimmt, an welchem Speicherplatz begonnen werden soll, alle weiteren sind Werte für die darauf folgenden Plätze:

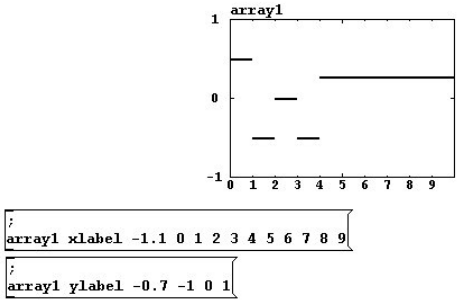


Zusätzlich können wir noch Raster an die Achsen anlegen und diese beschriften:



Die Argumente benennen 1. den Startplatz auf der Achse, 2. den Abstand zwischen den Strichen und 3. den Abstand zwischen größeren Strichen.

Und Bezifferung:

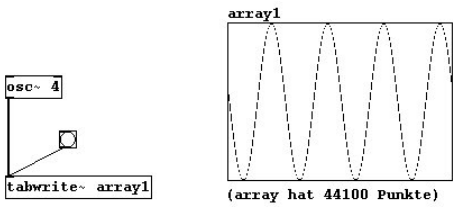


Beschriftet wird 1. die Position, 2. die anzuzeigenden Ziffern.

Man beachte: Striche und Beschriftung werden nicht im Patch gespeichert, müssen also bei jedem neuen Aufruf des Patches wieder erneut angebracht werden.

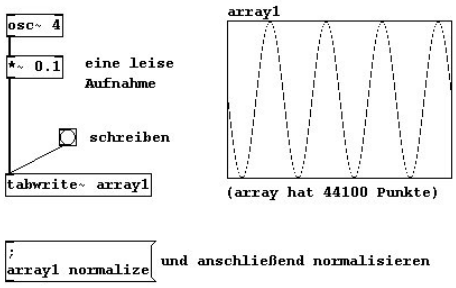
Nun können wir aber auch Klang in einem Array speichern. Klang ist ja, wie schon häufig betont, im Computer nichts als eine Folge von Zahlen, genauer gesagt 44100 Zahlen pro Sekunde. Wir können beispielsweise eine Sekunde Klang in einem Array speichern, nur muss dieser dafür auch 44100 Speicherplätze haben. Nun benötigen wir das Objekt „tabwrite~“. Es erhält den Klang-Input und ansonsten nur einen Bang. Anders als bei „tabwrite“ (ohne Tilde), wo wir rechts von Hand den Speicherplatz eingegeben haben und links den zugehörigen Wert, wird hier nun, wenn es einen „bang“ bekommt, automatisch beim ersten Speicherplatz angefangen und dann in Samplegeschwindigkeit weitergeschritten. Gleichzeitig wird jedem Platz ein von links kommender Wert des ablaufenden Klanges zugewiesen; damit werden insgesamt 44100 Zahlen gespeichert. Was danach an Klang kommt, wird nicht mehr gespeichert. Will man vorzeitig abbrechen, schickt man die Message „stop“.

patches/3-4-1-1-2-normalisieren.pd



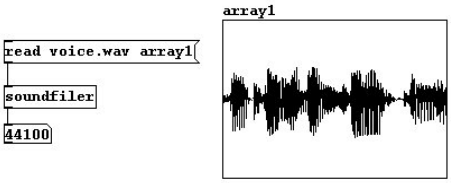
Erhält „tabwrite~“ eine float message, so wird diese Zahl als Sampleoffset interpretiert d.h. mit diesem Sample beginnend in den Array geschrieben.

Eine nützliche Funktion ist, dass man im Nachhinein die allgemeine Lautstärke auf das maximal Mögliche anheben kann. Ist eine Aufnahme zum Beispiel sehr leise gewesen, d. h. die Membran des Aufnahmemikrofons schwang nicht sonderlich stark, erhalten wir auch nur sehr kleine Werte. Die können wir aber quasi aufblasen bzw. „normalisieren“. Dafür gibt es folgende Message:



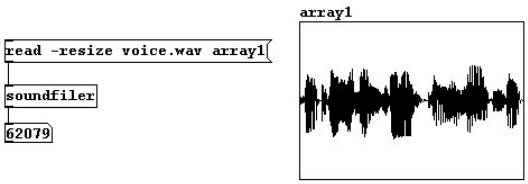
Wir können nun auch eine Verbindung zwischen Klangdateien auf der Festplatte und Klang im Hauptspeicher herstellen, also im Array. Dies erfolgt mit dem Objekt „soundfiler“. Damit können wir eine auf der Festplatte gespeicherte Klangdatei in einen Array laden, oder den Inhalt eines Arrays auf der Festplatte als Klangdatei speichern. Das Laden geschieht mit dem Befehl „read“. Die Argumente des Befehls sind der Name der Datei (ggf. mit Verzeichnispfad) und dahinter der Name des Arrays, in den diese geschrieben werden soll.

patches/3-4-1-1-2-soundfile-laden.pd



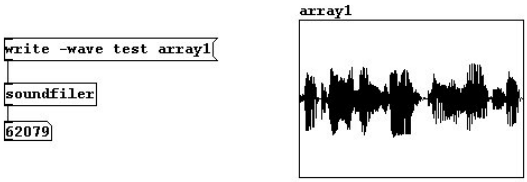
Nach dem erfolgreichen Laden der Datei wird die Größe der Klangdatei in Samples aus dem Outlet von „soundfiler“ ausgegeben.

Wir können noch zusätzliche Befehle („flags“ genannt) in die Message integrieren:



Durch den resize-Befehl wird die Größe des Arrays auf die genaue Größe der Klangdatei gebracht (wobei da bei 4000000 Samples, das sind ca. 90 Sekunden, die Grenze liegt, was zusätzlich mit „maxsize“ aber geändert werden kann).

Umgekehrt speichert der Befehl „write“ den Inhalt eines Arrays als Klangdatei auf der Festplatte. In diesem Fall muss das Format (wav oder aiff) als Flag angegeben werden, anschließendn der Name (ggf. mit Pfad) der zu schreibenden Datei und dahinter der Name des Arrays.

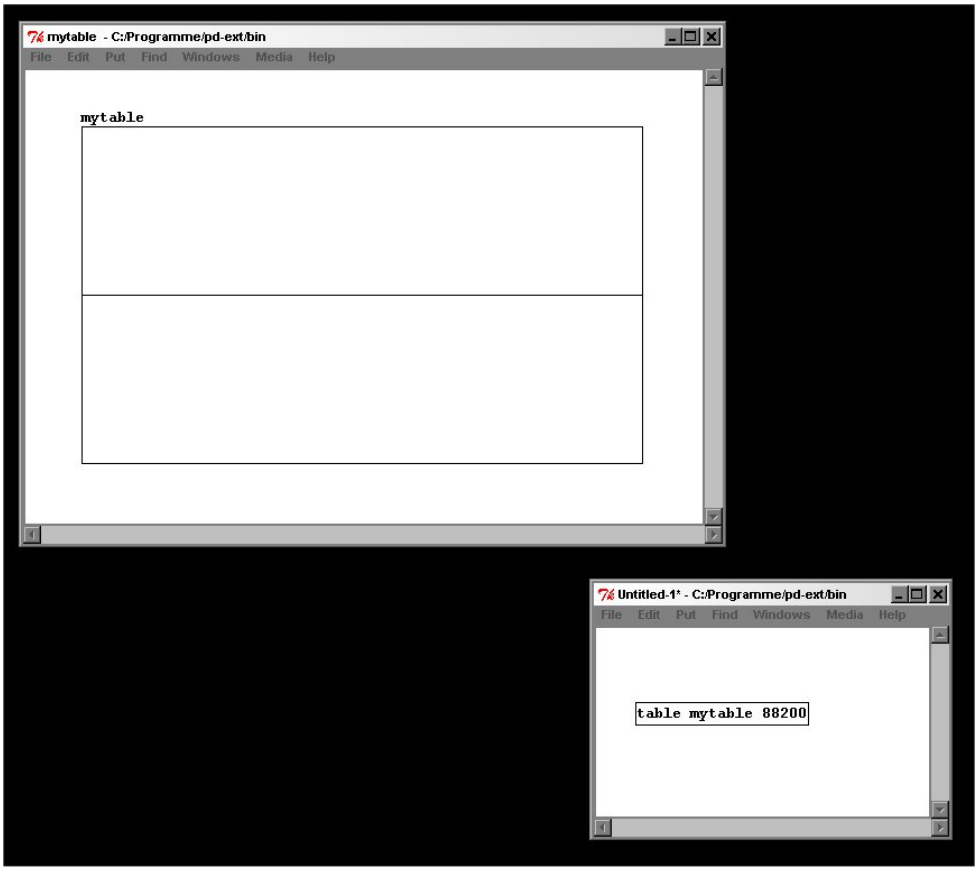


Weitere „Flags“ hierzu sind:

- normalize: Bringt die Datei auf optimale Amplitudenaussteuerung, wie vorhin beschrieben.

- rate: Damit kann die Samplerate für die Datei eingestellt werden.

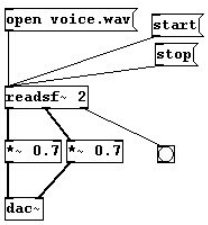
Alternativ zum Array gibt es auch den „table“. Man erstellt das Objekt „table“ und gibt als erstes Argument den Namen, als zweites die Größe in Samples. Daraufhin wird in einem Subpatch (im Execute Mode auf das Objekt klicken) ein Array erstellt, der wie ein normaler Array behandelt wird. Das hat folgenden Vorteil: Die Grafik eines normalen Arrays ist ab einer gewissen Größe sehr aufwändig; man merkt das, wenn man den Array auf dem Canvas verschiebt – es geht nur sehr langsam. Verbleibt die grafische Repräsentation des Arrays aber in einem Subpatch, lässt sich mit dem Objekt leichter hantieren.



3.4.1.2 Wiedergabe von gespeichertem Klang

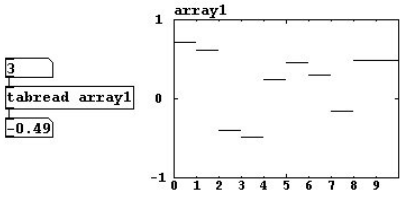
Klangdateien, die sich auf einem externen Speichermedium wie beispielsweise einer Festplatte befinden, können in Pd mit „readsf~“ gelesen, das heißt abgespielt werden. Wie bei „writesf~“ verwendet man die Messages „start“ und „stop“ (es geht auch „1“ und „0“). Als Argument gibt man dazu die Anzahl der Kanäle. Der Output ganz rechts macht einen Bang, wenn die Datei zu Ende gelesen ist.

patches/3-4-1-2-datei-spielen.pd



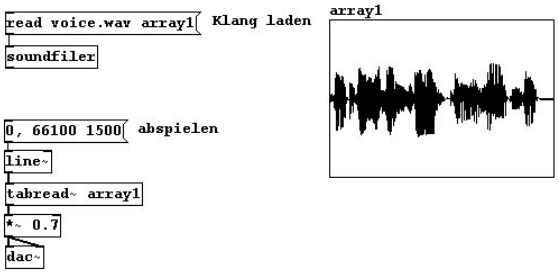
Für den Array betrachten wir zunächst wieder nur die Kontroll-Ebene: Haben wir zum Beispiel einen Array mit 10 Speicherplätzen, können wir mit „tabread“ jeden dieser Plätze einzeln abfragen:

patches/3-4-1-2-array-lesen1.pd



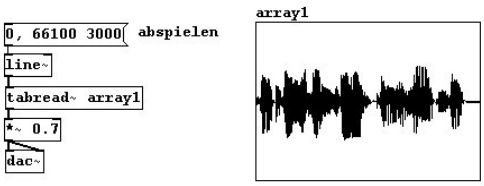
Das Prinzip ist nun für Signale im Grunde dasselbe, nur müssen die gespeicherten Werte nun in der Geschwindigkeit von 44100 Zahlen pro Sekunde erhalten werden. Dafür gibt es „tabread~“. Wollen wir beispielsweise einen in einem Array gespeicherten Klang von 1,5 Sekunden (= 66150 Samples) Dauer abspielen, heißt das, wir müssen hintereinander die Arraywerte 0 bis 66149 abfragen, und zwar in der Geschwindigkeit von 44100 Werten pro Sekunde. Dies machen wir mit „line~“:

patches/3-4-1-2-array-lesen2.pd



„tabread~“ erhält als Argument den Arraynamen. Wir können den zu lesenden Array aber auch mit einer Message „set [arrayname]“ bestimmen.

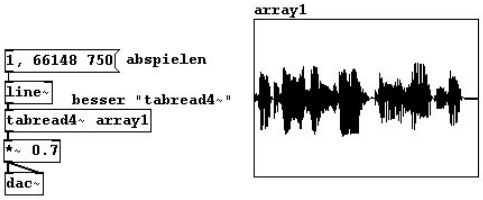
Nun können wir aber auch anfangen mit diesen Werten zu spielen. Etwa indem wir die Reihe einfach in der doppelten Zeit ablaufen lassen:



Dadurch wird die Abspielgeschwindigkeit halbiert. Dies führt dazu, dass alles eine Oktave tiefer klingt, denn durch die zeitliche Streckung werden alle Wellen doppelt so lang, das heißt in der Frequenz halbiert und damit eben eine Oktav tiefer.

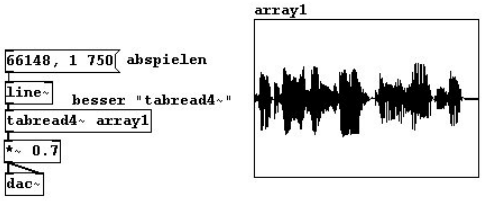
Hier gibt es nun aber das Problem, dass ein vorhandenes Sample zweimal hintereinander gespielt wird oder es zu Auslassungen kommt. Um diese Lücken zu schließen, gibt es eine modifizierte Form des „tabread~“-Objekts, „tabread4~“, das zwischen diesen Lücken interpoliert, also Zwischenwerte aus den Informationen des davor und danach liegenden Wertes erzeugt. (Mehr zu der Funktionsweise für besonders Interessierte unter 3.4.4.) In den meisten Fällen ist daher zum Auslesen von Arrays „tabread4~“ geeigneter. Dieses braucht ein Auslesespektrum von 1 bis n - 2, wobei n die Größe des zu lesenen Arrays ist.

Wir können natürlich auch etwas schneller und damit höher abspielen:

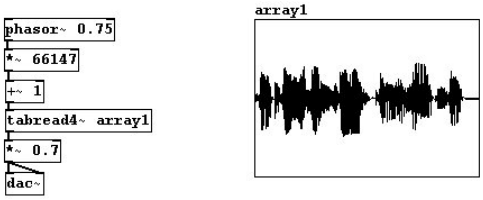


Wir werden später mit Hilfe der Granularsynthese Wege kennenlernen, wie wir Tempo und Tonhöhe unabhängig voneinander verändern können.

Außerdem liegt es natürlich noch nahe, etwas rückwärts abzuspielen:



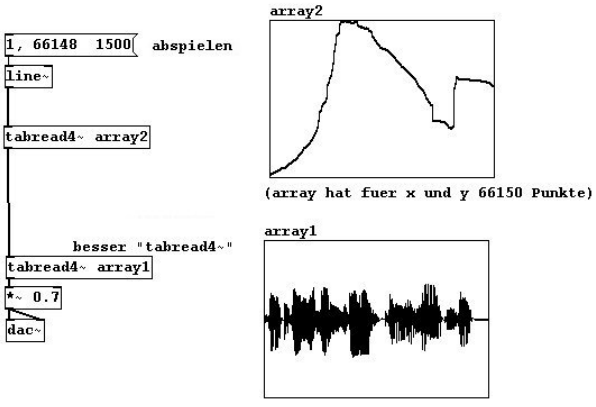
Man kann sich nun aber auch des „phasor~“-Objekts bedienen:



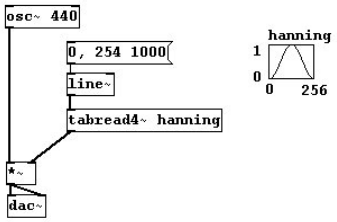
„phasor~“ erzeugt eine Reihe von Zahlen von 0 bis 1 als Signal. Multiplizieren wir diese Werte mit 66148, haben wir eine Reihe von 0 bis 66148. Als Frequenz geben wir 0.75 an, so dass die Reihe in genau 1,5 Sekunden abläuft.

Zudem könnte man auch einen eigenen Array für die Ausleseweise arrangieren, mit dem man den ersten Array abspielt:

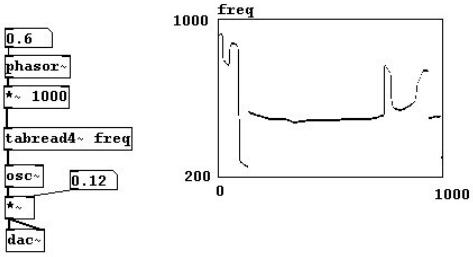
patches/3-4-1-2-array-lesen3.pd



Oder wir können für den Amplitudenverlauf einen Array verwenden:



Oder einen Array für einen Frequenzverlauf:



Erneut ist zu sehen: Wir haben es nur mit Zahlen zu tun, die verschiedentlich zur Steuerung eingesetzt werden können.

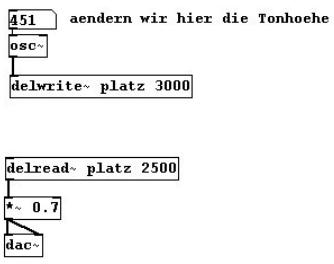
Man kann mit beliebig vielen „tabread~“-Objekten gleichzeitig aus einem Array lesen. Allerdings sollte man nie zwei Arrays mit dem gleichen Namen verwenden, das führt fast zwangsläufig zu Fehlern.

3.4.1.3 Audiodelay

In Kapitel 2.2.3.1.2 haben wir die Möglichkeit besprochen, Zahlen oder Zahlenfolgen zeitlich zu verzögern („delay“). Das können wir auch mit Signalen machen. Hierfür wird gleichfalls ein Buffer erstellt, in den die Signale zunächst hineingeschrieben werden und aus dem dann zeitverzögert wieder gelesen wird. Diesen Buffer erstellen wir mit „delwrite~“; als erstes Argument geben wir dem Buffer einen beliebigen Namen, als zweites Argument die Größe in Millisekunden. Geben wir dem nun einen Signal-Input, wird dies in den Buffer geschrieben. Ist der Buffer voll, wird vom Anfang an überschrieben. Ist der Buffer also 1000 Millisekunden groß, sind immerzu die letzten 1000 Millisekunden des eingehenden Signals darin gespeichert.

Gelesen wird auf dem Buffer dann mit „delread~“. Als erstes Argument erscheint wieder der Buffername, als zweites die Zeit der Verzögerung (in Millisekunden; kann durch den Input als Kontrolldateneingabe geändert werden):

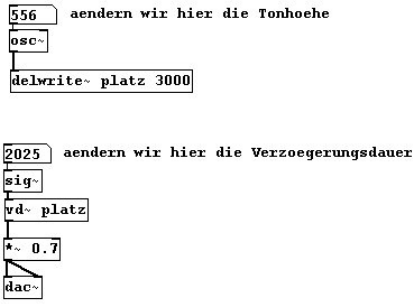
patches/3-4-1-3-delay.pd



Logischerweise muss die Verzögerungszeit in „delread~“ kleiner als die Buffergröße sein oder zumindest gleich groß. Wenn etwas 2000 Millisekunden später gespielt werden soll, aber in dem

Buffer nur für 1000 Millisekunden verbleibt, kann es nicht funktionieren. Ebenso ist eine negative Zahl als Verzögerung natürlich Unsinn, denn in die Zukunft kann Pd nicht schauen. Man kann aus einem Delay-Buffer mit beliebig vielen „delread~“-Objekten gleichzeitig lesen. Man kann nicht in den Wellenverlauf dieses Buffers hineinsehen.

Man kann zwar in „delread~“ die Verzögerungszeit verändern, aber nur als Kontrolldateneingabe und mit der entsprechenden Fehlerwahrscheinlichkeit ab einer bestimmten Geschwindigkeit (das ist wieder der Konflikt zwischen Kontrolldaten und Signalen). Darum gibt es noch ein eigenes Objekt zum variablen Lesen von Delay-Buffern, „vd~“ (Abk. für „variable delay“, also variable Verzögerung). Hier gibt man die Verzögerungszeit (in Millisekunden) als Signal ein, und kann diese permanent verändern (natürlich wiederum nicht unter 0 und nicht oberhalb der Buffergröße):

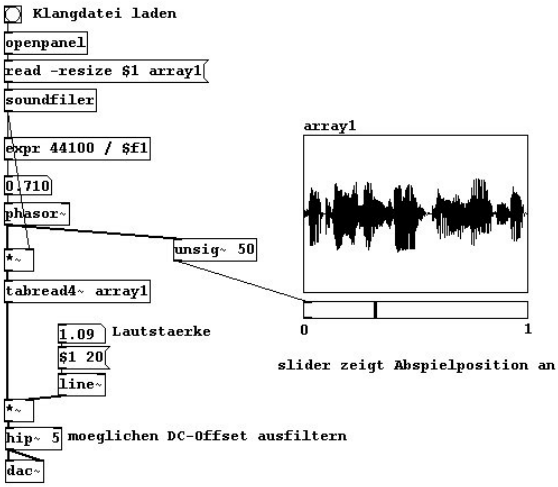


„vd~“ erzeugt wie „readsf4~“ eine Interpolation.

3.4.2 Anwendungen

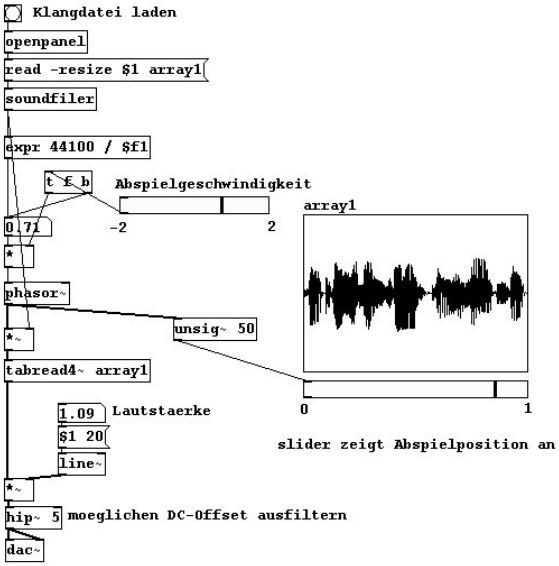
3.4.2.1 Ein einfacher Sampler

patches/3-4-2-1-einfacher-sampler.pd



3.4.2.2 Mit variabler Geschwindigkeit

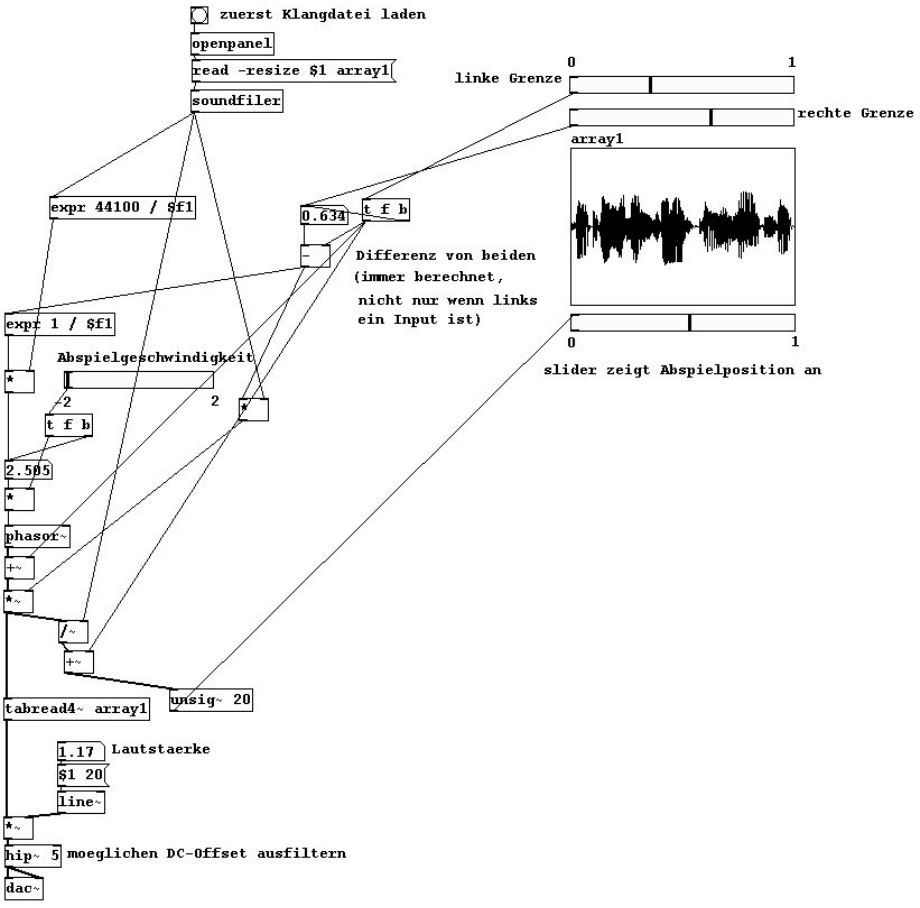
patches/3-4-2-2-sampler2.pd



3.4.2.3 Beliebige Positionen

Picken wir uns nun beliebige Positionen aus einem Sample heraus:

patches/3-4-2-3-sampler3.pd



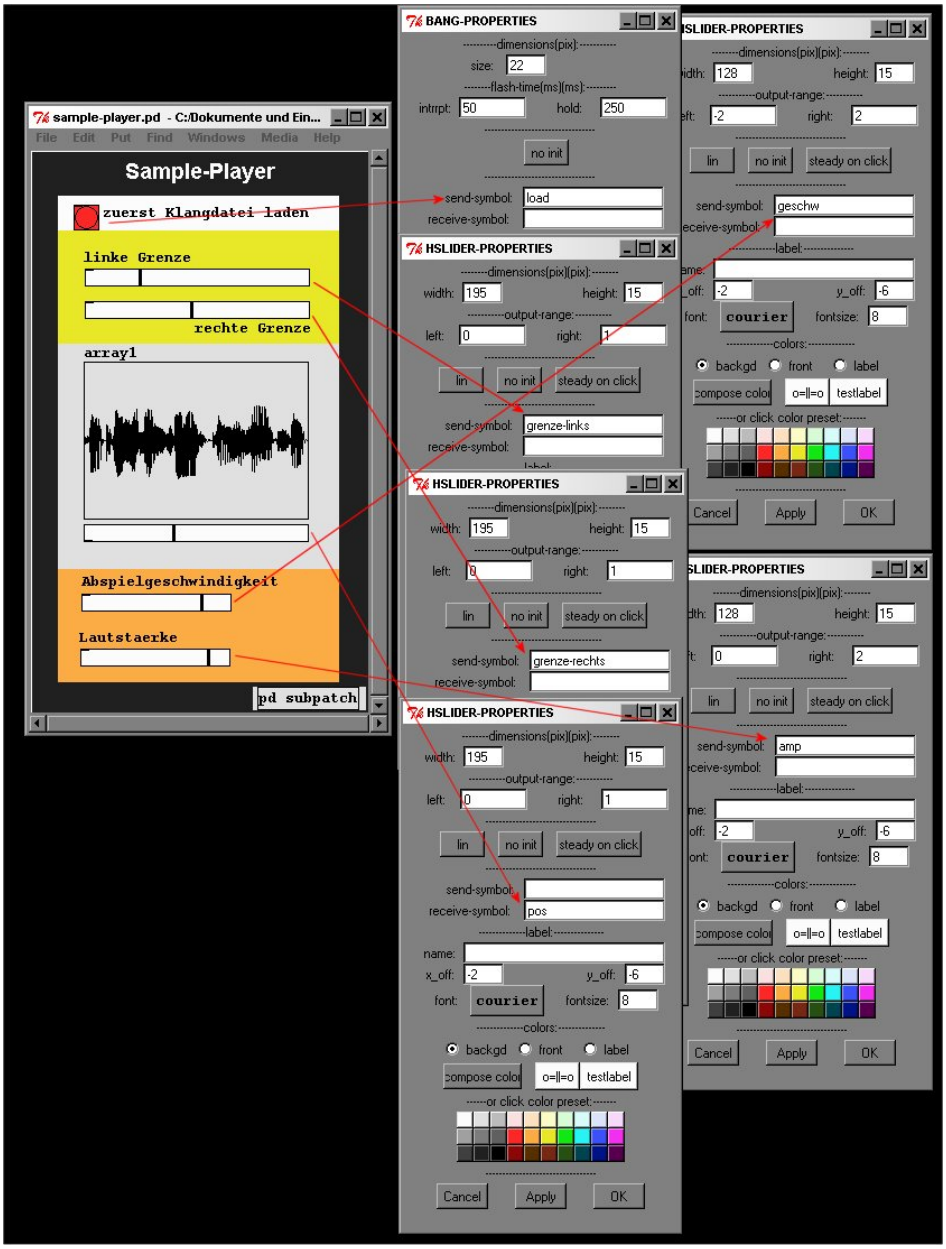
3.4.2.4 Sampler zusammengefasst

Wendet man die oben beschriebenen Funktionen zur optischen Verbesserung (2.2.4) an, dann sieht das Ganze beispielsweise so aus:

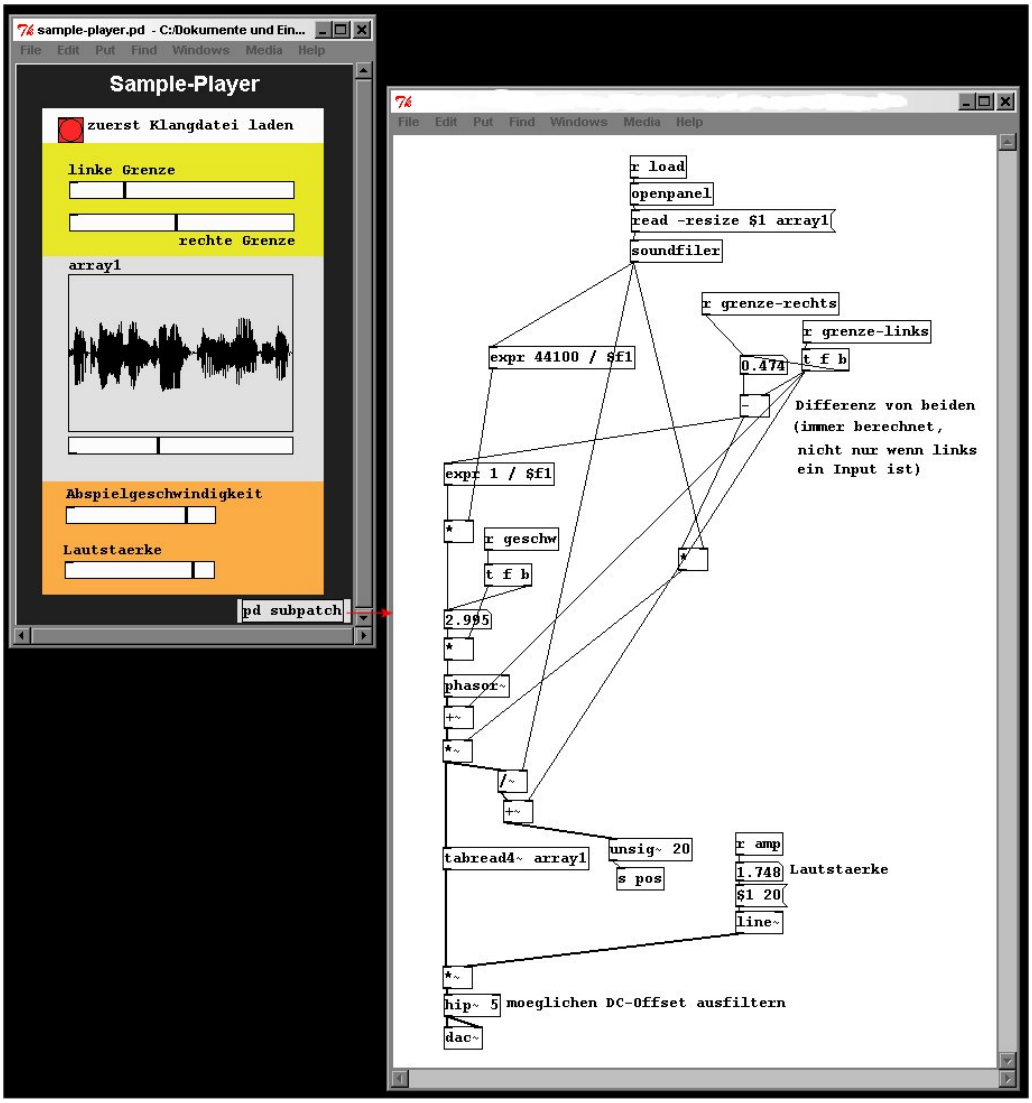
patches/3-4-2-4-sampler-gross.pd

Vier Canvas-Objekte bilden den farbigen Hintergrund der Slider und der Arraydarstellung. Man beachte: Es erscheinen immer die Grafiken, die zeitlich zuletzt erstellt wurden, über den anderen. Haben wir also z. B. zuerst den „array1“ und erstellen dann ein farbiges Canvas-Objekt, müssen wir den „array1“ noch einmal neu erstellen (einfach den alten kopieren und dann löschen), so dass er vor dem Canvas-Objekt erscheint.

Zur Erklärung, was jeweils dahinter steckt:

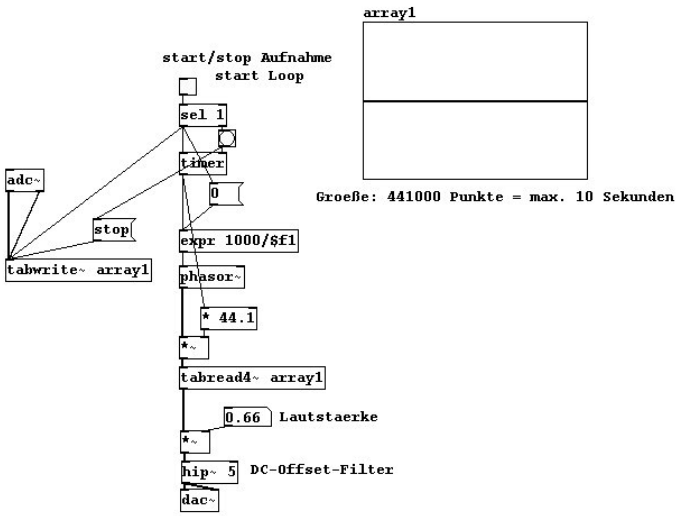


Und im Subpatch:

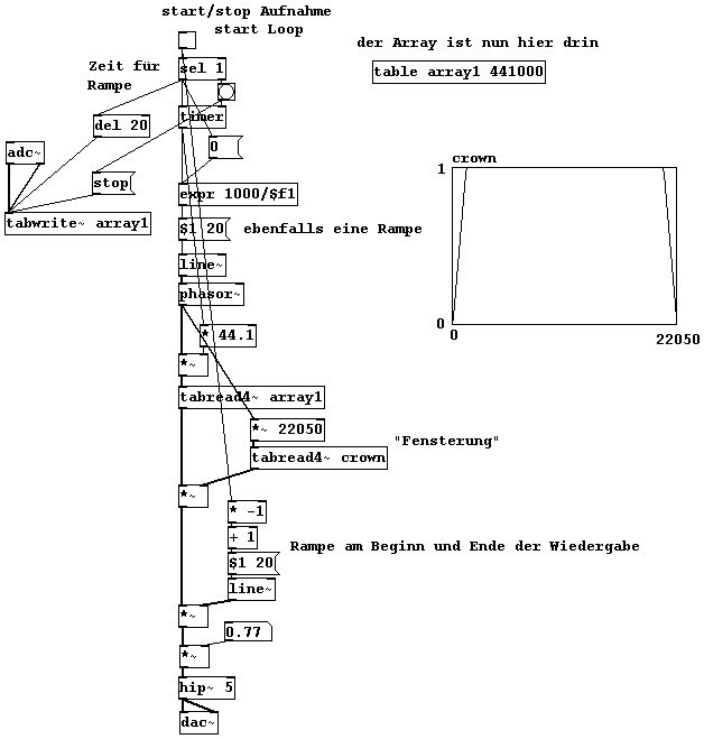


3.4.2.5 Loop-Generator

patches/3-4-2-5-loop-generator1.pd

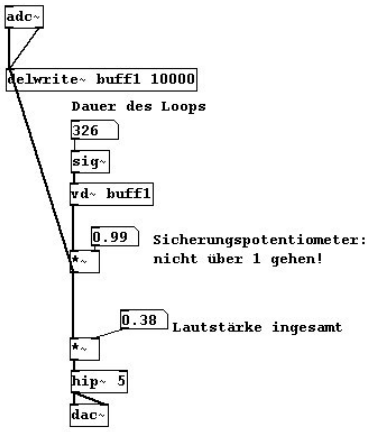


Bei diesem Loop-Generator kommt es allerdings immer wieder zu Klicks. Zum einen ist sehr zu empfehlen, den Array in ein Unterfenster auszulagern, denn die grafische Darstellung erfordert relativ viel Rechenaufwand. Zum anderen sollten wir an den Rändern des Loops kurz zu 0 gehen, damit es bei ihrem Anschluss nicht zu einem plötzlichen Wertesprung (und damit zu einem Klick) kommt. Hierzu machen wir eine „Fensterung“. Synchron zum Auslesen des Arrays wird dies in der Amplitude bestimmt durch einen anderen Array (hier „crown“), der die dynamische Hüllkurve vorgibt. Diese Hüllkurve hat zu Beginn und am Ende den Wert Null, so dass es beim Umbruch keine plötzliche Werteänderung gibt. Man könnte statt des „crown“-Fensters aber auch ein anderes wie das „Hanning“-Fenster verwenden, das einen Ausschnitt aus der Sinus-Funktion verwendet (dazu kommen wir später wieder). Eigentlich sollte sich der „crown“-Array natürlich auch in einem Unterfenster befinden, der Anschaulichkeit halber haben wir das hier jedoch unterlassen.



Eine einfachere Version des Loops lässt sich mit einer Rückkopplungsschaltung (engl. „feedback“) erzeugen:

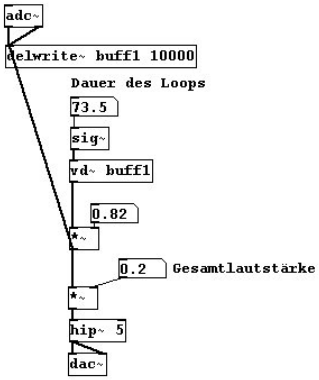
patches/3-4-2-5-loop-generator2.pd



Ein Nachteil ist hierbei, dass die maximale Loop-Dauer vorgegeben ist; hier beträgt sie 10000 Millisekunden.

3.4.2.6 Reverb

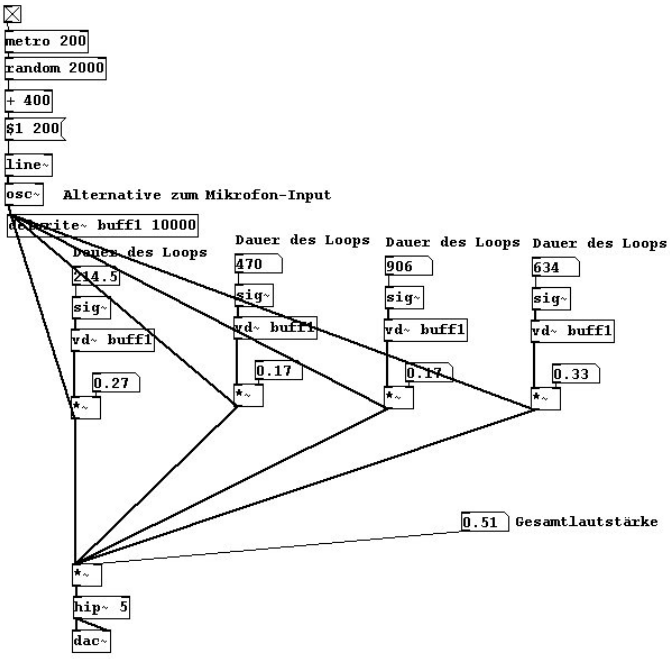
So kann man zum Beispiel eine „Hall“-Wirkung (engl. „Reverb“) erzielen, wenn das rückgekoppelte Signal immer leiser wird:



3.4.2.7 Textur

Oder man kann eine Textur erstellen:

patches/3-4-2-7-textur.pd

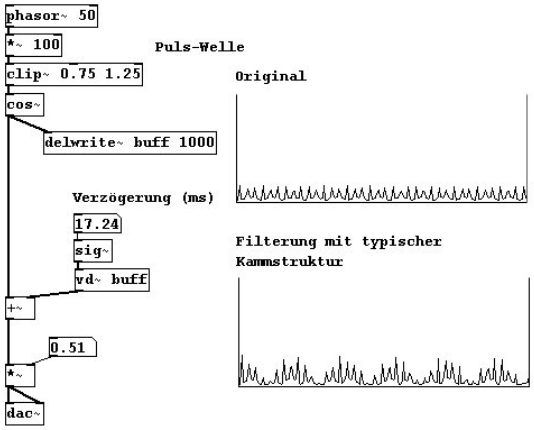


Hier muss man allerdings sehr aufpassen, dass die Rückkopplung nicht "explodiert", also an Lautstärke (exponentiell) zunimmt.

3.4.2.8 Kammfilter

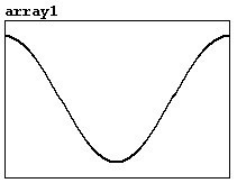
Mit dem Audiodelay kann man einen Kammfilter bauen. Die Idee ist, dass zum Originalsignal seine Verzögerung addiert wird. Dadurch enstehen Verstärkungen und Auslöschungen in regelmäßigen Abständen, was in der Spektraldarstellung aussieht wie ein Kamm:

patches/3-4-2-8-kammfilter.pd

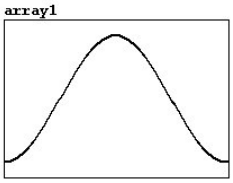


3.4.2.9 Oktavdoppler

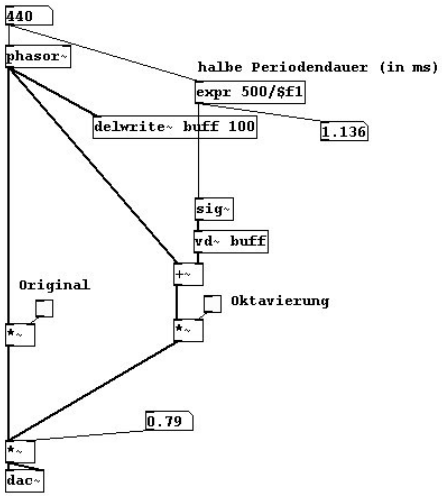
Sofern man die Frequenz des Grundtons eines Signals kennt, kann man nach folgender Idee einen Oktavdoppler konstruieren: Nehmen wir eine Welle ...



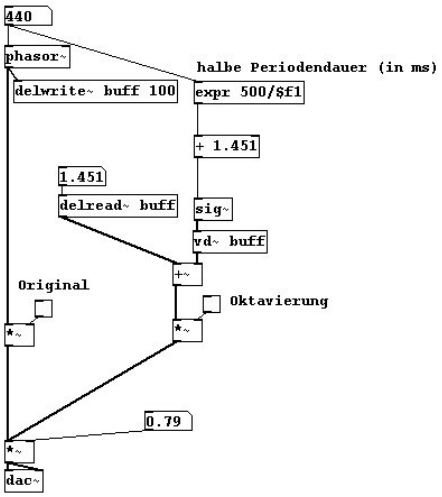
... und ihre Verzögerung um die Hälfte ihrer Periodenlänge ...



... ergibt ihre Addition 0. Wenn man also ein Signal um die Hälfte der Periodenlänge seines Grundtons verzögert und dies zum Original addiert, fällt der Grundton (und alle ungeraden Teiltöne) weg. Das könnte also so aussehen:



Das funktioniert jedoch so noch nicht. Wir müssen hierbei wieder in Betracht ziehen (vgl. 3.1.1.3.2), dass Pd alle Audio-Daten in Blöcken von 64 Samples (wenn nicht anders eingestellt) zusammgefasst abarbeitet (das ist effizienter als jedes Sample einzeln). In dem Fall hätten wir eine Verzögerung von 1.136 Millisekunden, das sind 50 Samples. Abhilfe können wir dem schaffen, wenn das Original auch noch einmal aus dem Buffer gelesen wird, mit Verzögerung eines Blocks (64 Samples = 1.451 ms), ebenso dies als Offset der Verzögerung:

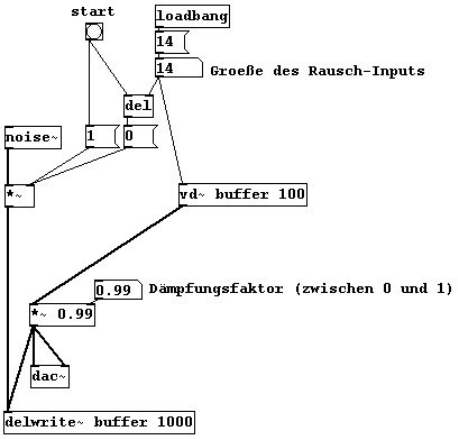


patches/3-4-2-9-oktavdoppler.pd

3.4.2.10 Karplus-Strong-Algorithmus

Eine besondere Anwendung von Rekursion ist der Karplus-Strong-Algorithmus. Er ist eines der ersten Beispiele für das Syntheseverfahren des physical modelling, ein Verfahren, das versucht, virtuell nachzubauen, was physikalisch beim Schwingen von Materie passiert. In unserem Beispiel wird dem physikalischen Modell einer gezupften Saite gefolgt: Durch den Impuls des Zupfens schwingt die Saite zunächst chaotisch und pendelt sich dann auf die Saitenlänge ein. Außerdem nimmt die Energie ab, d.h. die Saite verklingt. Dies lässt sich mathematisch nachbauen: Man nimmt einen Ausschnitt aus weißem Rauschen und spielt diesen periodisch immer wieder ab, indem es immer wieder in einen Buffer geschrieben bzw. daraus gelesen wird:

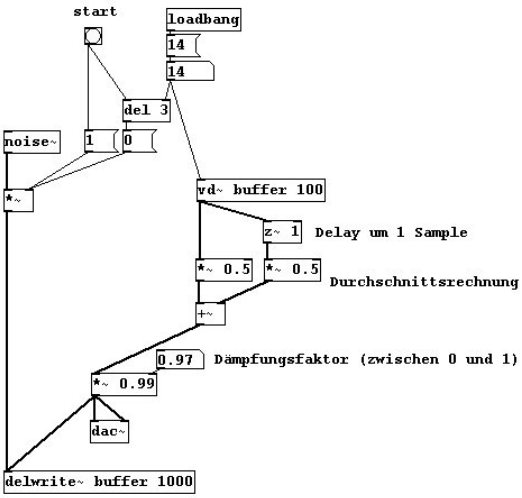
patches/3-4-2-10-karplus-strong1.pd



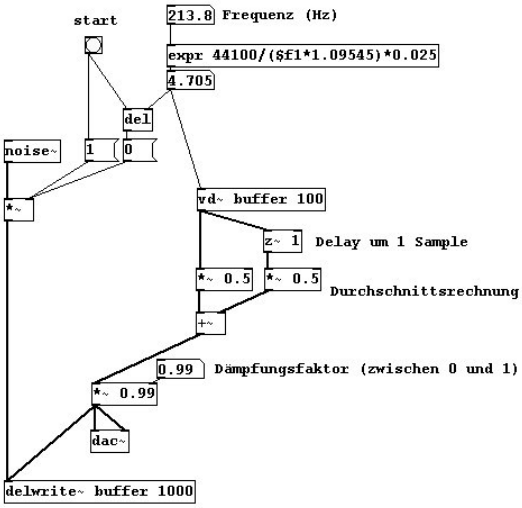
Der Saiteneffekt lässt sich aber noch vergrößern, in dem das anfänglich Material immer 'weicher' schwingt. Das funktioniert mit einer Durchschnittsrechnung: Es wird jeweils zwischen zwei

aufeinander folgenden Samples der Durchschnittswert gebildet und dieses Ergebnis dann wieder in den Buffer geschrieben. Dadurch verliert die Schwingung immer mehr an 'Ecken'. Den Delay um ein Sample erstellt man mit dem (Pd-extended-)Objekt „z~“, als Argument gilt die Anzahl der Samples:

patches/3-4-2-10-karplus-strong2.pd



Bei jedem Starten klingt der Ton etwas anders. Das liegt daran, dass „noise~“ Zufallszahlen produziert, die natürlich jedesmal verschieden sind. Wir können noch die Rechnung anfügen für die resultierenden Frequenzen:



patches/3-4-2-10-karplus-strong3.pd

3.4.2.11 Weitere Aufgabenstellungen

a) Bauen Sie in den Sample-Player eine Aufnahmefunktion ein.

b) Bilden Sie einen Hall bzw. eine Textur mit verschiedenen Delay-Zeiten des Eingangssignals, zum Beispiel mit Vielfachen der Fibonacci-Reihe (die nächste Zahl bezeichnet immer die Summe der zwei vorangegangenen, also 1 2 3 5 8 13).

c) Schaffen Sie mit verschiedenen Karplus-Strong-Klängen Texturen von variabler Dichte.

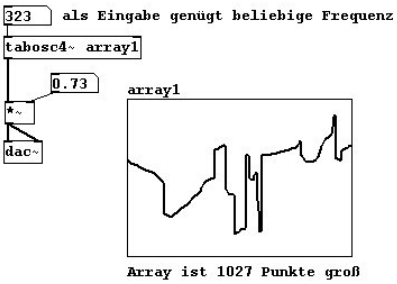
d) Wenden Sie den Kammfilter auf andere in den vorigen Kapiteln vorgestellte Klänge an.

3.4.3 Appendix

3.4.3.1 Arrayoszillator

Eine Vereinfachung der Kombination aus „tabread~“ mit einem multiplizierten „phasor~“-Signal ist „tabosc4~“. Dies liest mit gegebener Frequenz einen Array aus. Etwas umständlich ist dabei allerdings, dass der Array hierbei die Größe einer 2er-Potenz (also z. B. 128, 512, 1024) plus drei Punkte haben muss.

patches/3-4-3-1-arrayoszillator.pd

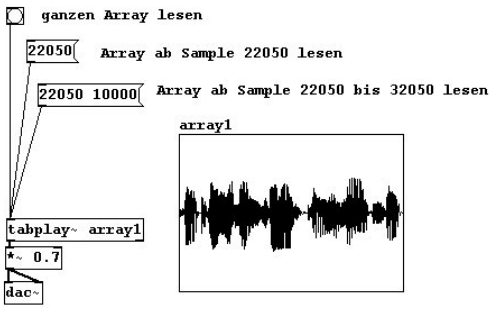


Hier berühren wir schon das in Kapitel 3.5 näher erläuterte „waveshaping“. Wir können mit der Maus eine beliebige Welle in den Array einzeichnen.

3.4.3.2 Array einfach abspielen

Noch eine Vereinfachung ist „tabplay~“. Es spielt einen Array einfach in Originalgeschwindigkeit ab (mit Bang). Komfortabel ist, dass der Start- und Endpunkt auch bestimmt werden kann (Startwert und Länge des Lesens (in Samples)):

patches/3-4-3-2-einfach-array-abspielen.pd



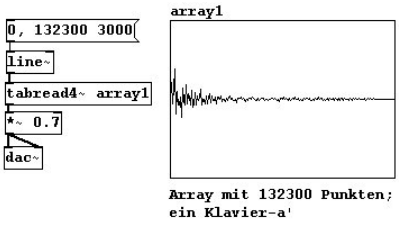
3.4.3.3 Array im Block abspielen

Eine Sonderform von „tabwrite~“ und „tabread~“ sind „tabsend~“ und „tabreceive~“. Sie schreiben / lesen einen Array synchron zu den „Blocks“ (3.1.1.3.2). „tabwrite~“ schreibt jeden Block in einen

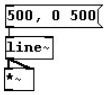
Array (der darum natürlich die Größe des Blocks haben muss, by default sind das in Pd 64 Samples). „tabreceive~“ liest in jedem Block den Array. Wir werden dies im fft-Kapitel (3.8) wieder antreffen.

3.4.3.4 Glissandi von Samples

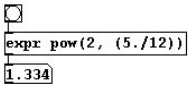
Man kann einen Array in normaler Tonhöhe abspielen oder eine Oktav höher etc. Was aber, wenn man ein Glissando von der Oktav zur Originalhöhe erstellen will? Hierzu brauchen wir eine Unterteilung in „Hauptzeiger“ und „Addition“. Der „Hauptzeiger“ verläuft in Normalgeschwindigkeit über den Array. Nehmen wir als Beispiel einen Array mit 132300 Punkten, also 3000 Millisekunden Länge:



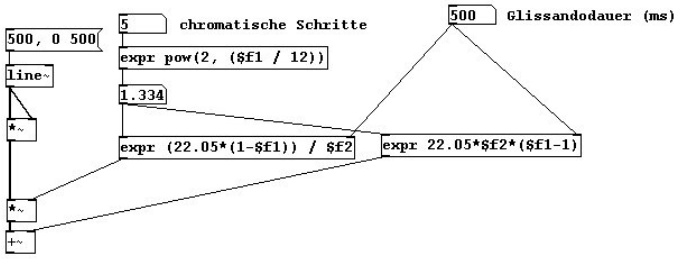
Dazu kommt die „Addition“, die das Glissando bewirkt. Nehmen wir als Beispiel, dass ein Glissando bei fünf chromatischen Schritten über der Originaltonhöhe beginnen soll und sich in 500 Millisekunden zur Originaltonhöhe hinbewegt. Wir erstellen eine umgekehrte „line~“ davon, und quadrieren die Werte:



Überdies müssen wir den Faktor der Frequenzen für die fünf chromatischen Schritte ermitteln (vgl. 3.1.1.4.3) ...

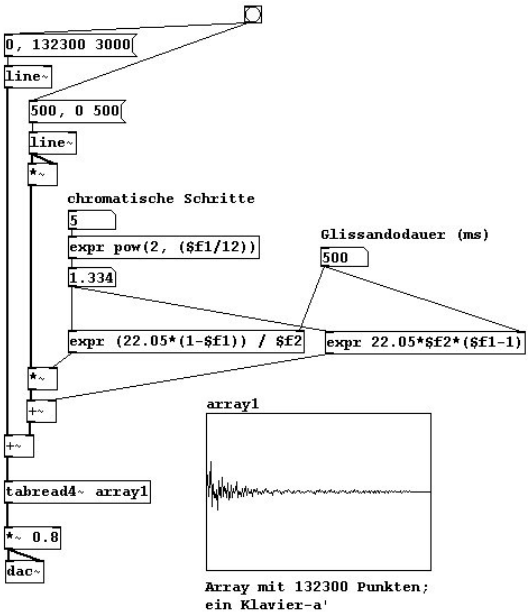


... und anschließend folgende Rechnung durchführen:



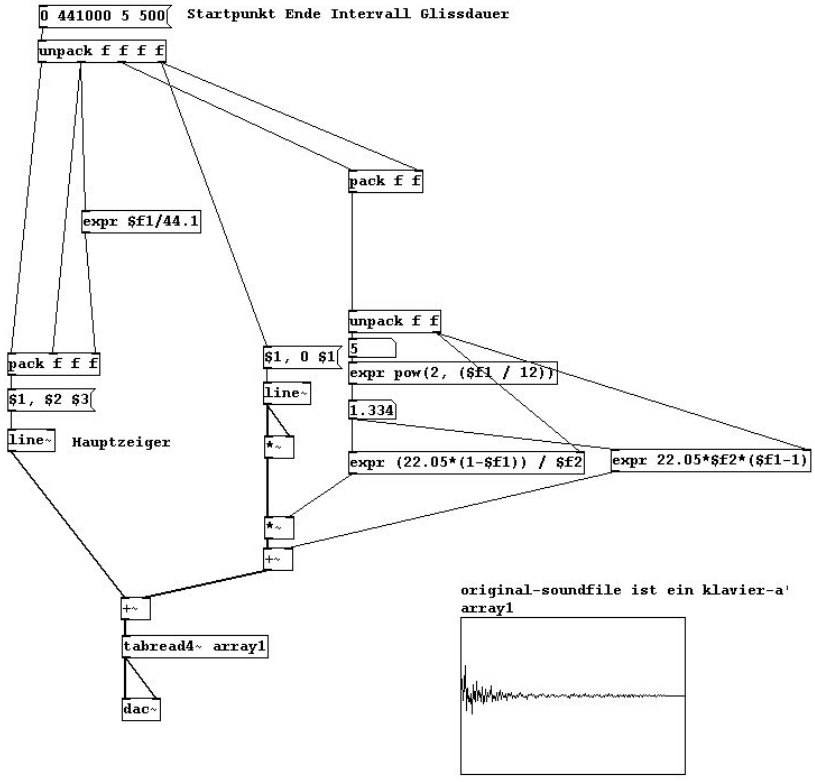
Dies ist die „Addition“, die zu dem „Hauptzeiger“ hinzukommt:

patches/3-4-3-4-sample-glissando1.pd



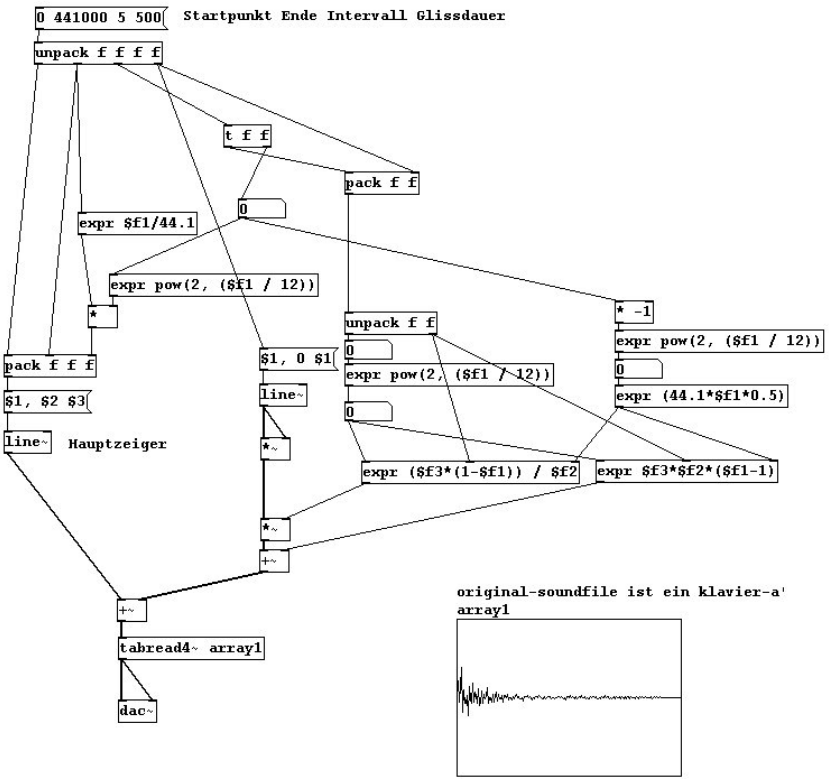
Letztlich können wir so beliebige Glissandi zum Zielton erheben; auch negative Werte sind möglich:

patches/3-4-3-4-sample-glissando2.pd



Umgekehrt, vom Originalton weg, lässt sich noch ergänzen:

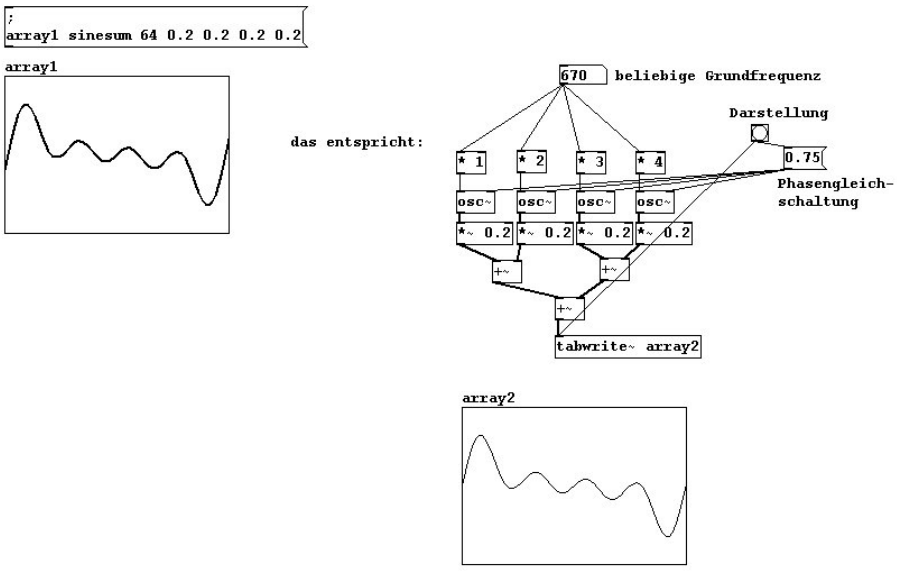
patches/3-4-3-4-sample-glissando3.pd



3.4.3.5 Additive Synthese mit Array

Als besondere Funktion kann in Pd in einem Array eine Summe von Sinuswellen erstellt werden, also eine additive Synthese wie in Kapitel 3.2 beschrieben. Dies erfolgt mit der Message „sinesum“. Als erstes Argument gibt man die (neue) Arraygröße (sollte eine 2er-Potenz sein; zu der werden dann automatisch noch drei Punkte zum idealen Phasenanschluss hinzugefügt, vom hinteren zum vorderen Ende) und zudem Lautstärkefaktoren für beliebig vieleTeiltöne:

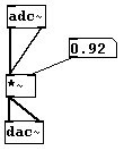
patches/3-4-3-5-sinesum.pd



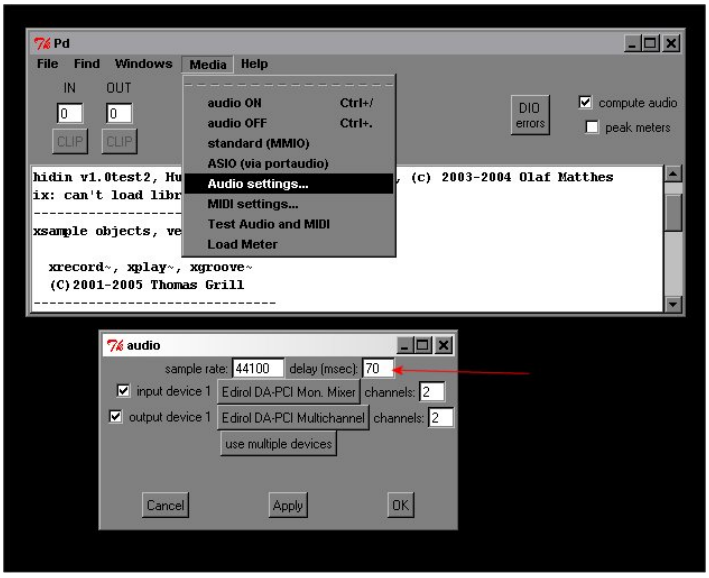
Statt Sinuswellen kann man auch Cosinuswellen einsetzen, entsprechend mit „cosinesum“.

3.4.3.6 Latenz

Audiodelay erscheint leider auch ungwollt. Bei einer einfachen Schaltung des Mikrofons an die Lautsprecher kann man das schon hören, wenn man einen ganz kurzen Klang ins Mikrofon gibt, zum Beispiel Schnalzen:



Je nach Soundkarte und vor allem Betriebssystem gibt es eine sogenannte „Latenz“. Bestenfalls ist sie so gering (unter 5 ms), dass die Verzögerung vom menschlichen Ohr nicht wahrgenommen werden kann. Dies erfordert aber einen schnellen Rechner, eine gute Soundkarte und ein dafür geeignetes Betriebssytem. Man kann in **Media** # **Audio settings** die Latenzzeit einstellen:



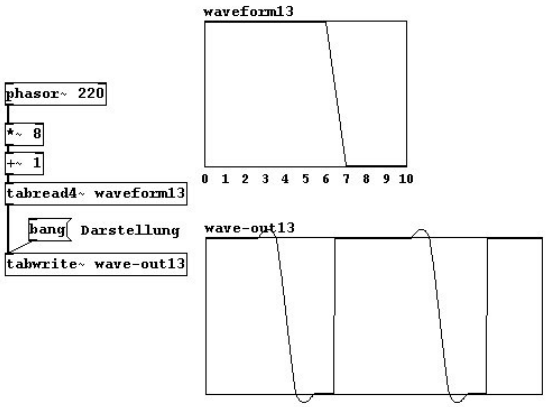
In Microsoft Windows gelangt man derzeit (Juni 2008) noch nicht unter eine Latenz von ca. 50 ms, ohne dass Störungen auftreten.

3.4.4 Für besonders Interessierte

3.4.4.1 4-Punkt-Interpolation

An diesem Beispiel kann man die Funktionsweise „tabread4~“ - Interpolation sehen:

patches/3-4-4-1-vier-punkt-interpolation.pd

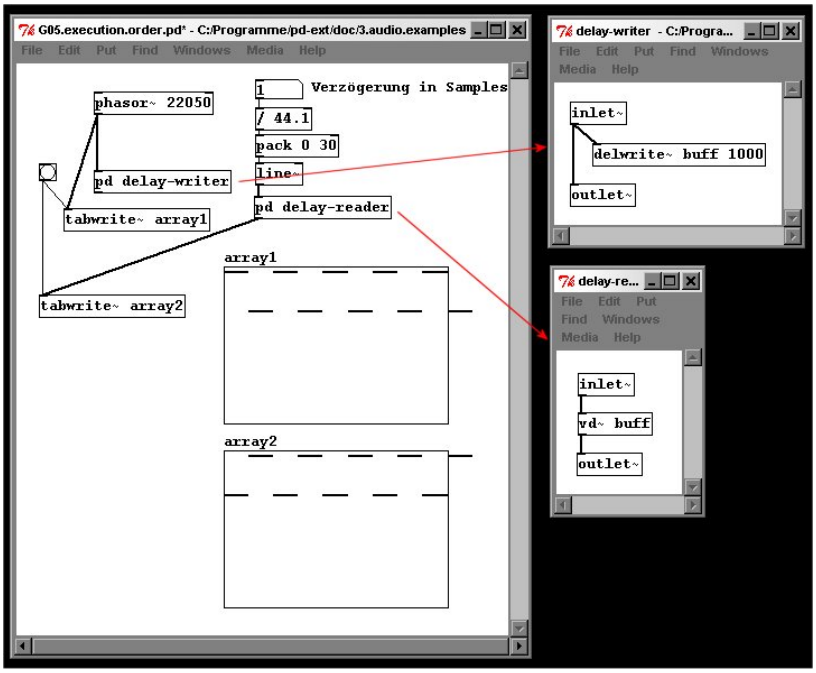


Der Sprung von 1 zu -1 wird 'weicher' gemacht durch eine Art sinusoide Interpolation. Für ein zu interpolierendes Intervall zwischen zwei Punkten werden, wie es der Name andeutet, zwei vor und zwei hinter dem Intervall verwendet und entsprechend geändert.

3.4.4.2 Sampleweise Delay

Eine Möglichkeit der sampleweisen Verschiebung mit „delread~“ und „vd~“ ist die Auslagerung in Unterpatches (sonst tritt das Problem der Blockgröße, wie vorhin bei der Oktavierung, ein):

patches/3-4-4-2-sampleweise-delay.pd



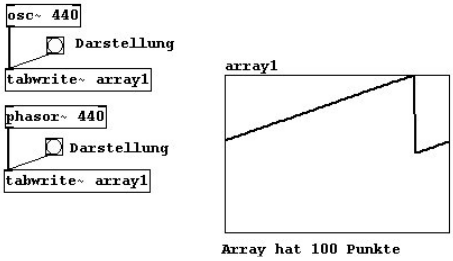
3.5 Wave-Shaping

3.5.1 Theorie

3.5.1.1 Wellenformen

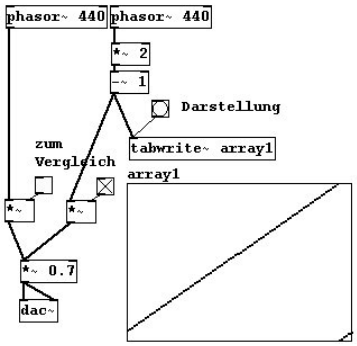
Wir haben bereits unter 3.1.1.1.2 verschiedene Wellenformen kennengelernt (Sinus, Sägezahn, Dreieck, Rechteck und Puls). Für zwei dieser Wellenformen gibt es in Pd Objekte, die sie eigens erzeugen, nämlich „osc~“ für den Sinus, „phasor~“ für den Sägezahn. Mit einem Array können wir die Wellenformen nun immer gleich darstellen:

patches/3-5-1-1-wellenformen-darstellen.pd



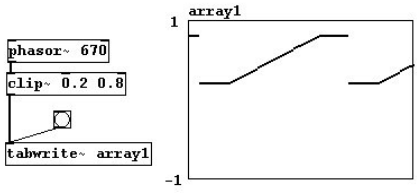
Man beachte: Der Sägezahn des „phasor~“ geht immer von 0 bis 1, kommt also nie in den negativen Bereich. Man könnte ihn aber dahingehend noch etwas kräftigen, mit einer kleinen Rechnung:

patches/3-5-1-1-starker-phasor.pd



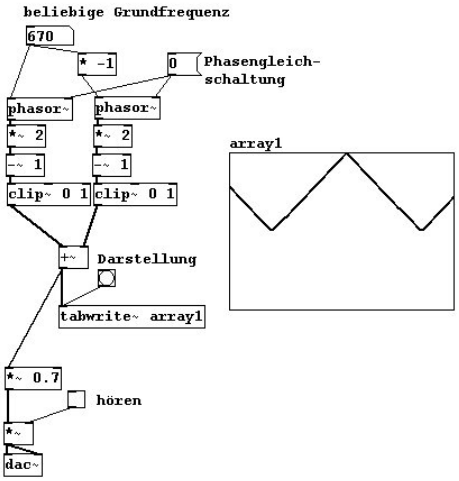
Wir können nun aber auch anderen Wellenformen erzeugen, wenn wir ein paar Rechenoperationen an den „phasor~“ anbringen. Neu ist dafür das Objekt „clip~“, das 'überstehende' Bereiche wegschneidet. Man gibt als Argumente zwei Zahlen für die untere und obere Grenze, an der jeweils 'abgeschnitten' werden soll:

patches/3-5-1-1-andere-wellenformen.pd



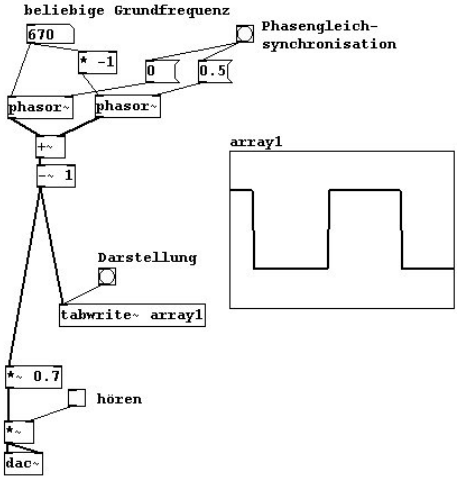
Nun zu den Wellenformen Dreieck:

patches/3-5-1-1-dreieck.pd



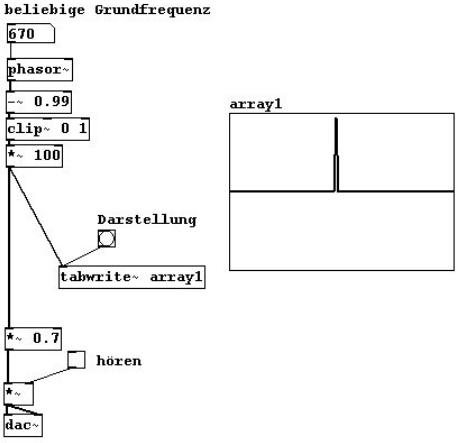
Rechteck:

patches/3-5-1-1-rechteck.pd



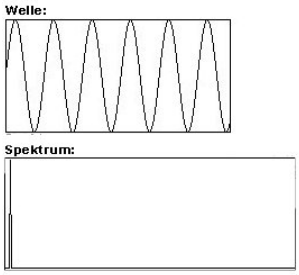
Puls:

patches/3-5-1-1-puls.pd

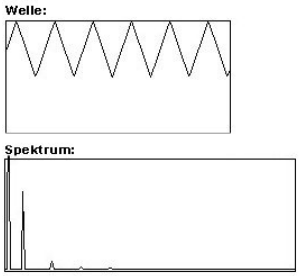


Dies sind alles noch standardisierte Wellenformen, die alle bestimmte Charakteristika haben:

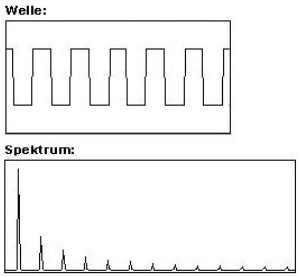
Sinus: nur Grundton ohne Obertöne



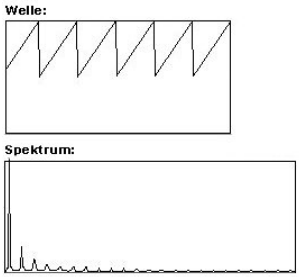
Dreieck: so ähnlich wie Sinus, plus ungerade Teiltöne



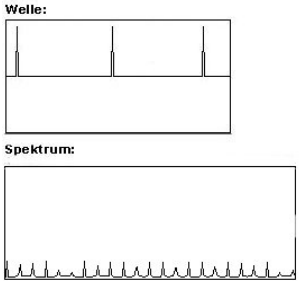
Rechteck: nur ungerade Teiltöne



Sägezahn: alle Teiltöne



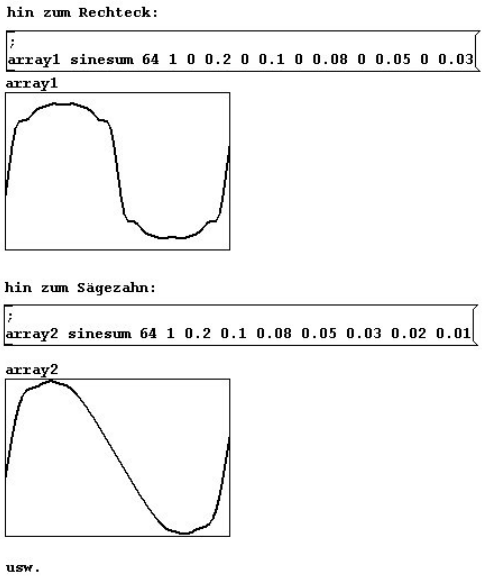
Puls: alle Teiltöne, fast gleich laut



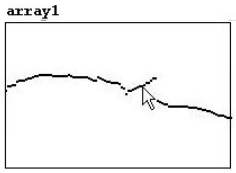
Wir sehen: Symmetrische Wellenformen haben nur ungerade Teiltöne (in jede Periode passen immer genau zwei Perioden des nächsten ungeraden Teiltons, sie sind darin also symmetrisch), während unsymmetrische immer auch gerade Teiltöne haben.

Diese Wellenformen lassen sich auch mit additiver Synthese (annäherungsweise) erstellen:

patches/3-5-1-1-wellenformen-fourier.pd



Der Experimentierfreude sind hier keine Grenzen gesetzt, und gerade dadurch gelangt man zu neuen Klängen. So kann man einfach eigene Wellenformen in einen Array einzeichnen. Man muss nur im Execute Mode auf die Linie im Array gehen, woraufhin der Cursor ihre Richtung ändert ...

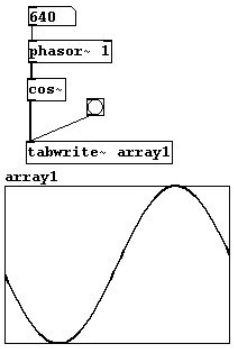


... und dann bei gedrückter Maustaste eine individuelle Welle zeichnen.

Dies ist aber natürlich etwas umständlich und 'unelegant'. Befassen wir uns also mit der Theorie des 'Waveshaping':

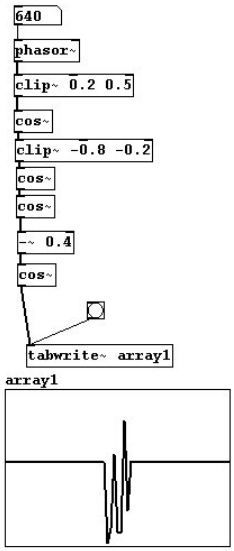
3.5.1.2 Transferfunktionen

Eine lineare Funktion geht durch eine sogenannte „Transferfunktion“. Für die lineare Funktion bietet sich freilich der Phasor~ an, der immer von 0 bis 1 geht. Aus ihm können wir zum Beispiel eine Cosinuswelle machen, mit dem „cos~“-Objekt, das eben eine Cosinusfunktion berechnet:



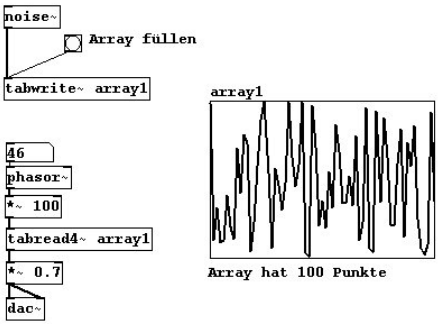
Und so können alle möglichen Transfers gemacht werden, nur als ein Beispiel:

patches/3-5-1-2-transferfunktion.pd



3.5.1.3 (Gelenkte) Zufallswellenformen

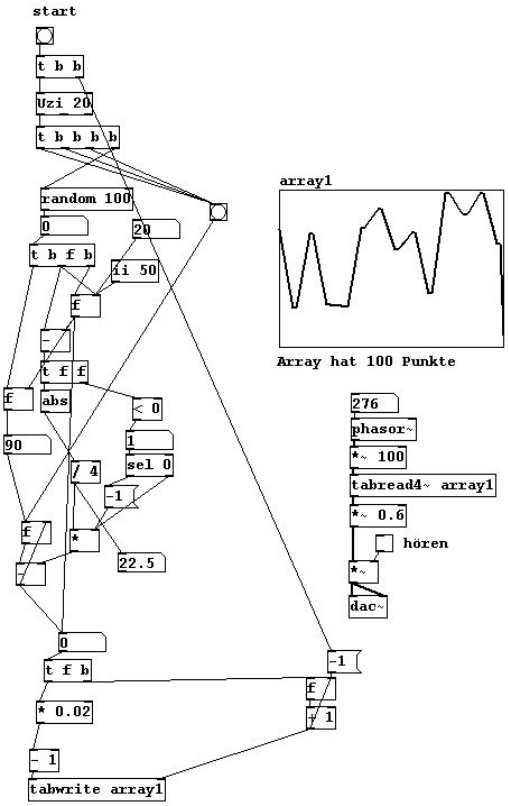
Wir können aber auch Rauschen in einen Array aufnehmen und nachher periodisch auslesen; d. h. dasselbe immer wieder hintereinander (vgl. Karplus Strong). Wenn es häufiger als 20 Mal pro Sekunde auftritt, wird daraus eine Tonhöhe:



Das Spektrum dieses Klanges ist natürlich nicht mehr genau absehbar. Jedes Mal, wenn wir den Array mit Zufallszahlen aus dem „noise~“ füllen, erhalten wir eine neue Welle mit neuen Charakteristika.

Aber eine gewisse Systematik lässt sich doch erkennen. Wir können beispielsweise Klicks, also große Sprünge innerhalb der Welle interpolieren, um zu einem weicheren Resultat zu kommen. Generieren wir also Zufallspunkte mit linearer Interpolation dazwischen („Uzi“ aus Pd-extended erzeugt so schnell wie möglich so viele Bangs hintereinander, wie als Argument gegeben sind):

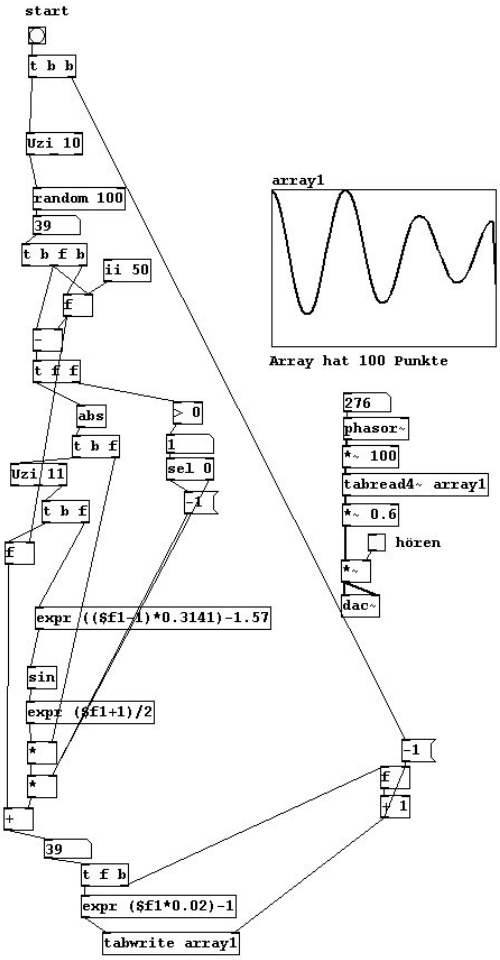
patches/3-5-1-3-wavgorithm.pd



In diesem Beispiel wird immer mit vier Punkten interpoliert.

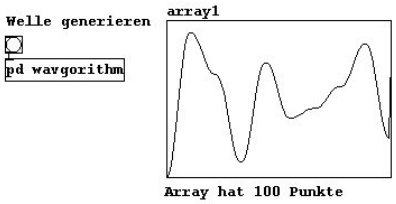
Noch weicher wird das Ergebnis, wenn wir statt einer linearen eine sinusoide Interpolation erzeugen. Wir verwenden nun zehn Interpolationspunkte:

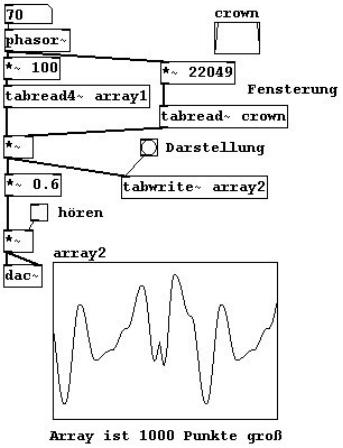
patches/3-5-1-3-wavgorithm+sin.pd



Nun sollte auch noch der Anschluss von hinten nach vorne interpoliert werden. Dazu empfiehlt sich eine Fensterung; die Berechnung bringen wir in einem Subpatch unter.

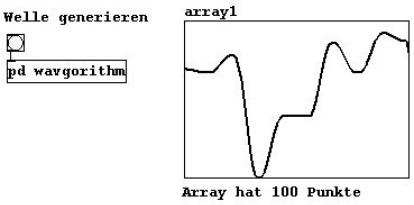
patches/3-5-1-3-wavgorithm+sin+fenster.pd

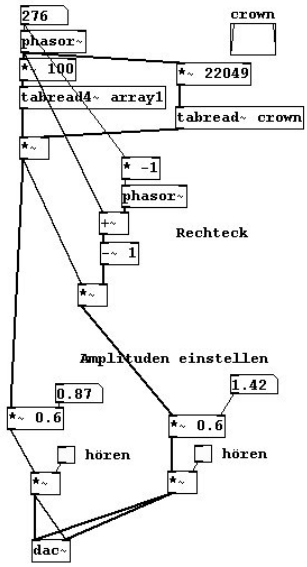




So haben wir am Anfang wie am Schluss immer die Membran auf 0. Noch weicher wäre das Ergebnis, wenn wir es mit dem Hanning-Window „fensterten“ (siehe 3.9.4.1).

Außerdem sind wiederum Transferfunktionen mit den bekannten Wellenformen möglich, etwa mit einem Rechteck, so dass wir wieder weit in den Bereich der ungeraden Teiltöne kommen.



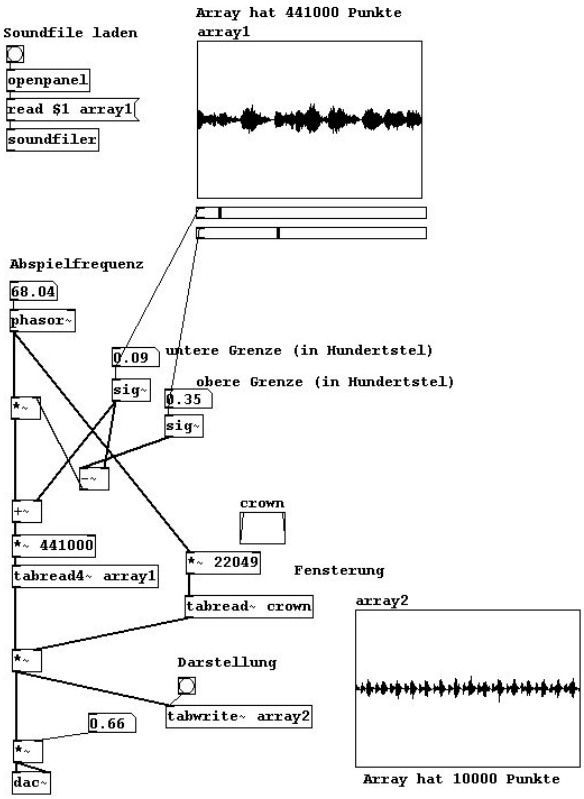


Etc. In den letzten Beispielen mussten wir allerdings auf Kontroll-Ebene die Welle erstellen; im Gegensatz zu den ersten Beispielen nur mit Transferfunktionen können wir diese nicht „live“ verändern.

3.5.1.4 Wave-Stealing

Eine letzte, quasi-Waveshaping-Synthese-Technik ist „wave-stealing“: Dafür können wir einfach aus bekannten Musikstücken einen kleinen Teil herausnehmen...

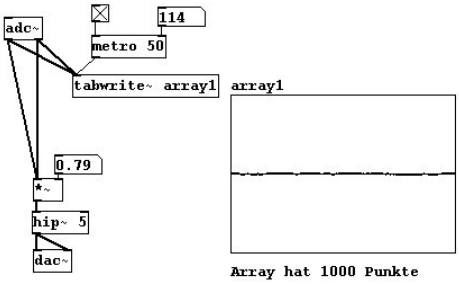
patches/3-5-1-4-wavestealing.pd



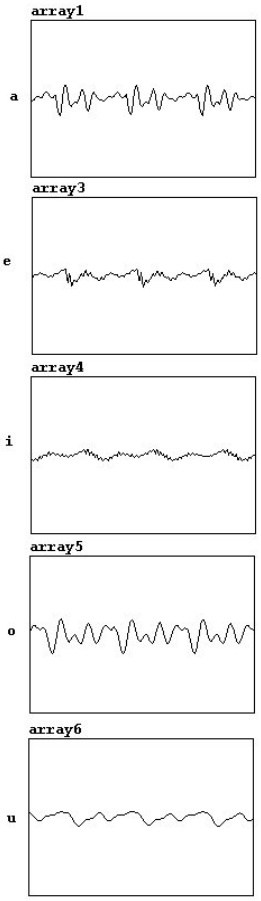
3.5.2 Anwendungen

3.5.2.1 Wellenformen singen

Mit dem folgenden Patch ist es möglich, Wellenformen mit dem Mikrofon aufzunehmen und so beispielsweise Wellenformen zu singen.

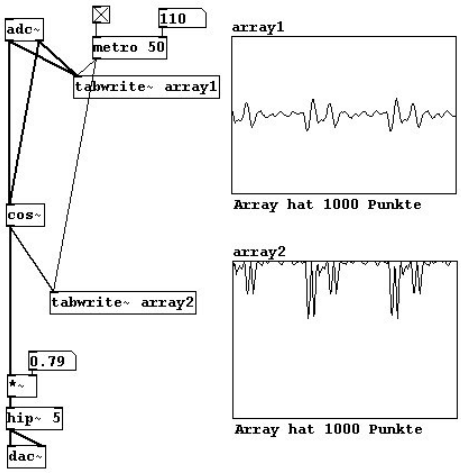


Verschiedene Vokale sehen ungefähr so aus:



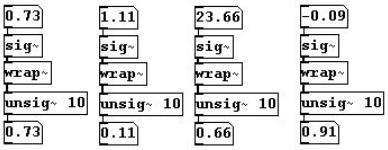
3.5.2.2 Transfers

Und dieses Eingangssignal können wir natürlich auch durch eine Transferfunktion schicken:



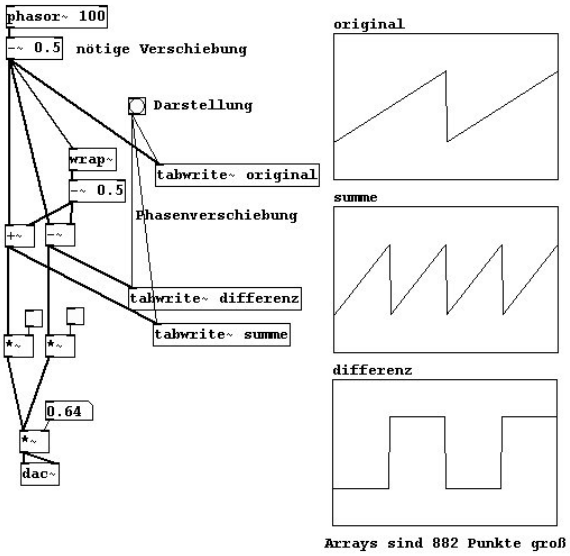
3.5.2.3 Gerade / ungerade Teiltöne

Man kann auch einen Sägezahn in gerade und ungerade Teiltöne zerlegen. Wir bedienen uns hierbei des „wrap~“-Objekts. Es berechnet die Differenz von der Eingangszahl zur nächst darunter liegenden Integerzahl (Ausgabe wird immer in den positiven Bereich gebracht). Kurz ein paar Beispiele hierfür:



Und nun die Sägezahn-Zerteilung. Durch das „wrap~“-Objekt erzielen wir eine Phasenverschiebung des Sägezahns – diese addiert bzw. subtrahiert mit dem Originalsignal ergibt einen doppelt so schnellen Sägezahn und ein Rechteck.

patches/3-5-2-3-gerade-ungerade-teiltoene.pd



3.5.2.4 Weitere Aufgabenstellungen

a) Erzeugen Sie eine sich ständig ändernde Welle.

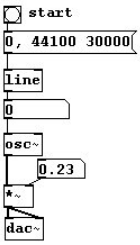
b) Machen Sie die Anzahl der Interpolationspunkte und der Arraypunkte bei den (gelenkten) Zufallswellenformen variabel.

3.5.3 Appendix

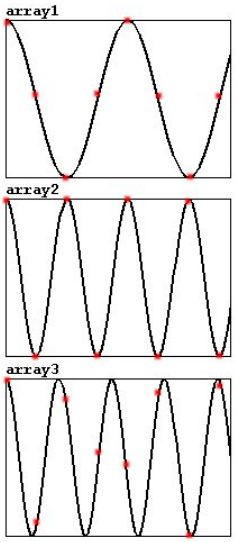
3.5.3.1 Foldover

An dieser Stelle bekommen wir es mit einem sehr heiklen Problem in der digitalen Klangverarbeitung zu tun: dem Foldover. Betrachten wir zunächst diesen Fall:

patches/3-5-3-1-foldover1.pd



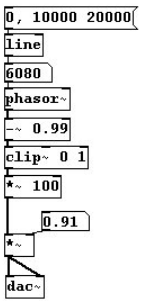
Was passiert? Ab 22050 Hz kehrt sich die Richtung um, bis wir bei 44100 Hz wieder bei einer Tonhöhe von 0 Hz angekommen sind (danach würde es wieder aufwärts gehen). Der Grund dafür ist, dass wir bei einer Samplerate von 44100 Hz maximal eine Welle von 22050 erstellen können (siehe 3.1.1.3.1). Darüber hinaus kommt es zu charakteristischen Lesefehlern. Betrachten wir Wellen mit verschiedenen Frequenzen: die obere hat 11025 Hz, die mittlere 22050 und die untere etwas mehr als 22050. Die roten Markierungen stehen für die Abtastpunkte, die immer dieselbe Geschwindigkeit von 44100 pro Sekunde haben.



Jede Periode einer Welle von 11025 Hz kann mit immerhin vier Punkten erfasst werden (die spezielle Sinuscharakteristik geht dabei natürlich verloren). 22050 ist die höchste Frequenz, die überhaupt noch korrekt erfasst werden kann, da wir ja, laut Nyquist-Theorem, mindestens zwei Punkte pro Periode brauchen. Bei einer Frequenz, die noch höher ist, kommt es folglich zu Fehlern; es wird nicht mehr jede Periode erfasst und statt einer höheren Frequenz ergeben die Ablesepunkte wieder eine tiefere.

Noch schärfer tritt das Problem aber bei solchen Wellen auf, die Obertöne haben, zum Beispiel beim Puls:

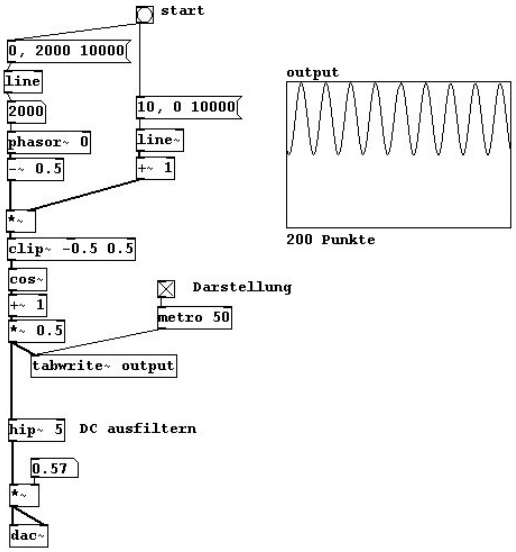
patches/3-5-3-1-foldover2.pd



Hier tritt der Effekt schon viel früher ein: Ab ca. 700 Hz kehren sich bereits Obertöne wieder um. Das liegt daran, dass die Puls-Welle praktisch nur aus einem einzigen Strich besteht; und dieser wird

schon bald 'verfehlt'. Eine Lösung für das Problem ist, bei einer Puls-Welle zu beginnen, sie aber, je höher die Frequenz ist, umso breiter zu machen, bis sie am Ende ein Sinus ist:

patches/3-5-3-1-foldover3.pd



3.5.4 Für besonders Interessierte

3.5.4.1 GENDY

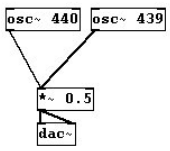
Iannis Xenakis hat ebenfalls in seinen späten Jahren ein Verfahren zur Wellengenerierung entwickelt mit dem Namen GENDY. Er 'komponierte' nichts als Wellenformen und deren Entiwcklungen, zum Beispiel im Tonbandstück „Gendy 3“.

3.6 Modulationssynthesen

3.6.1 Theorie

3.6.1.1 Ringmodulation

Beobachten wir zunächst dieses Phänomen:

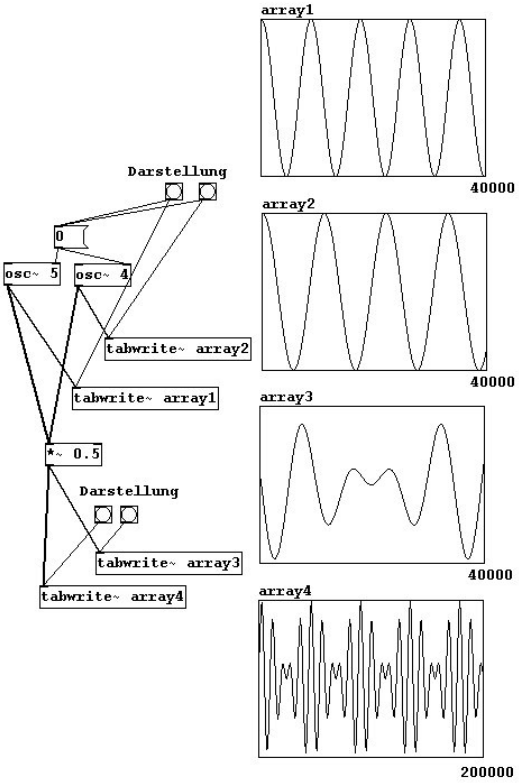


Wenn wir zwei Sinustöne sehr nah beieinander haben, ergeben sich schwankende Auslöschungen. Dies liegt an der Überlagerung der fast (aber eben nur fast!) gleichen Wellen. Man nennt dieses

Phänomen *Schwebungen*. Die Geschwindigkeit des Rhythmus' ist die genau die der Differenz zwischen den beiden Frequenzen – in diesem Beispiel also 440 - 439 = 1 Hz.

Nehmen wir als anschauliches Beispiel Oszillatoren mit 4 und 5 Hz:

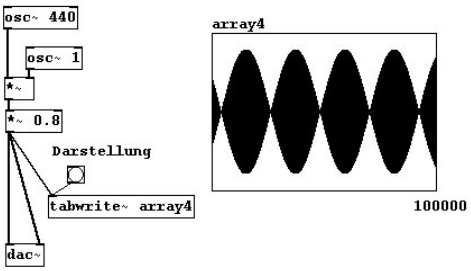
patches/3-6-1-1-ringmodulation1.pd



Abwechselnd kommt es zu gegenseitiger Verstärkung (vergessen wir nie: Wellen addieren sich!) und Auslöschung. In array1 und array2 sehen wir einen kleinen Teil der Originalwellen, in array3 deren Addition, selbige in array4, nur auf einen etwas längeren Zeitraum hin betrachtet. Es ergibt also ein pulsierendes An- und Abschwellen der Lautstärke. (Zwei verschiedene Töne hören wir erst ab einer Differenz von ca. fünf Cent.)

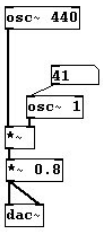
Diesen Wellenverlauf können wir aber umgekehrt erstellen, indem wir tatsächlich genau diesen Rhythmus der Amplitude bestimmen. Wie wir in der vorigen Grafik in array4 sehen konnten, hat der Verlauf der Amplitude wiederum genau die Form eines Sinuses. Daher verwenden wir nun auch zur Bestimmung der Amplitude einfach einen Oszillator:

patches/3-6-1-1-ringmodulation2.pd



(Dass der Array jetzt im Gegensatz zur Grafik davor so schwarz ist, liegt daran, dass wir es hier mit viel höheren Frequenzen zu tun haben.) Der Clou an der Sache ist nun: Die resultierende Welle entspricht

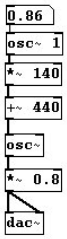
der Addition zweier Wellen. Wenn wir nun die Frequenz der modulierenden Amplitude immer weiter erhöhen ...



... hören wir zwei Frequenzen, die symmetrisch auseinandergehen, eine nach oben, eine nach unten, und zwar um den Betrag der Frequenz, den die Amplitude von der Mittelachse weggeht, also der Ausgangsfrequenz. Wie sprechen davon, dass die Amplitude moduliert wird; wir führen also eine Amplitudenmodulation durch, wegen der Symmetrie auch Ringmodulation genannt. Ist die Ausgangsfrequenz 440 Hz und die Amplitudenfrequenz 100 Hz, erhalten wir zwei Töne, einen mit 340 Hz und einen mit 540 Hz.

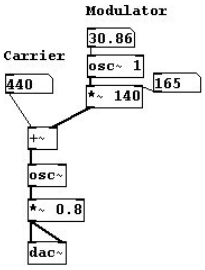
3.6.1.2 Frequenzmodulation

Nicht nur die Amplitude eines Sinussignals, sondern auch dessen Frequenz kann man mit einem Oszillator modulieren. In diesem Fall spricht man von Frequenzmodulation:

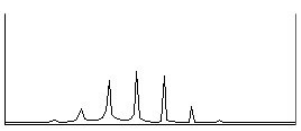


Der eine Oszillator ist der „carrier“ (Träger), der andere der „modulator“ (Modulator). Bei einer geringen Frequenz des Modulators erhalten wir ein Vibrato. Erhöhen wir die Frequenz aber über 20 Hz, kommt es zu einem immer vielstimmigeren Akkord:

patches/3-6-1-2-frequenzmodulation.pd



Denn die resultierende Welle entspricht einer Überlagerung mehrerer Sinustöne, wobei die Trägerfrequenz (engl. "carrier frequency") in der Mitte liegt und darunter und darüber jeweils im Abstand der Modulationsfrequenz (engl. "modulator frequency") die anderen Töne.



Mit dem Ansteigen der Amplitude des Modulators steigen auch die Amplituden der zusätzlichen Frequenzen an. Dieser Anstieg ist allerdings mathematisch relativ schwer nur zu beschreiben.

Ein Sonderfall ergibt sich, wenn die Modulationsfrequenz ein ganzes Vielfaches der Trägerfrequenz (also das 1, 2, 3, 4, 5, 6-fache etc.), beträgt – dann sind die Töne über dem Carrier ebenfall ganzzahlige Vielfache des Carriers, das heißt, es sind seine Obertöne.

Zu beachten ist ferner, dass negative Frequenzen wieder nach oben gespiegelt werden. Im eben genannten Sonderfall decken sich diese mit den „normalen“ Frequenzen. Hat man zum Beispiel einen Carrier von 200 Hz. und einen Modulator von 100 Hz, kommt es ab dem dritten Unterton (der ja wieder 100 Hz hat und die folgenden 200, 300 etc.) zu Überdeckungen, die je nach Phasenlage zu Verstärkungen oder Abschwächungen führen.

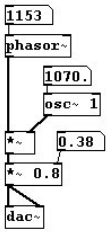
Der Nutzen der FM-Synthese gegenüber der einfachen Addition von Sinustönen ist, dass wir nur zwei Oszillatoren benötigen, um einen reichhaltigen Klang zu erzeugen (man verändere dabei die Frequenz und vor allem die Amplitude des Modulators!). Typisch für die FM-Synthese sind die disharmonischen Spektren, also ein Quasi-Spektrum über einem Grundton, dessen Obertöne aber nicht ganzzahlige Vielfache sind. Solche Spektren weisen einige Metallinstrumente auf wie z. B. Glocken und Gongs. Daher haben FM-Klänge häufig auch einen 'metallischen' Charakter.

3.6.2 Anwendungen

3.6.2.1 Ergiebige Ringmodulation

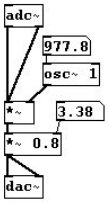
Die Ringmodulation ist bei obertonreicheren Klängen natürlich ergiebiger:

patches/3-6-2-1-ringmodulation3.pd



3.6.2.2 Ringmodulation live

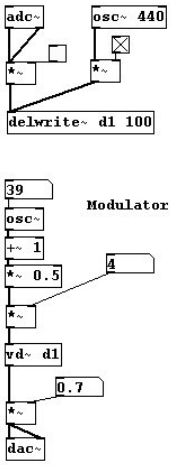
patches/3-6-2-2-ringmodulation-live.pd



3.6.2.3 Frequenzmodulation live

Für eine Live-Anwendung der Frequenzmodulation müssen wir mit einem variablen Delay arbeiten, um die Frequenz ändern zu können:

patches/3-6-2-3-frequenzmodulation-live.pd



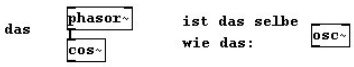
3.6.2.4 Weitere Aufgabenstellungen

Kombinieren Sie alles mögliche bislang Gelernte.

3.6.3 Appendix

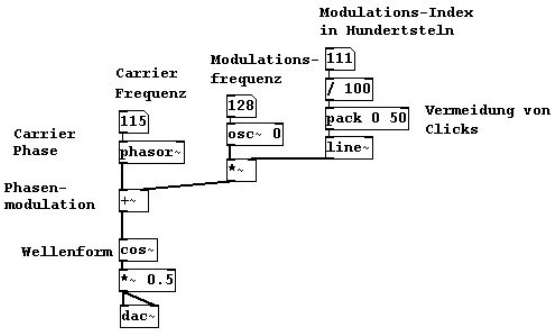
3.6.3.1 Phasenmodulation

Die Freqenzmodulation wird auch Phasenmodulation genannt und kann in dieser Form erstellt werden. Dazu wird der Carrier-Oszillator aufgeteilt in Phasenberechnung und Wellenformberechnung. Das funktioniert so:



Und die Phasenmodulation sieht so aus:

patches/3-6-3-1-phasenmodulation.pd



3.7 Granularsynthese

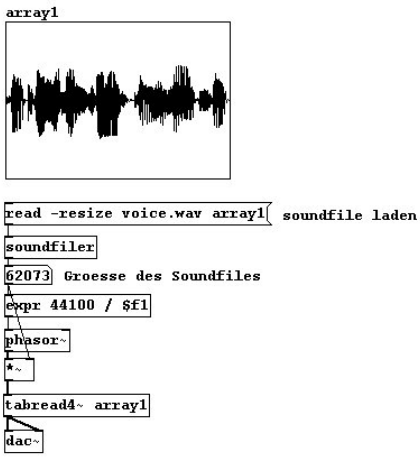
3.7.1 Theorie

3.7.1.1 Theorie der Granularsynthese

Bei der Anwendung des Samplings (3.3) konnten wir die Geschwindigkeit eines bestehenden Klanges in einem Array ändern, gleichzeitig änderte sich damit aber auch die Tonhöhe. Eine Möglichkeit, diese beiden Parameter zu entkoppeln, ist mit der Granularsynthese gegeben. Ihre Idee ist, dass ein Klang original abgetastet, aber von jedem Lesepunkt aus in anderer Geschwindigkeit abgespielt wird. Es findet also eine Trennung statt zwischen Lesepunkt und Abspielung.

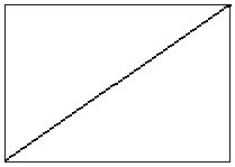
Wir haben einen „Zeiger“, der über den Array in Normalgeschwindigkeit geht:

patches/3-7-1-1-granular-theorie1.pd

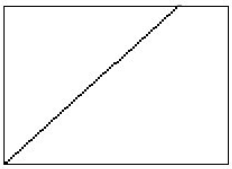


Von diesem Zeiger holen wir uns aber nur in gewissen Abständen die Information darüber, wo er gerade steht und spielen von da aus jeweils den Array ab – dann allerdings in einer anderen Geschwindigkeit.

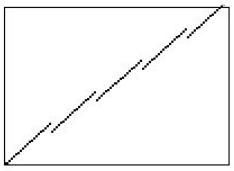
Stellen wir uns zum bessern Verständnis vor, dies wäre die normale Abspielgeschwindigkeit:



... und dies eine zu schnelle Geschwindigkeit:

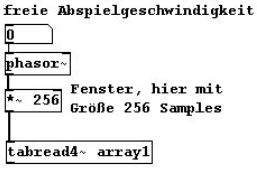


Dann macht die Granularsynthese Folgendes:



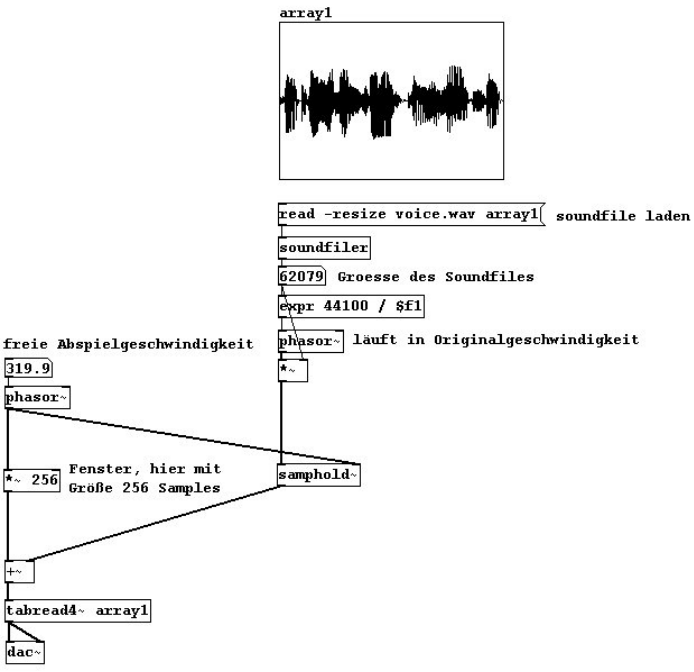
Sie spielt zu schnell (oder zu langsam) ab, aber beginnt immer wieder an einem Punkt, der der normalen Geschwindikeit entspricht. Diese einzelnen Teile nennt man „Grains“, ihre Größe bezeichnet man als „Grain Size“ oder „Fenstergröße“. „Grain“ ist englisch für „Korn“; tatsächlich handelt es sich um sehr viele kleine „Körner“, so klein, dass sie nicht einzeln wahrgenommen werden können. Das ist der Trick der Granularsynthese.

Jedes einzelne „Grain“ spielen wir so ab:



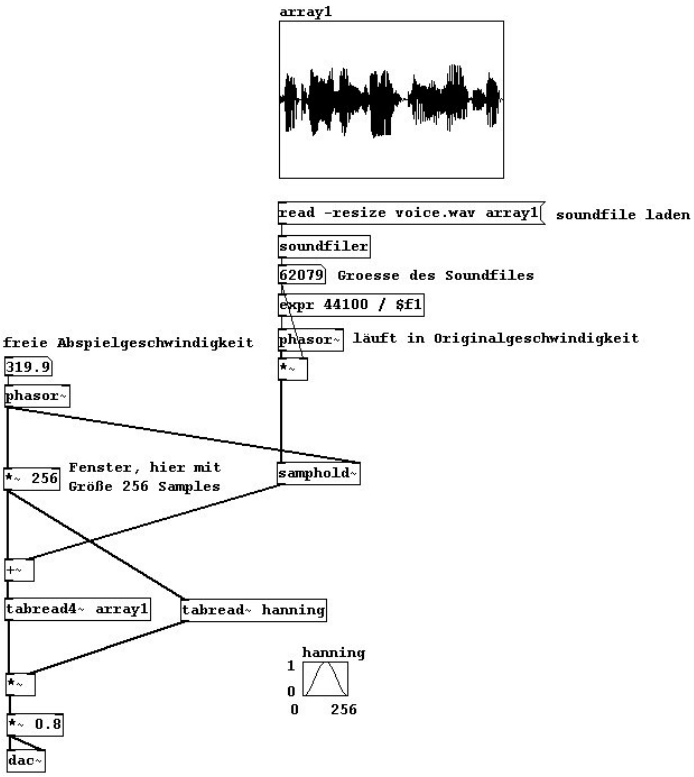
Nun folgt nach jedem Abspielen die Versetzung zur nächsten Position, die wir dem „Hauptzeiger“ entnehmen. Hierfür gibt es eigens ein Objekt, das dies reibungslos erledigt, das „samphold~“. Es funktioniert ähnlich wie „spigot“, nur eben auf der Signal-Ebene. In den linken Inlet und in den rechten Inlet kommt ein Signal. Wenn sich nun im Signal des rechten Inlets ein fallender Schritt ereignet, wird in dem Moment das aktuelle Sample des linken Inputs ausgegeben und so lange wiederholt, bis sich erneut der Wert im rechten Eingang niedriger als der vorangegangene ist. Diese etwas eigenartige Einstellung hat ihren Sinn, wenn der rechte Input ein „phasor~“ ist. Der enthält nämlich nur einmal, ganz am Ende seiner Periode, einen fallenden Schritt. So können wir also ein Grain auslesen, und an dessen Ende den nächsten Offset addieren:

patches/3-7-1-1-granular-theorie2.pd



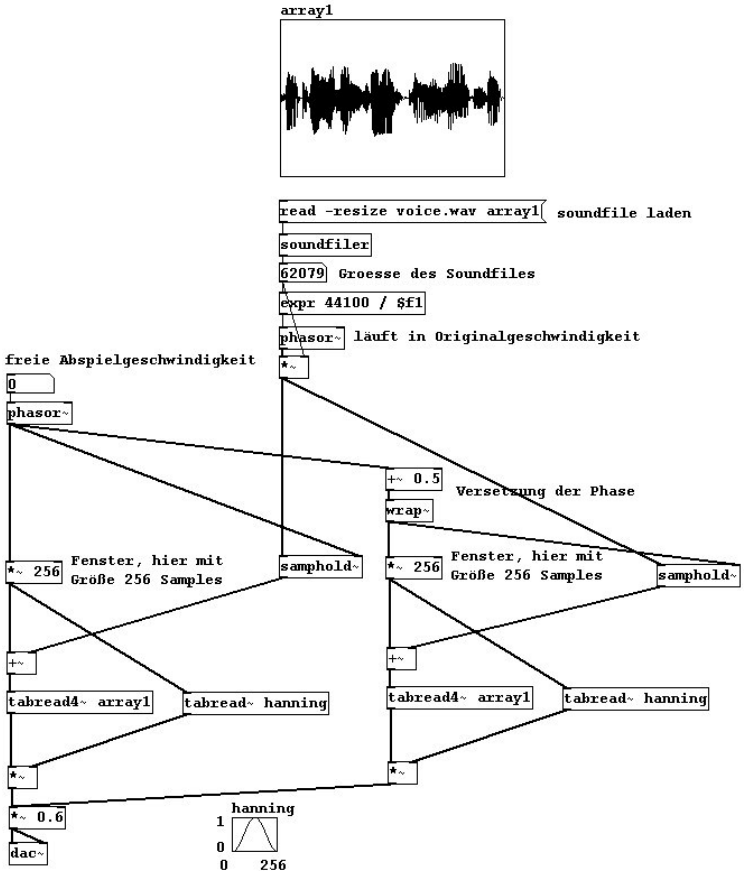
So erklingt das Ganze höher, aber dennoch insgesamt in der Originaldauer. Wenn man genau 'hinsieht', offenbart sich einem natürlich, dass es hier zu Komplikationen kommt. Wird von einem Lesepunkt aus schneller abgespielt als der Lesezeiger (der in Originalgeschwindigkeit läuft), überschreiten wir den Lesepunkt, ehe wir beim nächsten „samphold~“ wieder zu ihm zurückkehren. Umgekehrt, wenn wir die Fragmente („Grains“, dt. „Körner“) langsamer als das Original abspielen, gehen einige Teile verloren. So lange aber Originalgeschwindigkeit und Abspielgeschwindigkeit nicht allzu weit auseinanderklaffen, fällt dies nicht (sehr) auf. Um dies zu erreichen, bringen wir noch einige Verbesserungen an. Zunächst müssen wir mit einem Hanning-Window die Klicks unterbinden, die beim Sprung zu jedem neuen Wert entstehen:

patches/3-7-1-1-granular-theorie3.pd



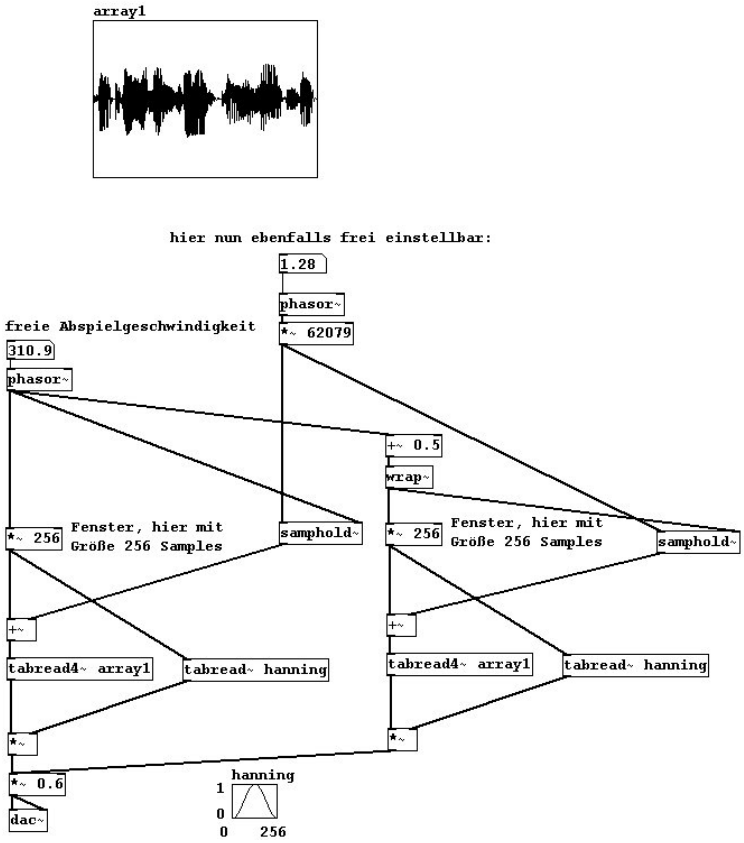
Die entstehenden Lücken schließen wir, indem wir einfach, um die halbe Periode versetzt, einen zweiten Grain-Leser anbringen:

patches/3-7-1-1-granular-theorie4.pd



Das Schöne ist nun, dass wir nicht nur die Tonhöhe unabhängig von der Geschwindigkeit ändern können, sondern auch umgekehrt:

patches/3-7-1-1-granular-theorie5.pd

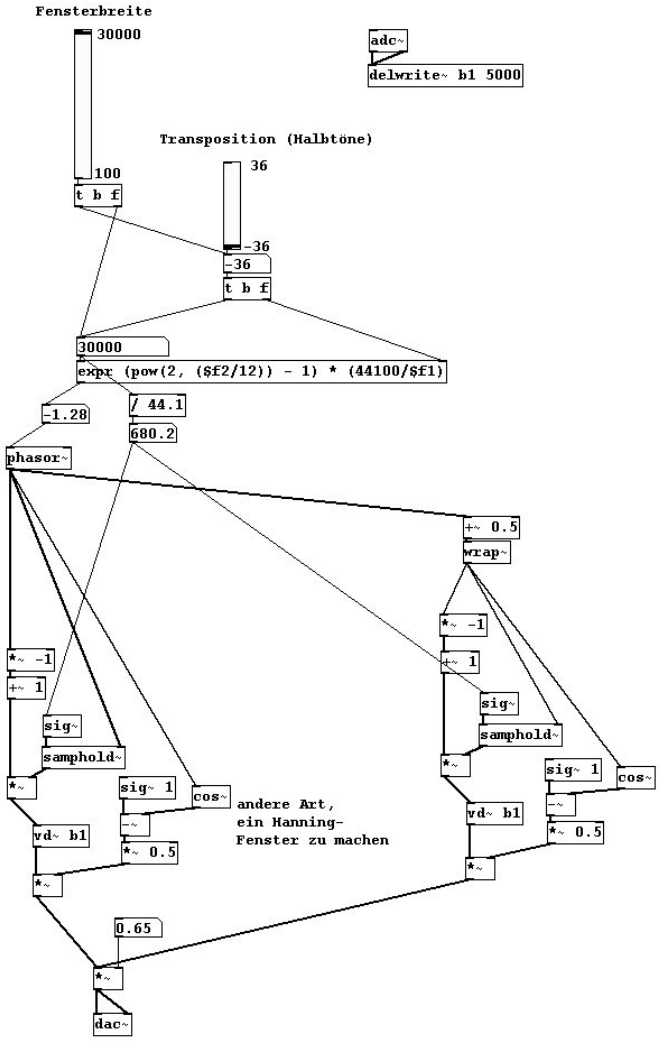


3.7.2 Anwendungen

3.7.2.1 Granularsynthese live

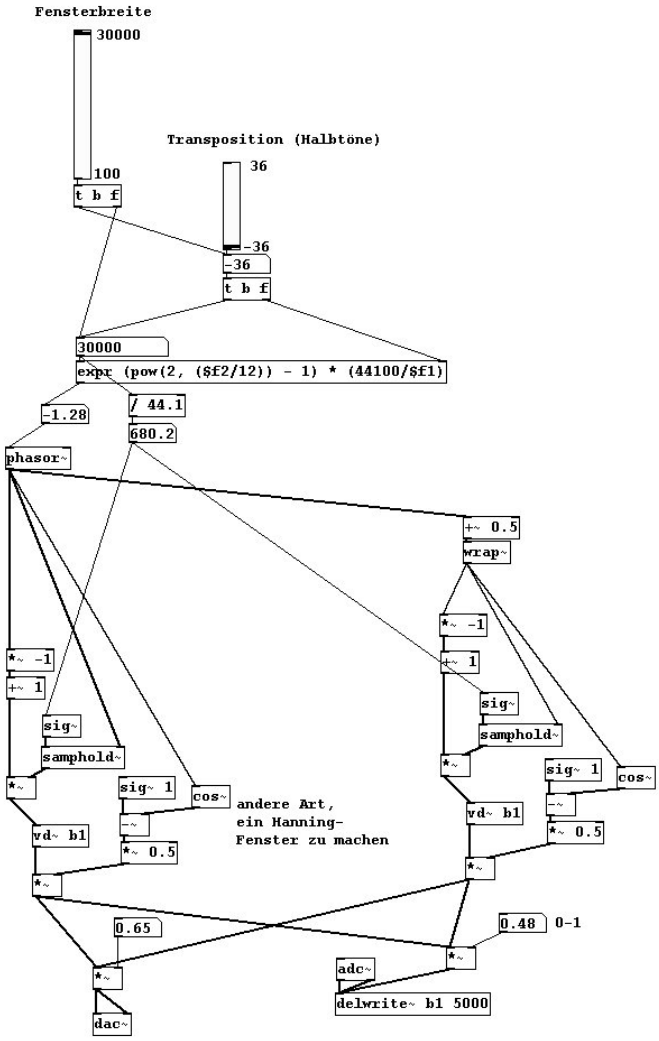
Zur Live-Anwendung bedienen wir uns wiederum variablen Delays:

patches/3-7-2-1-granular-live.pd



3.7.2.2 Live mit Feedback

patches/3-7-2-2-granular-live-feedback.pd



3.7.2.3 Weitere Aufgabenstellungen

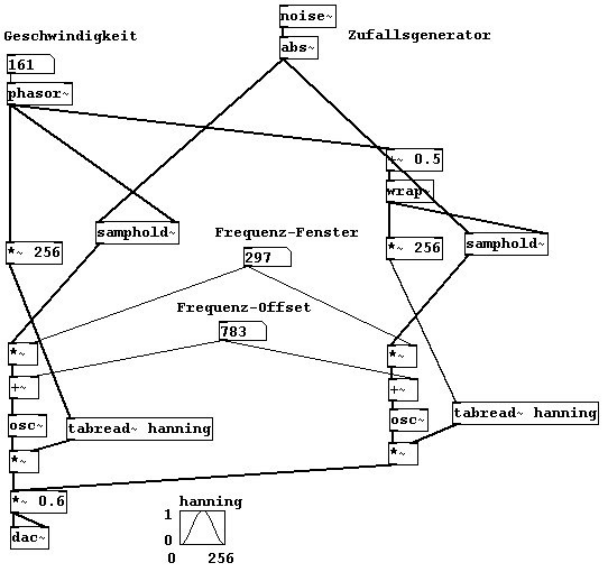
Erstellen Sie vier Leser, wobei die Fenstergröße variabel sein soll. Qualitäten ausprobieren!

3.7.3 Appendix

3.7.3.1 Granulartechnik als Synthesizer

Die Granularsynthese ist auch als Synthesizer für Tonschwärme verwendbar, am einfachsten per Zufallsgenerator:

patches/3-7-3-1-granularsynthesizer.pd



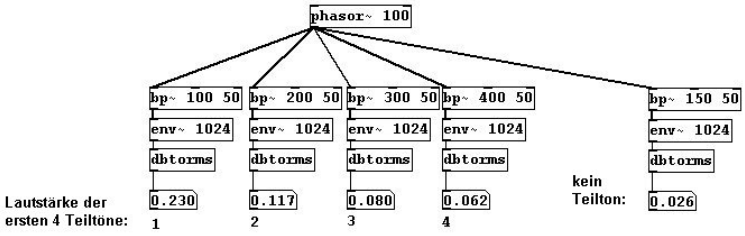
3.8 Fourieranalyse

3.8.1 Theorie

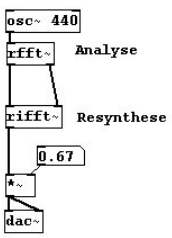
3.8.1.1 Teiltöne analysieren

Erinnern wir uns an die additive Synthese: Ein Klang wird aus Teiltönen zusammengestellt. Wollen wir nun umgekehrt die Bestandteile von einem bestehenden Klang in Erfahrung bringen, können wir für jeden Teilton ein Set von Bandpassfiltern anbringen:

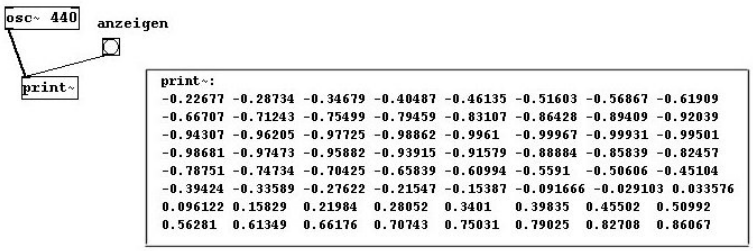
patches/3-8-1-1-teiltoene-analysieren.pd



Dieses Verfahren vollzieht die Fourier Transformation. Sie teilt das gesamte Frequenzspektrum in gleich große Teile und ermittelt in jedem dieser Teile die Amplitude und die Phase. Aus diesen Werten kann danach wieder das Ausgangssignal rekonstruiert werden. Die Ermittlung nennt man *Analyse*, die Rekonstruktion *Resynthese*. Wir realisieren dies mit den Objekten „rfft~“ und „irfft~“:

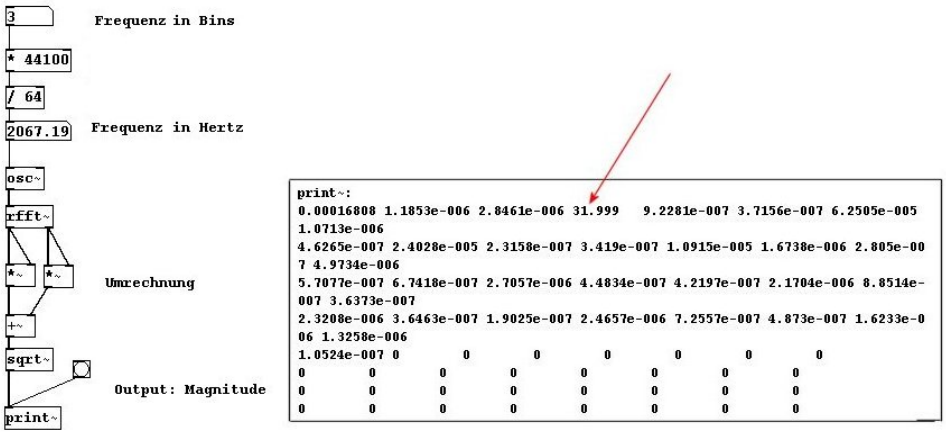


Die Größe der einzelnen Abschnitte, genannt Bins, ist durch die Block Size gegeben. Wie in Kapitel 3.1.1.3.2 besprochen, arbeitet Pd alle Aufgaben immer in Blöcken ab. Normalerweise beträgt die Bock Size in Pd 64 Samples. Mit „print~“ können wir alle Werte eines Blocks sehen:



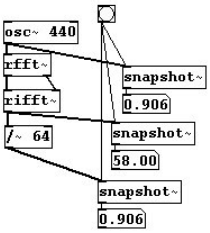
Ähnlich wie bei „snapshot~“ oder „unsig“ sehen wir die produzierten Amplitudenwerte. Bei „print~“ sehen wir tatsächlich ALLE erzeugten Werte, in der Anzahl eben begrenzt auf einen DSP-Block. Bleiben wir vorerst bei den 64 Samples; das heißt, das gesamte Spektrum bis 44100 Hertz wird in Bins zu je 44100/64 = 689 Hertz unterteilt. Als nächstes müssen wir in Betracht ziehen, dass die Daten von Amplitude und Phase bei fft nicht in der uns gewohnten Weise ausgegeben werden; sie erscheinen als Sinus- und Cosinuswerte. Dies wollen wir vorerst nicht weiter verfolgen. Begnügen wir uns damit, dass wir die Amplitude wie folgt in eine uns verständliche Weise umrechnen können:

patches/3-8-1-1-rfft1.pd



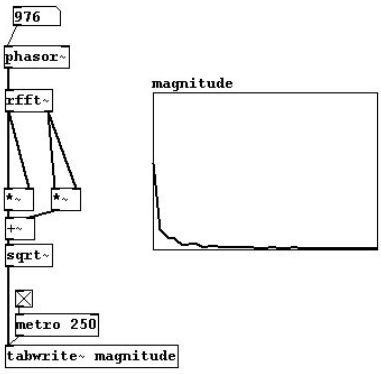
Wie wir sehen erhalten wir bei „print~“ für die Amplitude 64 Werte. Die Amplitude wird hier als Magnitude angegeben, als stets positiver Wert (aufgrund der Quadrierung). Schauen wir genau hin: Mit Ausnahme des dritten Bins, das einen Wert von (ca.) 32 hat, haben wir lauter minimal kleine Werte. Oberhalb der Nyquistfrequenz erfolgt gar keine Rechnung.

Üblicherweise führt man am Ende einer fft-Prozedur eine Normalisierung durch, da die Amplitudenwerte relativ hoch werden. Vorerst ist diese die Größe der Block Size:



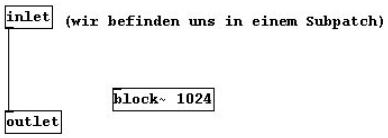
Wir können die fft-Analyse in einem Array darstellen:

patches/3-8-1-1-rfft-array.pd



So sehen wir nun also das Spektrum eines Signals. Halten wir fest: fft macht aus einer Information in der Zeit eine Information in Frequenzen; diese wird in jedem neuen Block aktualisiert. Man spricht von *time domain* und *frequency domain*.

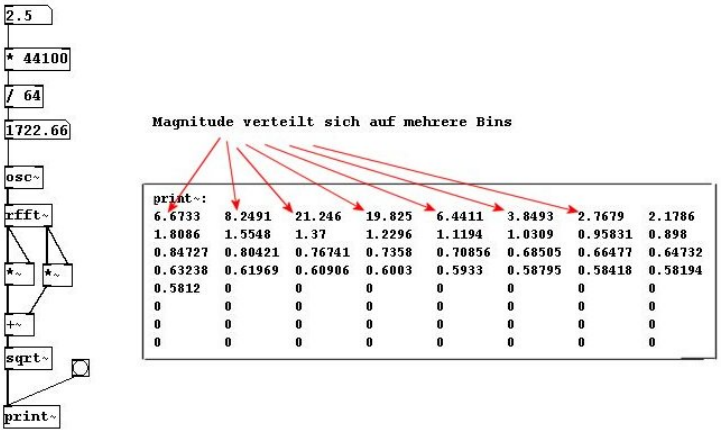
Die Block Size können wir bei Pd nur in einem Subpatch ändern. Dies funktioniert mit dem Objekt „block~“:



Bei der Wahl der Blocksize gilt zu bedenken: Mit einer größeren Blocksize können wir tiefere Frequenzen erfassen; also beispielsweise mit einer Größe von 1024 Samples ist jedes Bin 44100/1024 = ~43 Hz groß, haben wir also eine feinere Auflösung. Dafür dauert die Prozedur länger.

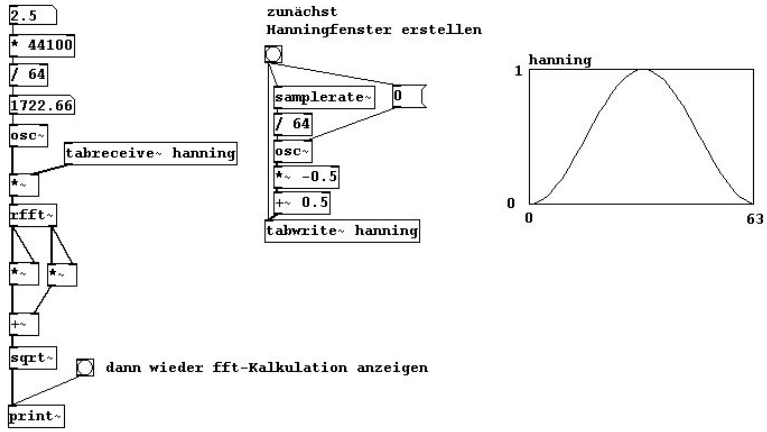
3.8.1.2 Beliebiges analysieren

Bleiben wir bei 64 Samples als Block Size, womit wir Vielfache der Grundfrequenz 689 Hz analysieren können. Was aber, wenn Frequenzen dazwischen auftreten?

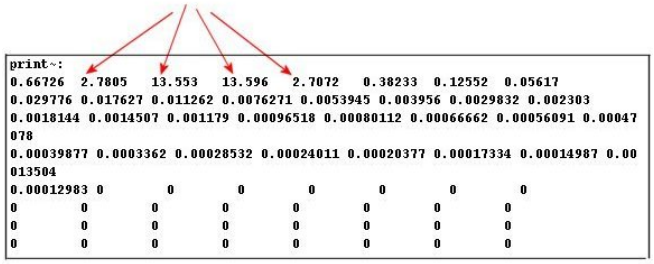


Dann verteilt sich die Information auf mehrere Bins und die Phase ändert sich bei jedem Analysedurchgang. Dieses Problem kann nicht gänzlich gelöst werden, man muss dazu etwas tricksen. Üblicherweise verwendet man hierzu überlappende Fenster wie in der Granularsynthese. Wir erstellen eine gefensterte Version des Originals. Dazu wir verwenden wir „tabreceive~“, das den angegeben Array immer genau in Block Size mit einem Hanning-Fenster liest, also hier in 64 Samples.

patches/3-8-1-2-rfft3.pd

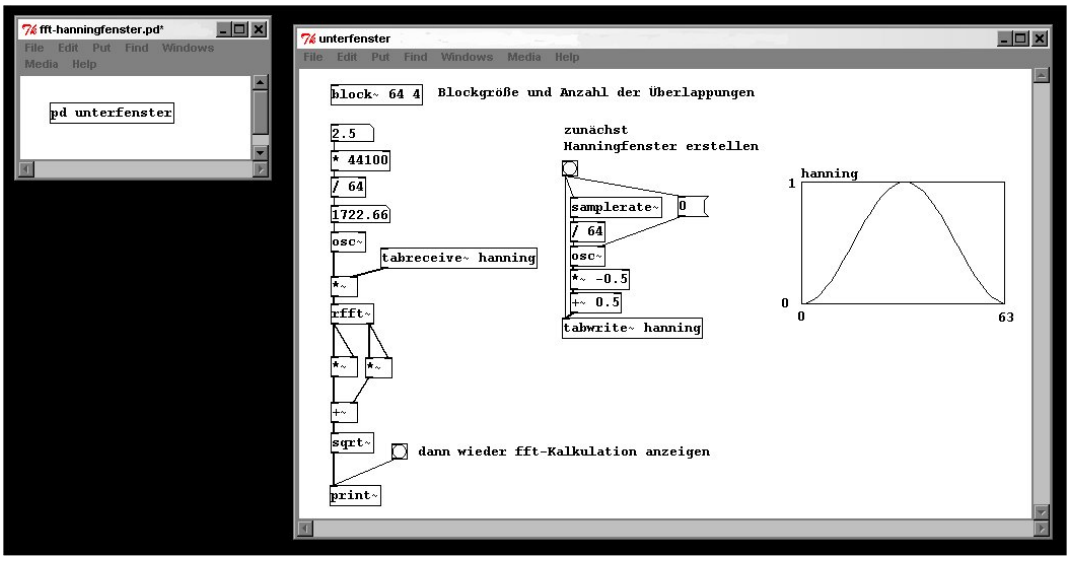


Dadurch „verbreitern“ sich die Magnitudenwerte immerhin nicht mehr so stark.



Zusätzlich zur Fensterung lassen wir noch die Fenster überlappen; das ist bei fft in Pd ganz leicht: Als zweites Argument von „block~“ geben wir die Anzahl der Fenster an, üblicherweise 4. Erforderlich ist noch, das Resultat am Ende wieder zu fenstern. Die passende Normalisierung bei 4 überlappenden Fenstern ist (3 * Blocksize) / 2. Das Ganze muss nun, da wir „block~“ anwenden, in einem Unterfenster passieren.

patches/3-8-1-2-fft-unterfenster.pd



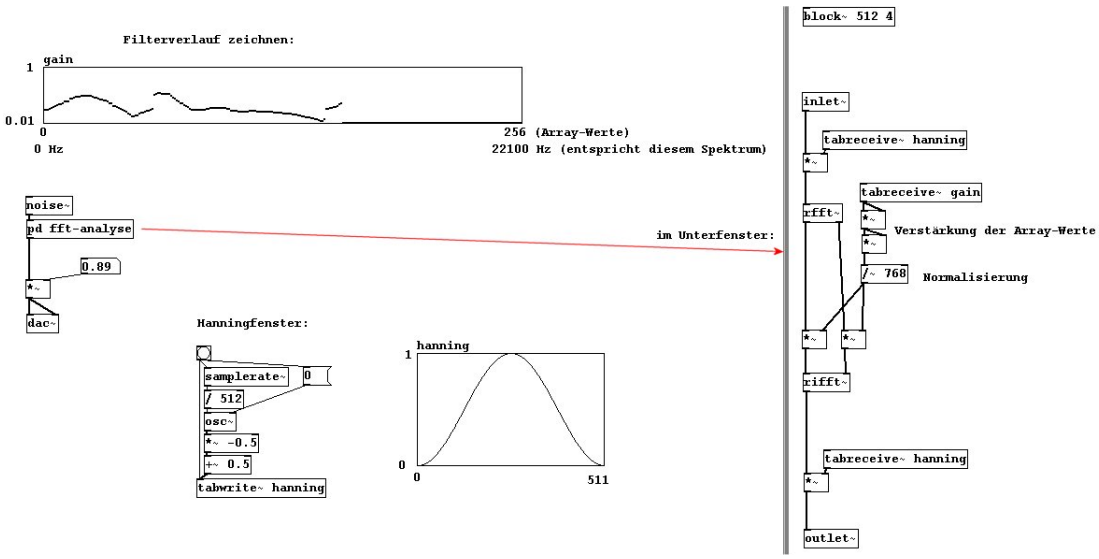
Durch Überlappung und Fensterung stehen die Chancen nun gut, ein Signal korrekt zu erfassen.

3.8.2 Anwendungen

3.8.2.1 Filter

Der Nutzen von fft ist natürlich, die aus der Analyse gewonnen Werte zu ändern, ehe man sie wieder in ein klingendes Resultat verwandelt. Beispielsweise können bestimmte Bins leiser oder ganz aus gestellt werden. So kann man sich Filter wie Hipass, Lowpass etc. selbst 'zeichnen'.

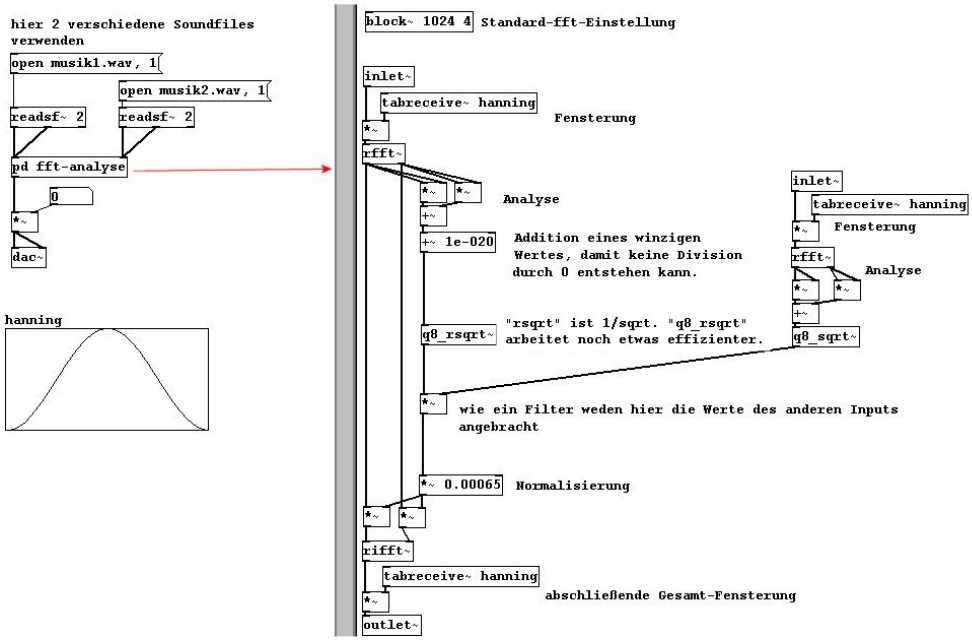
patches/3-8-2-1-fft-filter.pd



3.8.2.2 Faltung

Beliebt ist die Convolution – die Faltung eines Signals mit einem anderen; das heißt, nur die Schnittmenge beider Amplituden werden gespielt. Die Hanning-Kalkluation sollte nun bekannt sein. Standard ist eine Block Size von 1024 Samples und vier Überlappungen.

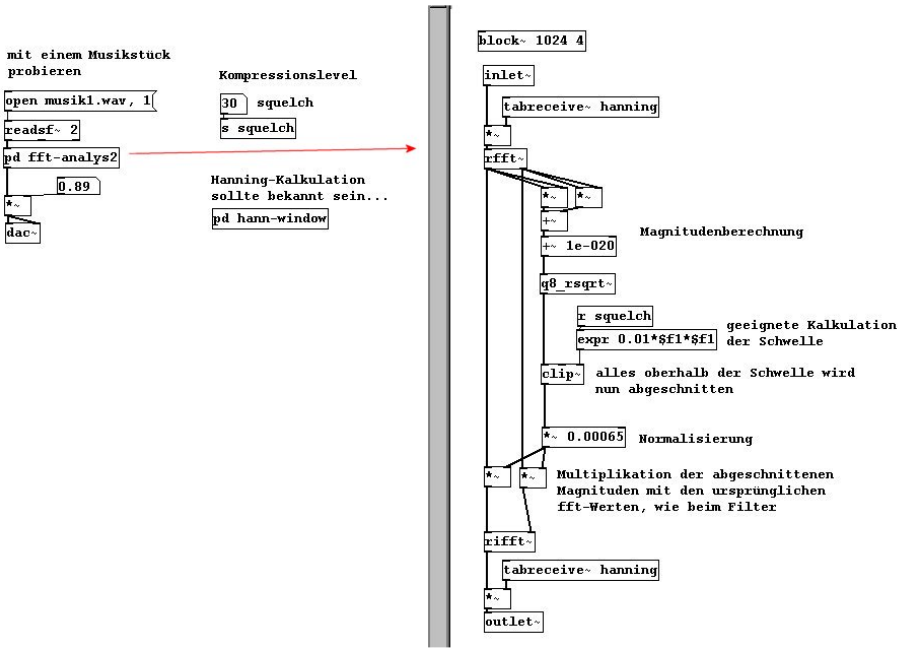
patches/3-8-2-2-faltung.pd



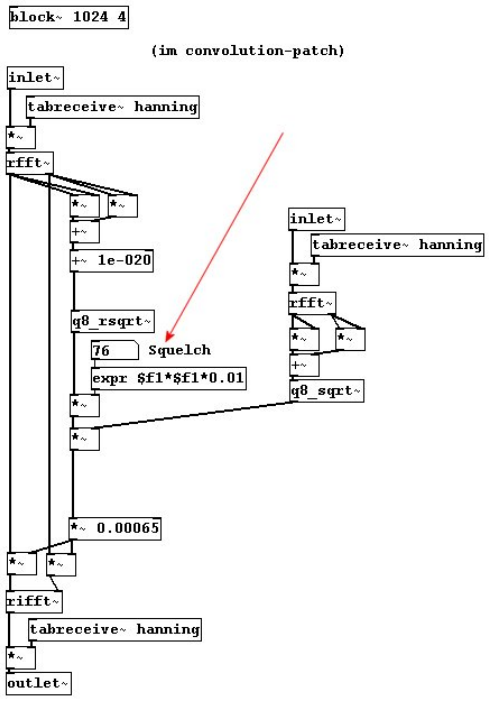
3.8.2.3 Kompressor

Wir können auch einen Kompressor bauen. Das heißt, dass schwächere Lautstärken an die stärkeren mehr angeglichen werden. Wir verwenden die Magnitudenwerte einfach als Faktoren für die Outputs von „rfft“, wobei Werte, die eine gewisse Schwelle („squelch“) übersteigen, einfach an dem Punkt abgeschnitten werden:

patches/3-8-2-3-kompressor.pd



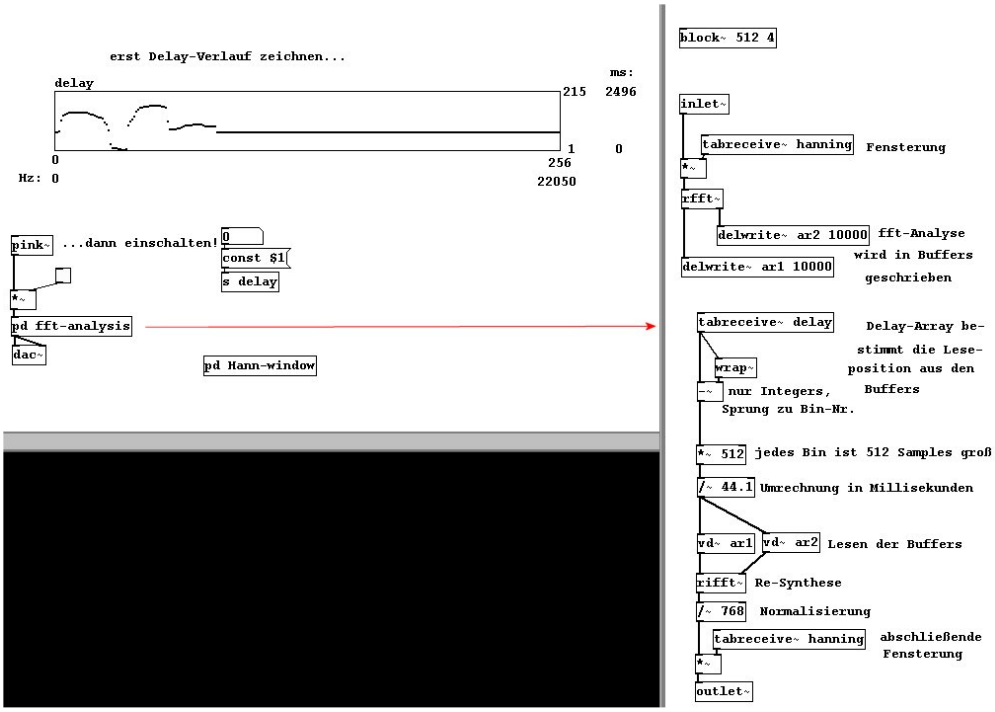
Implementiert man dies bei der Faltung in eine der beiden Analysen, erhalten wir einen reichhaltigeren Convolution-Effekt:



3.8.2.4 Spectral Delay

Wir können auch verschiedene Bins mit verschiedenem Delay abspielen, und erhalten dadurch ein „spectral delay“. Die fft-Analyse wird in zwei Buffer geschrieben. Mit einem Array bestimmen wir dann für jedes Bin, mit welcher Verzögerung es gespielt werden soll. Maximale Delay-Zeit ist ca. 2500 Millisekunden, denn wir haben einen Buffer von 10000 Millisekunden, aber 4fache Überlappung („block~“), darum 10000/4. Genauer gesagt sind es 2496 Millisekunden, das sind 2496 * 44.1 = ca. 110080 Samples, das entspricht 110080 / 512 = 215 möglichen Bin-Positionen. Da in den meisten Fällen das Eingangssignal nicht in die Bin-Größe passt, verteilen sich die Analysewerte auf mehrere benachbarte Bins (vgl. 3.8.1.2). Darum kommt es, wenn diese benachbarten Bins zu verschiedenen Zeitpunkten auftreten, zu Lautstärkeminderungen.

patches/3-8-2-4-spectral-delay.pd

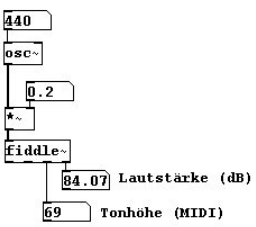


Probieren Sie dies einmal mit einer bewegten Musik aus!

3.8.3 Appendix

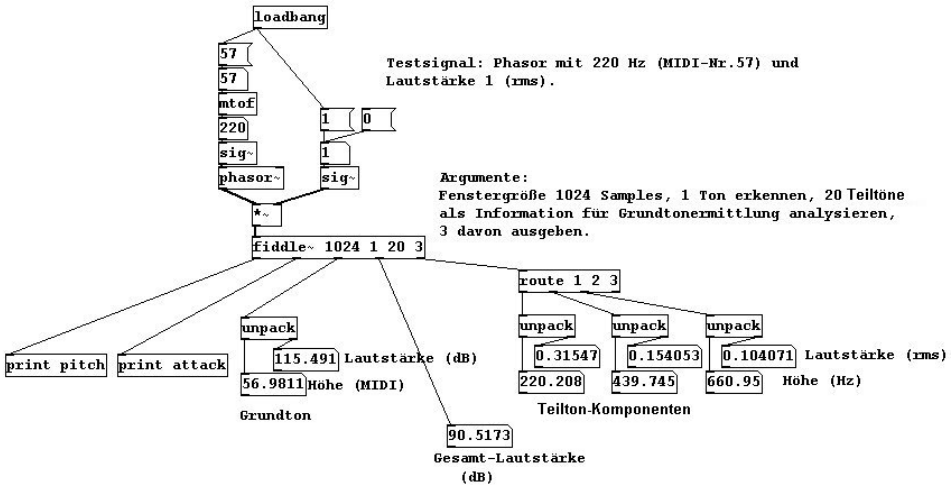
3.8.3.1 fiddle~

Basierend auf dem fft-Algorithmus gibt es in Pd ein Objekt zur Erkennung von Lautstärke UND Tonhöhe. Es heißt „fiddle~“. Außerdem gibt es die Lautstärken der Teiltöne des Eingangssignals aus.

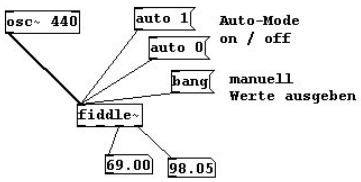


Als Argumente erhält es: 1. Fenstergröße (in Samples), 2. Anzahl der gleichzeitig zu erkennenden Töne (maximal drei verschiedene), 3. Anzahl der zu erkennenden Teiltöne, 4. Anzahl der auszugebenden Teiltöne. Default sind: 1. 1024, 2. 1, 3. 20, 4. 0. Als Outputs erhalten wir (von links nach rechts): 1. Tonhöhe in MIDI (nur bei Änderung), 2. Lautstärke in dB (nur bei starker Änderung („attack")), 3. Tonhöhe und Lautstärke des Grundtones als Liste, 4. die Lautstärke insgesamt, 5. die einzelnen Teiltöne mit Lautstärke (in Hertz / rms!) als Liste.

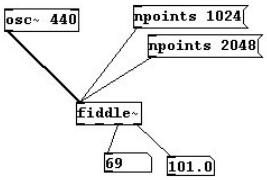
patches/3-8-3-1-fiddle.pd



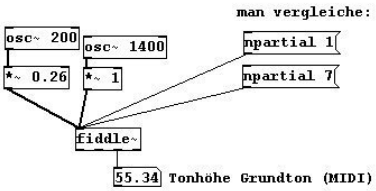
Messages für „fiddle~": Damit nicht andauernd Werte ermittelt werden, kann der „Auto-Mode" augeschaltet und statt dessen der „Poll-Mode" aktiviert werden, bei dem nur Zahlen ausgegeben werden, wenn man mit dem Objekt eine "bang"-Message sendet:



Wir können die Fenstergröße bestimmen (Vielfache von zwei):



Höhere Teiltöne werden zur Ermittlung der Grundfrequenz weniger stark untersucht. Das kann aber geändert werden, in dem wir das Objekt anweisen, einen bestimmten Teilton zumindest mit der Hälfte der Stärke der Grundtonuntersuchung zu behandeln:



Als Grundfrequenz gehen wir von 200 Hz. Doch den Grundton spielen wir nur leise, dazu den 7.Teilton (1400 Hz) laut. Zunächst geht fiddle davon aus, dass der lautere Ton der Grundton ist, bis wir angeben, dass hier der 7. Oberton besonderes Gewicht hat, woraus fiddle dann schließt, dass der Ton um 1400 Hz also der 7.Teilton sein muss, zur Grund-frequenz 200 Hz.

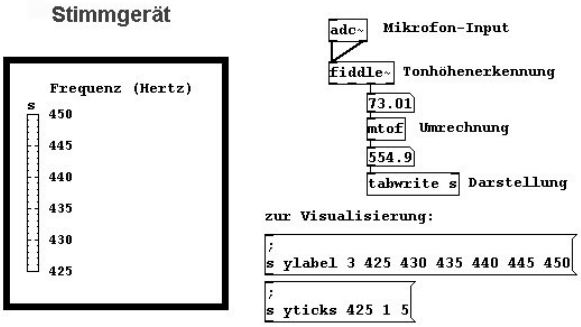
Dies ist hilfreich, wenn wir wissen, dass einige Teiltöne des Eingangssignals besonders stark sind (z. B. bei einer Klarinette der dritte Teilton).

Es gilt zu beachten: Eine Analyse des Eingangssignals wird alle halbe Fenstergröße durchgeführt, d. h. bei einer Fenstergröße von 1024 Samples alle 512 Samples, also alle 11.6 Millisekunden. Die minimale Frequenz, die „fiddle~“ erkennen kann ist, ist (44100 / Fenstergröße) * 2.5, also bei 1024 Samples ca. 108 Hertz.

3.8.3.2 Stimmgerät

So können wir uns beispielsweise ein Stimmgerät erstellen:

patches/3-8-3-2-stimmgeraet.pd

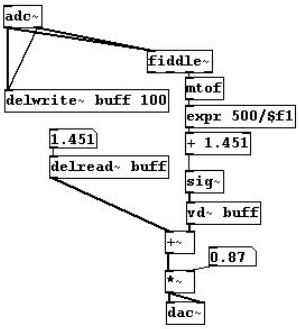


Zur Visualierung wird hier ein Array mit nur einem Speicherplatz verwendet.

3.8.3.3 Oktavdoppler #2

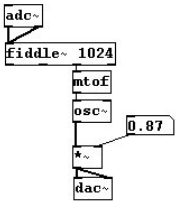
Für den unter 3.4.2.9 beschriebenen Oktavdoppler kann man nun auch einen Mikrofoninput verwenden, sofern sich von diesem der Grundton grundsätzlich ermitteln lässt (am Eingang also ein periodisches Signal anliegt, das „fiddle~“ erfassen kann):

patches/3-8-3-3-oktavdoppler-fiddle.pd



3.8.3.4 Pitch Follower

Für das fiddle~ Objekt sind auf diese Weise viele interessante Anwendungen vorstellbar. Prototypisch ist, dass wir einen Mikrofoninput, etwa eine Singstimme haben und mit einem Sinuston die Melodie der Stimme wie mit einem Laserpointer quasi 'nachzeichnen':



Folgendes Dilemma wird hier ersichtlich: Bei „fiddle~“ ergibt sich immer eine Verzögerung. Diese ist umso kürzer, je kleiner die Fenstergröße ist. Doch je kleiner die Fenstergröße, desto weniger tief nach unten reicht die Erkennung der Tonhöhen. Außerdem ist das Resultat von „fiddle~“ immer auch etwas chaotisch. Wie man dies minimieren kann, erfahren wir unter 4.3.1.3

3.8.3.5 Weitere Aufgabenstellungen

Erstellen Sie statt dem einfachen 'Nachzeichnen' nun eine parallele Stimme im Quintabstand oder einen Akkord parallel.

3.9 Amplitudenkorrekturen

3.9.1 Theorie

Zum Abschluss des großen Kapitels über Audiotechniken in Pd behandeln wir nun noch Verfahren der Amplitudenbearbeitung.

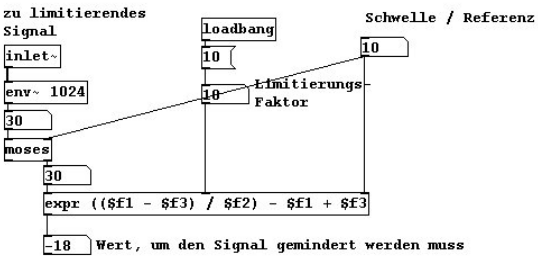
3.9.1.1 Limiter

Wie wir unter 3.1.2.1.2 erfahren haben, kann die Lautsprechermembran nur bis zu einer gewissen Grenze schwingen, danach wird die Welle abgeschnitten („geclipped“). Wir können nun einen

Automatismus bauen, der zu laute Signale rechtzeitig abdämpft. Ein solches Gerät wird in der Tontechnik als Limiter bezeichnet.

Beim Limiter haben wir nun ein Eingangssignal, dessen Lautstärke gemessen wird. Überschreitet sie die zuvor eingestellte Obergrenze, wird alles darüber mit dem eingestellten Reduktionsfaktor zum Referenzpunkt hin gedämpft. (Im Folgenden werden wir bei der Lautstärke mit dB rechnen.)

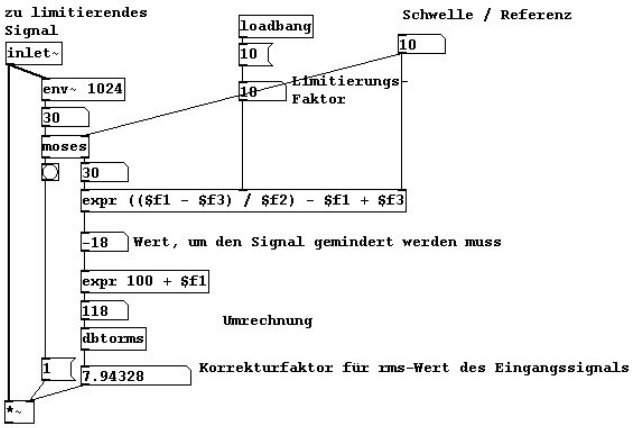
patches/3-9-1-1-limiter1.pd



Im ersten Beispiel ist die Schwelle 10 dB, der Faktor 10 und das Eingangssignal 30 dB laut. Seine Differenz zur Referenz beträgt 20 dB; der Faktor bestimmt, dass diese Differenz um das 10-fache reduziert werden soll auf 2 dB, das heißt vom Eingangssignal muss um -18 dB auf 12 dB gemindert werden.

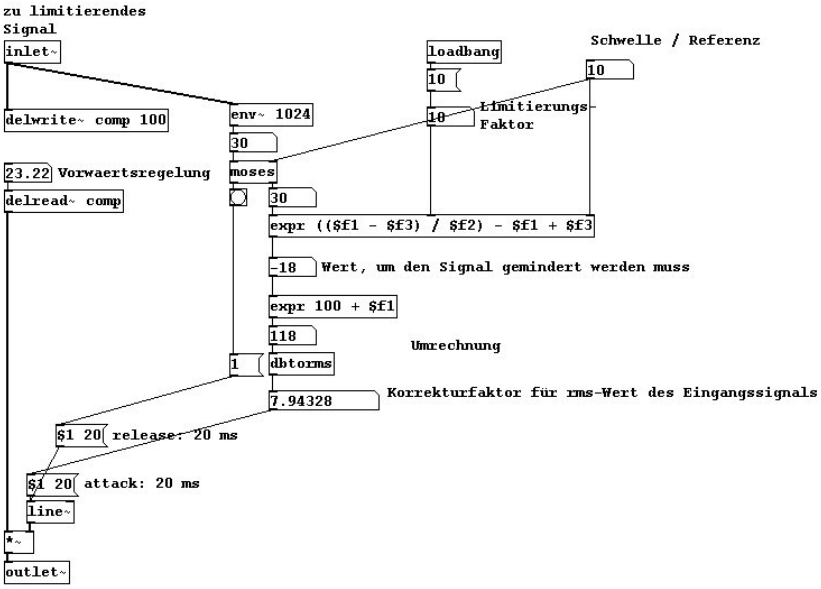
Nun führen wir dem Originalsignal, immer wenn es die Schwelle („Threshold“) übersteigt, den Limitierungsfaktor zu (bleibt es darunter, erhält es einfach den Faktor 1):

patches/3-9-1-1-limiter2.pd



Beachten wir dabei zwei Aspekte: „env~“ macht eine Durchschnittsrechnung innerhalb der angegebenen Samples, hier immer in einem Zeitfenster von 1024 Samples. Das ergibt eine Verzögerung von 1024 / 44.1 = 23.22 Millisekunden, die wir einstellen können. Wir können aber auch die Reduktion schon beginnen lassen, ehe das Signal die Schwelle überschreiten wird, das heißt das Originalsignal noch länger verzögern. In der Signalverarbeitung ist der Fachausdruck dafür „Vorwärtsregelung“. Ist die Verzögerung des Originalsignals geringer bzw. die Verzögerung der Korrektur größer, spricht man von „Rückwärtsregelung“. Dabei kommt es aber kurz zum Überschreiten der Schwelle, ehe die Korrektur einsetzt. Andererseits stellt sich die Frage, in welcher Geschindigkeit die Korrektur vollzogen werden soll und ebenso in welcher Geschwindigkeit nach Rückkher unter die Schwelle wieder zur originalen Lautstärke zurückgekehrt werden soll. Man spricht von „attack“ und „release“-Zeiten.

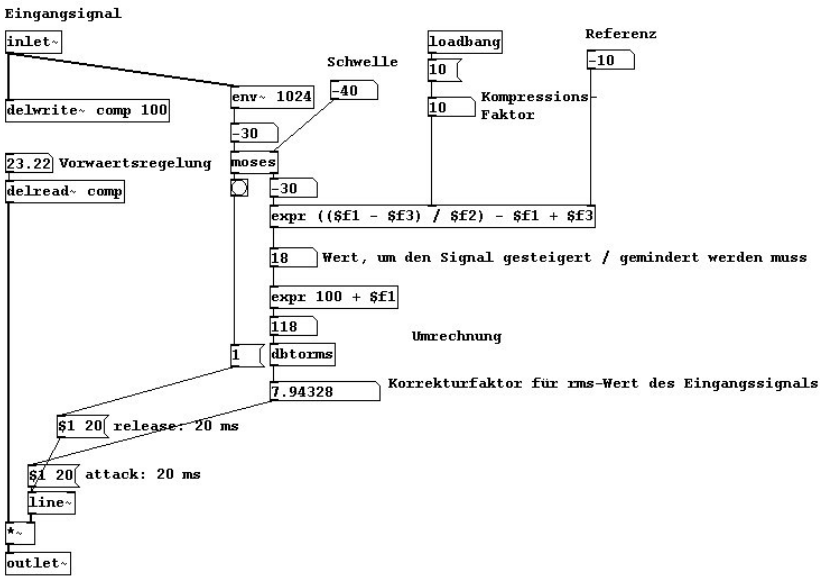
patches/3-9-1-1-limiter3.pd



3.9.1.2 Kompressor

Wenn wir nun noch differenzieren zwischen dem Schwellenwert, ab dem eine Korrektur vorgenommen werden soll und dem Referenzpunkt, zu dem hin korrigiert wird, können wir auch einstellen, dass Lautstärken unterhalb des Referenzpunktes zum Referenzpunkt hin lauter gedreht werden, so wie sie oberhalb des Punktes leiser gestellt werden. Solch ein Gerät wird Kompressor genannt:

patches/3-9-1-2-kompressor.pd



3.9.2 Anwendungen

3.9.2.1 Larsen Tones

Schließt man den Mikrofoninput an den Lautsprecher an, also den „adc~“-Output direkt an den „dac~“und hält dann das Mikrofon an den Lautsprecher, erklingt bald ein Ton (dann schnell das Mikrofon wegnehmen!). Dies liegt daran, dass die Luft immer etwas rauscht, dieses Rauschen dann vom Mikrofon zum Lautsprecher gelangt, aus dem Lautsprecher ins Mikrofon etc. Dabei wird, abhängig vom Abstand zwischen Mikrofon und Lautsprecher, das Signal bei jedem Durchlauf etwas verstärkt. So kommt es dann, je nach Raum, Kabellänge und Latenz im Computer zu verschieden hohen periodischen Tönen, auch „Larsen Tones“ genannt. Zugleich nimmt die Lautstärke sehr schnell extrem zu, da ja ständig weiter verstärkt wird. Das ist der klassische Fall der Rückkopplung, ein Kreislauf, ein rekursives System. Wo immer man mit Mikrofonen und Verstärkung zu tun hat, ist man mit der Gefahr von Rückkopplungen konfrontiert. Dem kann beispielsweise ein dazwischengeschalteter Limiter entgegenwirken.

3.9.2.2 Weitere Aufgabenstellungen

a) Erstellen Sie einen „Expander“: Schwache Amplitudenunterschiede sollen starke Unterschiede werden!

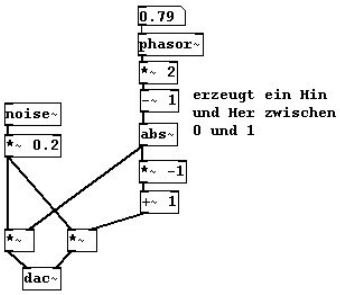
b) Eine Lautstärkeninvertierung: Leises wird laut, Lautes wird leise.

3.9.3 Appendix

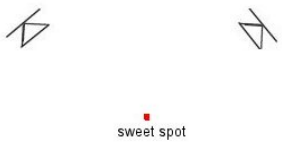
3.9.3.1 Raumbewegungen

Mit der Lautstärke können wir auch Raumbewegungen simulieren. Normalerweise haben wir ein Stereo-Lautsprecherpaar und darum zwei Inputs für den „dac~“. Wenn wir nun allmählich mit der Lautstärke zwischen beiden Eingängen wechseln, erleben wir, sofern wir uns genau zwischen den beiden Lautsprechern befinden, wie der Klang zwischen den beiden Lautsprechern hin und her 'wandert'. Dies bezeichnet man als Phantomschallquelle.

patches/3-9-3-1-raum-stereo.pd

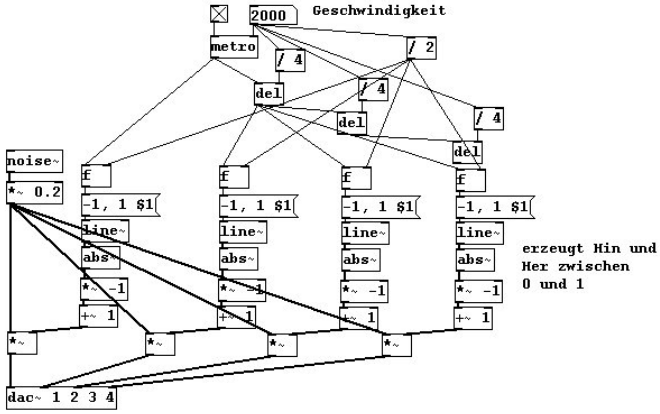


Lautsprecher-Aufstellung:

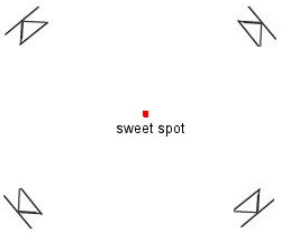


Hat man z. B. vier Lautsprecher zu einem Quadrat aufgebaut, kann man Kreisbewegungen durch den Raum erzeugen (dazu braucht man natürlich auch eine Soundkarte mit vier separaten Ausgängen; als Argumente kann man dann dem „dac~“ die Inputs angeben):

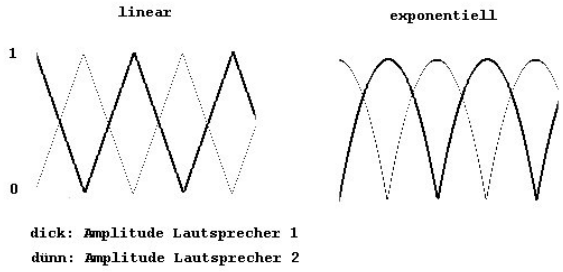
patches/3-9-3-1-raum-quadro.pd



Aufstellung:



Auch hier funktioniert der Effekt aber nur dann richtig, wenn man sich genau in der Mitte („sweet spot“) befindet. Zudem hört man im obigen Beispiel noch ein deutliches 'Loch' in der Lautstärke zwischen den beiden Höchstwerten der Lautstärke der jeweiligen Lautsprecher. Hier gilt es zu experimentieren, wie die Überlappung der Lautstärken sein soll, ob linear oder beispielsweise exponentiell (vgl. die Fenstertypen unter 3.9.4). Das muss das Ohr des Komponisten entscheiden.

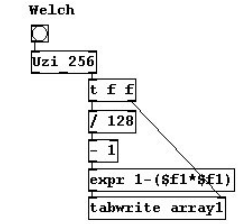


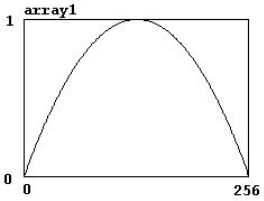
3.9.4 Für besonders Interessierte

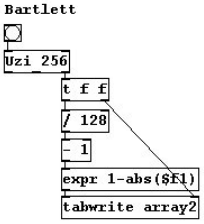
3.9.4.1 Andere Fenster

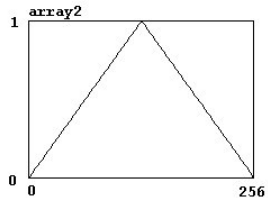
Um Klicks zu vermeiden, wurde in den vergangenen Kapiteln immer wieder das „Hanning“-Fenster verwendet, das einem Ausschnitt aus der Cosinusfunktion entspricht. Es gibt aber auch noch andere Fenstertypen, mit denen Interessierte hörend experimentieren mögen:

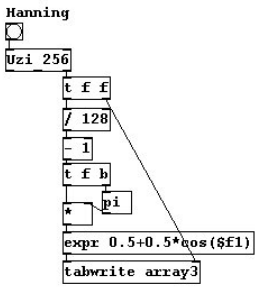
patches/3-9-4-1-windowing.pd

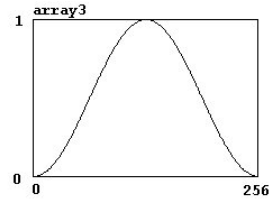


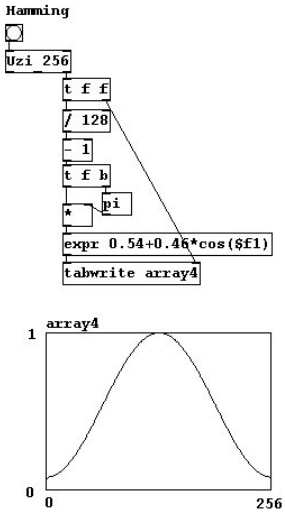


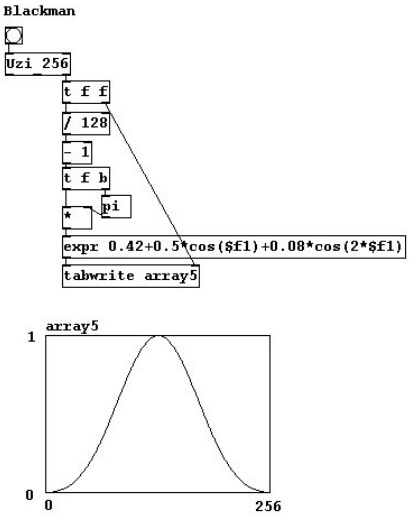


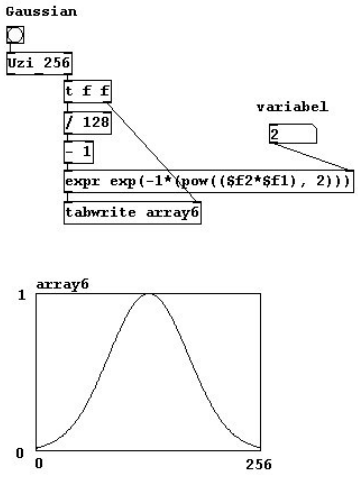







Hamming
Uzi 256
t f f
/ 128
- 1
t f b
pi
*
expr 0.54+0.46*cos($f1)
tabwrite array4
array4
1
0
0
256


Blackman
Uzi 256
t f f
/ 128
- 1
t f b
pi
*
expr 0.42+0.5*cos($f1)+0.08*cos(2*$f1)
tabwrite array5
array5
1
0
0
256


Gaussian
Uzi 256
t f f
/ 128
- 1
variabel
2
expr exp(-1*(pow(($f2*$f1), 2)))
tabwrite array6
array6
1
0
0
256

Kapitel 4. Klangsteuerung

Musik läuft in der Zeit ab und ein Komponist möchte natürlich gerne, dass sich im Laufe der Zeit etwas klanglich ändert. Im vorigen Kapitel haben wir alle Grundlagen für die Klangerzeugung kennengelernt. Nun befassen wir uns mit den Möglichkeiten von Pd, die erzeugten Klänge bzw. die Klangerzeugung selbst in der Zeit zu steuern.

4.1 Algorithmen

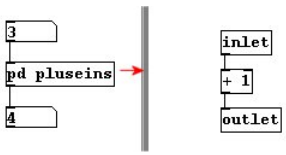
4.1.1 Theorie

4.1.1.1 Was sind Algorithmen?

Algorithmus ist der Fachausdruck für die Beschreibung der Abfolge von Arbeitsschritten, die ein Computerprogramm ausführt.

Haben wir einen Subpatch, der zu einer eingegeben Zahl 1 addiert, kann man schon von einem Algorithmus sprechen: Der Algorithmus des Subpatches ist die Addition um 1.

patches/4-1-1-1-plus-eins-algorithmus.pd



In Grunde läuft in jedem Objekt von Pd ein eigener Algorithmus ab. Was früher ein Gerät wie den Noise-Generator erfordert hat, übernimmt heute der Algorithmus, der in dem „noise~“-Objekt steckt.

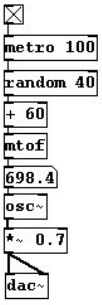
In diesem Kapitel interessiert uns nun speziell, Algorithmen zu entwickeln, die der Computer wie gesagt selbständig ausführt und die den Zweck erfüllen, in der Zeit den Klang zu ändern. Wir haben schon einige Beispiele dafür kennengelernt, etwa unter 2.2.3.2.7

4.1.2 Anwendungen

4.1.2.1 Stochastik

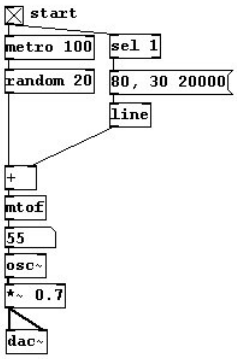
Eine sehr einfache, aber ergiebige Weise, den Computer für sich arbeiten zu lassen, ist die Verwendung des Zufallsgenerators.

patches/4-1-2-1-random.pd

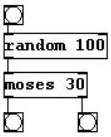


Wir können nun der zufälligen Auswahl Grenzen setzen, die sich aber ändern:

patches/4-1-2-1-random-grenzen.pd

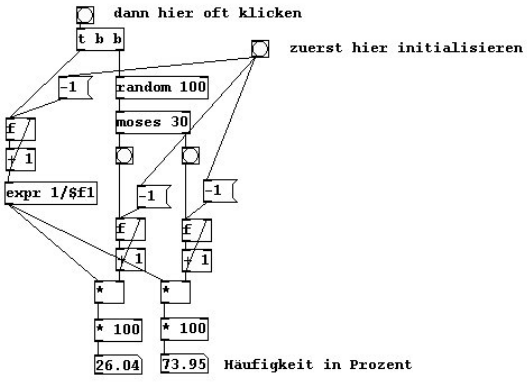


Das Ergebnis eines Zufallsgenerators folgt den Gesetzen der Stochastik, das heißt, der Wahrscheinlichkeit. Bei „random 6“ erscheint jede Zahl von 0 bis 5 mit einer Wahrscheinlichkeit von 1/6. Was dennoch heißen kann, dass eine der Zahlen sehr lange nicht oder nie erscheint, wenn auch das wiederum sehr unwahrscheinlich ist. Wir können die Wahrscheinlichkeit aber auch selbst lenken:



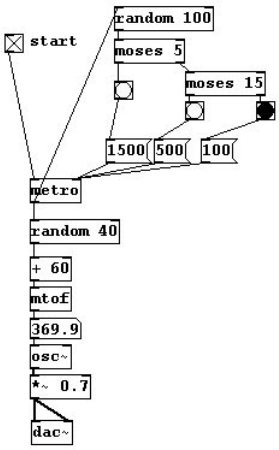
In Prozent ausgedrückt liegt hier die Wahrscheinlichkeit, dass links ein Bang kommt, bei 30 Prozent und dass rechts einer kommt, bei 70 Prozent. Wir können das tatsächliche Aufkommen feststellen:

patches/4-1-2-1-wahrscheinlichkeit.pd

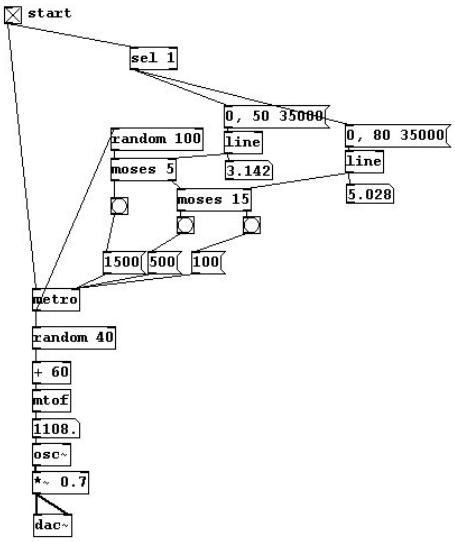


So können wir für unsere Tonpunkte unterschiedliche Dauern differenzieren: Kurze sind sehr häufig, mittlere kommen ab und zu vor, lange selten.

patches/4-1-2-1-wahrscheinlichkeit-beispiele.pd

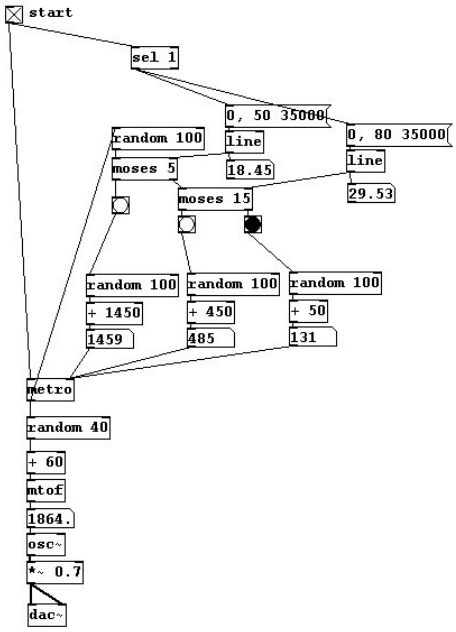


Wir können eine Verteilung auch mit der Zeit umdrehen...



Am Anfang kommen nur schnelle Dauern, am Ende hauptsächlich langsame (die natürlich auch noch besonders viel Zeit einnehmen).

Die verschiedenen Zeitebenen lassen sich noch einmal ein bisschen in sich variieren:



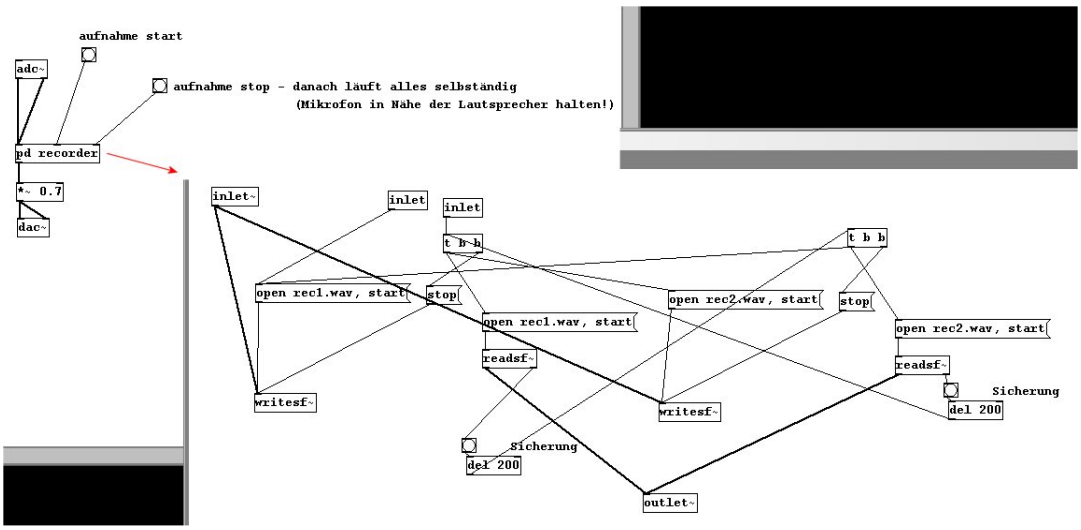
Und so kann man natürlich immer mehr Parameter dem Zufall überlassen.

4.1.2.2 Rekursionen

Von Alvin Lucier gibt es ein Stück mit einer relativ einfachen Idee: Jemand sitzt in einem Raum und spricht etwas in ein Mikrofon. Das Gesprochene wird aufgenommen und danach in dem Raum abgespielt. Das Abgespielte wird wieder aufgenommen und das wiederum abgespielt und wieder aufgenommen und so weiter. Mit jedem Durchgang wird die Qualität der Sprachaufnahme schlechter, es gehen immer mehr Informationen verloren, genauer gesagt: Die Frequenzen, die die Lautsprecher, das Mikrofon und der Raum gut darstellen können, werden immer weiter transportiert, während andere allmählich herausgefiltert werden.

Dies in Pd zu programmieren ist ziemlich einfach:

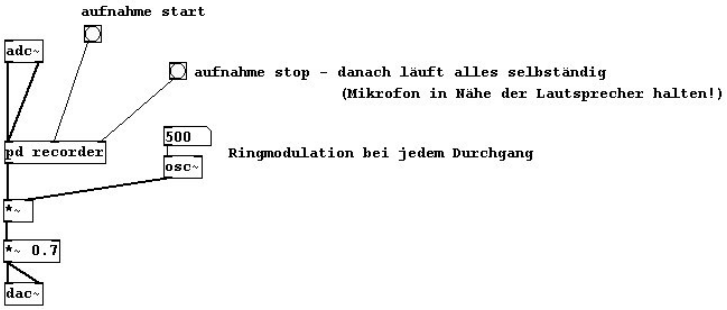
patches/4-1-2-2-lucier.pd



Wir können wieder sagen: Der ablaufende Algorithmus entspricht der Aufnahme und Wiedergabe. In dem Fall wird das Ergebnis eines Algorithmus' wieder in diesen Algorithmus integriert und

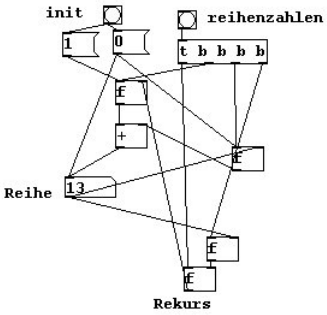
immer so fort. Ein derart automatisch ablaufendes Verfahren nennt man Rekursion. Wir haben schon Rekursionen unter 3.4.2.9 und 3.4.2.10 gesehen.

Im nächsten Beispiel arbeiten wir auch mit Veränderung und Wieder-Aufnahme. Eine rekursive Ringmodulation:



Rekursionen können aber auch allein mit Zahlen interessant sein. Eines der bekanntesten Beispiele dafür, das in der Musik auch häufig vorkommt, ist die Fibonacci-Reihe. Der Algorithmus besteht darin, dass die beiden letzten Zahlen einer Liste addiert werden und das Ergebnis der Liste hinzugefügt wird.

patches/4-1-2-3-fibonacci.pd



4.1.2.3 Weitere Aufgabenstellungen

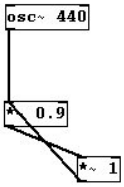
a) Nehmen Sie ein Sample auf, das in falscher Geschwindigkeit abgespielt wird, spielen Sie das Ergebnis wieder falsch ab, nehmen es auf und so weiter. Probieren Sie aus, das Sample in immer gleicher Weise falsch abzuspielen oder aber bei jedem Durchgang anders.

b) Erzeugen Sie einen Waveshaping-Algorithmus, dessen Ergebnis mit Delay wieder oben eingespeist wird.

4.1.3 Appendix

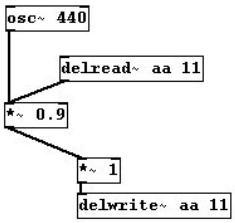
4.1.3.1 DSP Loop

Der rekursiven Verschaltung von Klang sind technische Grenzen gesetzt. Wenn wir Folgendes tun ...



... erscheint die Fehlermeldung „DSP loop detected“ und es wird kein Audio aus dieser Schaltung abgespielt. Ohne zeitliche Verzögerung des Signals können wir keine (Audio-)Rekursion anlegen.

So gibt es keinen Fehler:



4.1.4 Für besonders Interessierte

4.1.4.1 Algorithmisches Komponieren

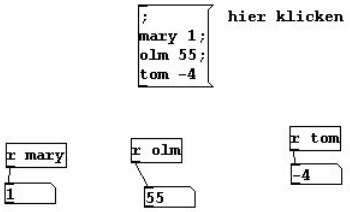
Komponieren mit Algorithmen ist ein weites Feld. Schon bei mittelalterlichen Komponisten sind derlei Rechenprinzipien festzustellen und seit dem 20. Jahrhundert ist dies ein ausgiebig genutzter Bereich in der Musik. Schon allein aus mathematischer Sicht sind algorithmische Kompositionen faszinierend. Auch in der Natur lassen sie sich vielerorts entdecken. Weitere Informationen unter:

http://de.wikipedia.org/wiki/Algorithmische_Komposition

4.2 Sequenzer

4.2.1 Theorie

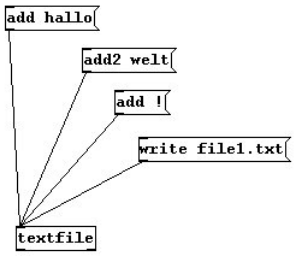
Statt automatisch ablaufender Prozesse können wir aber auch regelrechte „Partituren“ für einen Pd-Patch schreiben. Ein einfaches Beispiel ist die Zusammenfassung mehrerer „send“-Befehle, wie unter 2.2.4.1.3 beschrieben:



Um aber noch viel mehr Informationen so festzulegen und diese wiederum in ihrer zeitlichen Abfolge zu bestimmen, werden im folgenden Kapitel verschiedene Möglichkeiten beschrieben, dies innerhalb von Pd zu realisieren.

4.2.1.1 Textfile

Wir können Zahlen und Symbole von einer einfachen Textdatei abrufen oder darin mit dem Objekt „textfile“ speichern. Schauen wir uns zunächst die Speicherfunktion an. Wir klicken die Messages von oben nach unten an:

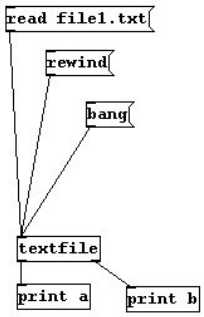


Nun hat Pd eine Textdatei namens „file1.txt“ im Verzeichnis des Patches erstellt. Darin steht:

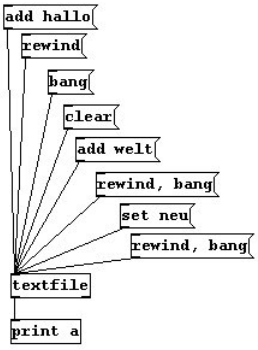
```
hallo;
welt!;
```

„add“ erstellt ein Symbol oder eine Zahl und setzt dahinter ein Semikolon. „add2“ setzt statt eines Semikolons nur ein Leerzeichen.

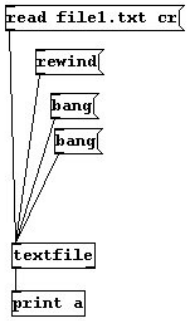
Wollen wir das Gespeicherte nun lesen, laden wir zunächst die Datei, gehen mit „rewind“ ganz an den Anfang, woraufhin mit jedem Bang eine Zeile (bis zum Semikolon) im linken Outlet ausgegeben wird. Ist die letzte Zeile erreicht, kommt aus dem rechten Outlet ein Bang.



Wir können mit einem Objekt schreiben und dies dann wieder lesen, ohne es als Datei zu speichern. Zudem kann alles mit „clear“ gelöscht werden. Mit „set“ wird erst alles gelöscht und dann eine neue Zeile erstellt. Man klicke von oben nach unten:



Wir können eine Datei auch so laden, dass die Semikola nicht erscheinen:

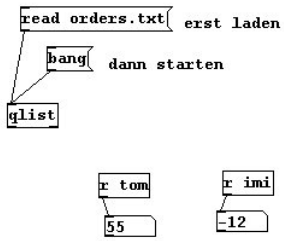


Auf gleiche Weise funktioniert auch „write name.txt cr“.

4.2.1.2 Qlist

Eine praktische Erweiterung von „textfile“ ist „qlist“. Hiermit können wir zeitlich organisierte Nachrichten mit einer Textdatei an „receive“-Objekte senden. Wir haben die Datei „orders.txt“ mit dem Inhalt:

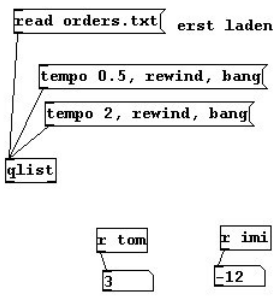
```
0 tom 55;
1000 imi -12;
4000 tom 3;
2000 imi -2;
```



Zu Beginn erhält „tom“ die Zahl 55, eine Sekunde später erhält „imi“ -12, und wiederum vier Sekunden danach „tom“ 3, zwei Sekunden danach „imi“ -2. Genauso funktioniert das mit Symbolen.

Ansonsten hat „qlist“ Funktionen wie „textfile“: add, add2, rewind, clear.

Außerdem lässt sich noch das Tempo modifizieren mit „tempo“ und einem Faktor:

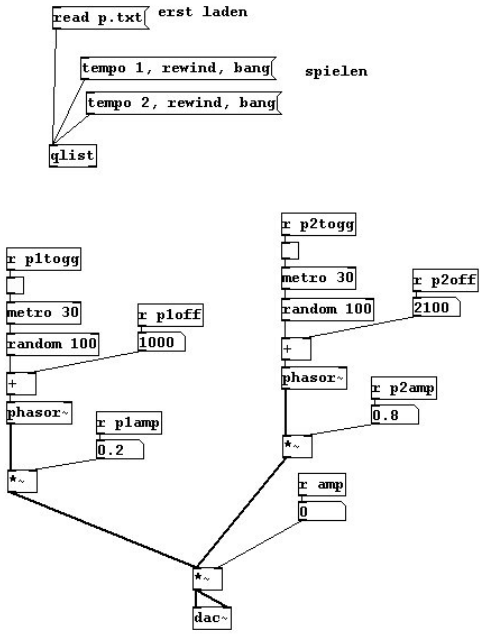


4.2.2 Anwendungen

4.2.2.1 Partitur für einen Patch

Nun können wir mit vorab zusammengestellen Klängen ein Musikstück als Textdatei schreiben. Wir haben diesen Patch ...

patches/4-2-2-1-partitur.pd

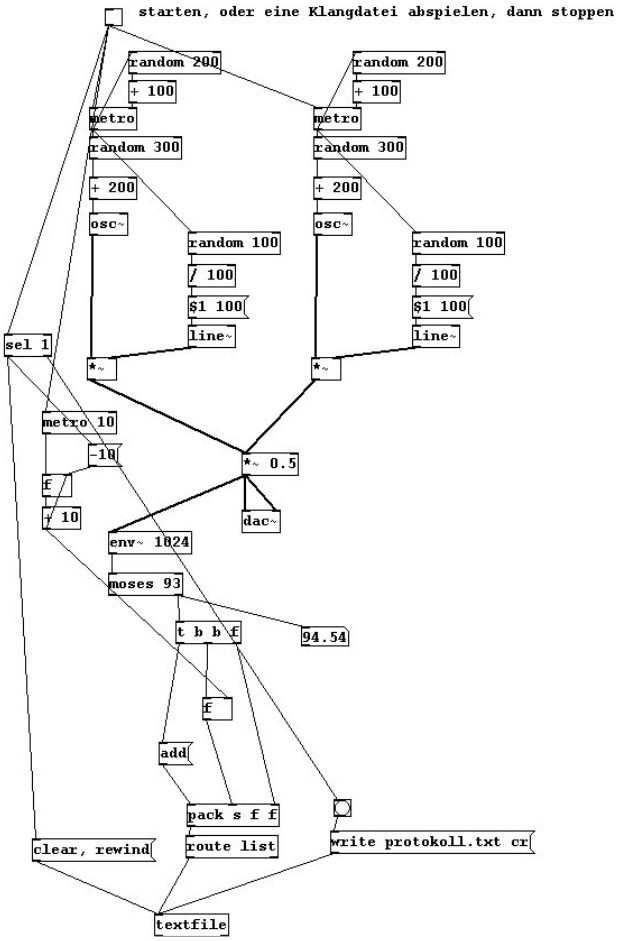


... mit dieser „Partitur“ (patches/p.txt):

```
0 p1off 1000;
0 p1togg 1;
0 p1amp 1;
0 amp 0.5;

3000 p2off 100;
0 p2togg 1;
0 p2amp 1;

2000 p2off 400;

3000 p1amp 0.2;

3000 p2amp 0;

1000 p2off 2100;
0 p2amp 0.8;


5000 amp 0;
0 p1togg 0;
0 p2togg 0;
```

Wir können auch in eine Textdatei Informationen aus Klängen schreiben:

patches/4-2-2-1-partitur-schreiben.pd



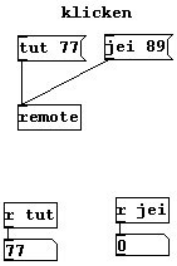
4.2.2.2 Weitere Aufgabenstellungen

Schreiben Sie Zufallsalgorithmen in eine Textdatei, mit der sich per „qlist“ der Patch von 4.2.2.1 (in verschiedenen Geschwindigkeiten) abspielen lässt.

4.2.3 Appendix

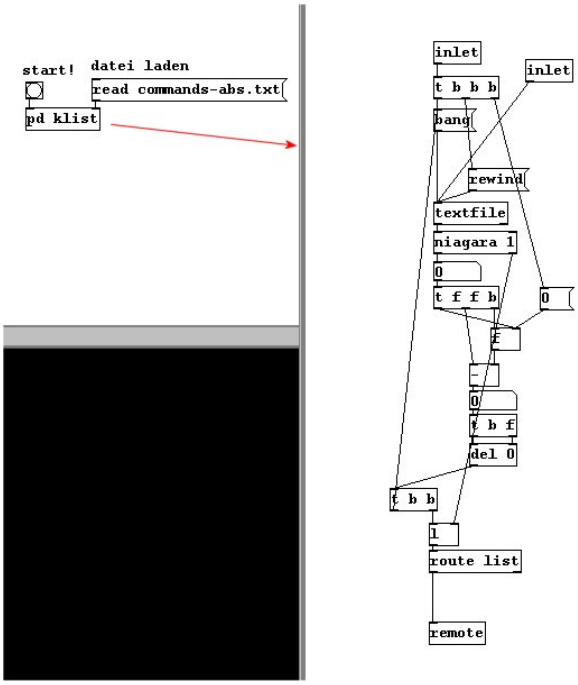
4.2.3.1 Modifikation von qlist

Die Zeiten für qlist sind Delta-Werte, das heißt, wir geben immer den Abstand von einem Ereignis zum nächsten an. Manches Mal wäre es aber auch praktisch eine Textdatei mit absoluten Zeitangaben zu verwenden. Mit Hilfe von „remote“ können wir diese in der gleichen Weise gebrauchen. Das Objekt „remote“ (das nicht in der ursprünglichen Pd-Version enthalten ist, sondern Teil der Maxlib-Library in Pd-extended ist) erhält als Liste den Namen eines receive-Objekts und dahinter den dorthin zu schickenden Wert. Damit erspart man sich mehrere „sends“:



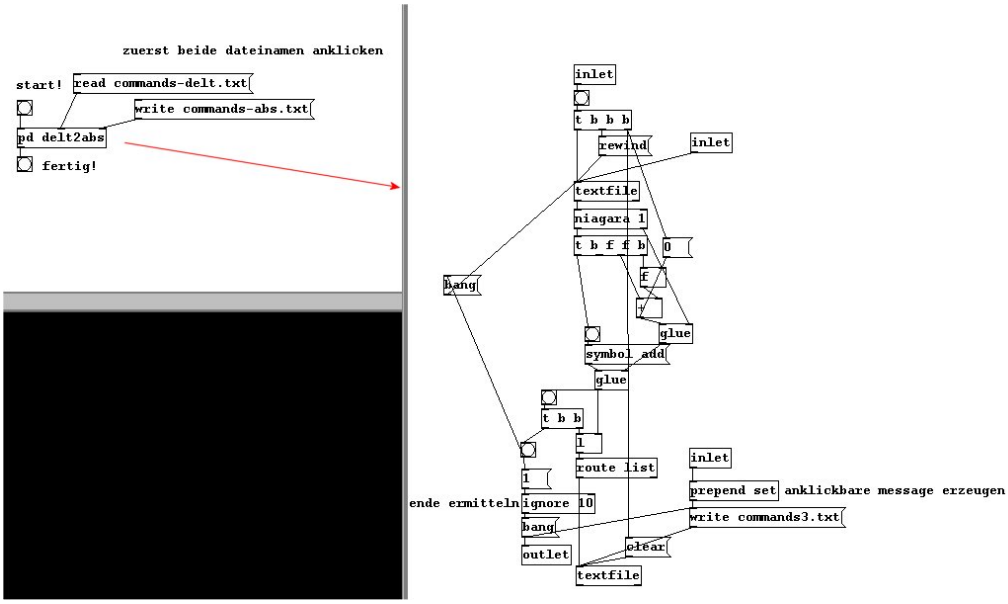
Damit bauen wir uns nun unsere eigene qlist mit absoluten Werten:

patches/4-2-3-1-klist.pd



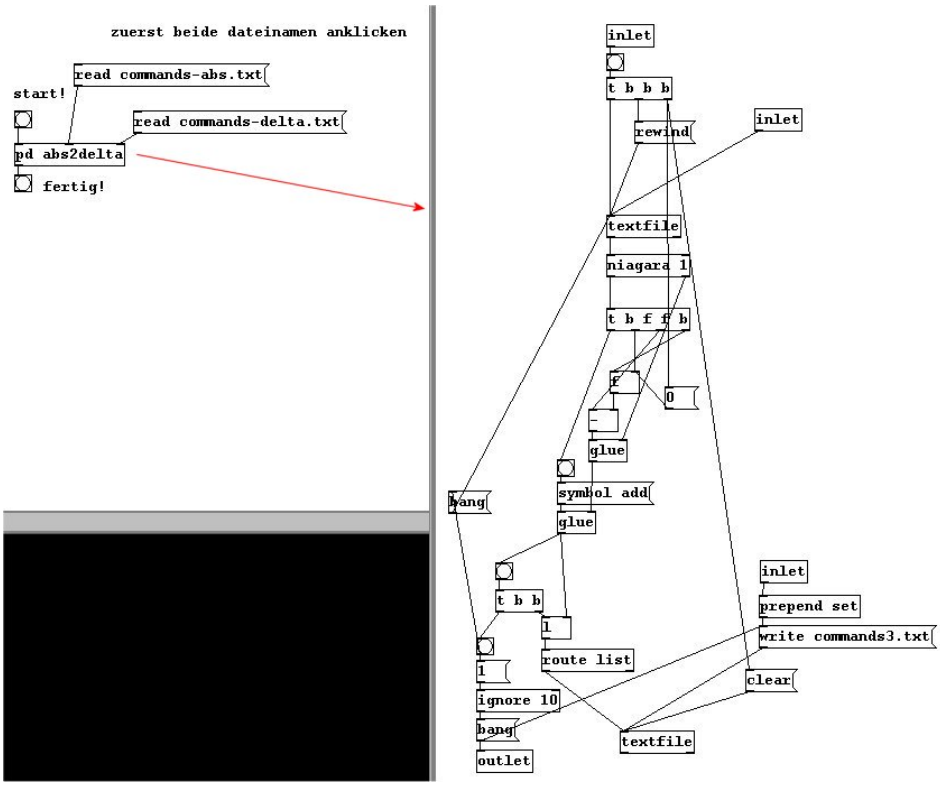
Und so wandeln wir eine Liste mit Delta-Werten in eine mit absoluten Wert um:

patches/4-2-3-1-klist-konvertierung1.pd



Und umgekehrt:

patches/4-2-3-1-klist-konvertierung2.pd



4.2.4 Für besonders Interessierte

4.2.4.1 Listen extern erstellen: Lisp

Mit „textfile“ können wir Textdateien erstellen, zum Beispiel vorher berechnete Algorithmen speichern. Hierzu gibt es aber auch spezielle Programmiersprachen. An dieser Stelle sei auf LISP hingewiesen, eine Programmiersprache die für die Erstellung und Bearbeitung von Listen besonders gut geeignet ist:

http://de.wikipedia.org/wiki/LISP

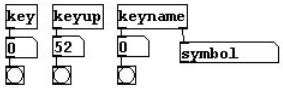
4.3 HIDs

4.3.1 Theorie

So wie man ein Instrument spielen kann, möchte man häufig auch einen Patch live spielen. Während etwas im Patch abläuft, können wir ja auf die GUI-Objekte klicken; die Verwendung einer Maus ist allerdings unpraktisch, wenn man dabei exakt in der Zeit arbeiten will. Hierfür gibt es HIDs, Human Interface Devices, das heißt Schnittstellengeräte zwischen Mensch und Maschine, zu denen Tastatur und Maus gehören, aber auch etliche weitere, zum Beispiel speziell für Musik Gemachte. Mit ihnen lässt sich in Pd ebenfalls ein Patch steuern.

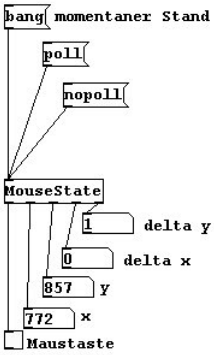
4.3.1.1 Tastatur und Maus

Klicken wir in eine Nummern-Box, können wir per Tastatur einen Wert eingeben. Wir können aber auch direkte Informationen über die Tastatur erhalten:



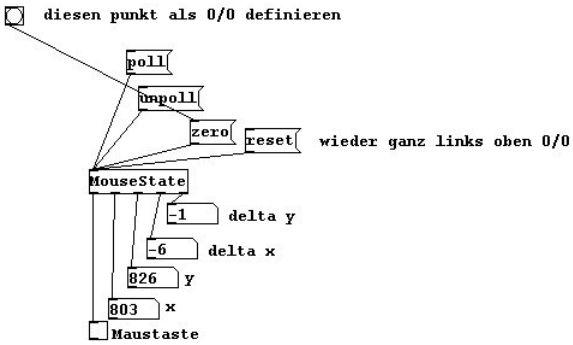
Die Tasten sind durchnummeriert (allerdings werden nicht alle Tasten einer Tastatur richtig erfasst, zum Beispiel die F1- bis F12-Tasten nicht). „key“ registriert das Hinunterdrücken einer Taste, „keyup“ das Loslassen. Es wird dann jeweils die Nummer der Taste ausgegeben. „keyname“ zeigt den normalen Namen einer Taste an.

Mit dem „MouseState“-External (Groß-Kleinschreibung beachten!) aus Pd-extended können wir auch die Daten der Maus nutzen:



Mit „poll“ und „unpoll“ starten / stoppen wir die Anzeige (in meiner Version muss man zuerst auf „unpoll“ und dann auf „poll“ klicken, damit es funktioniert). Es werden absolute x/y-Koordinaten und Delta-Werte angegeben, sowie, ob die linke Maustaste gedrückt wird.

Normalerweise sind die Koordinaten 0/0 ganz links oben auf dem Monitor; mit „zero“ können wir aber auch einen anderen Punkt als Referenz einrichten:



4.3.1.2 MIDI

Anfang der 80er Jahre beschlossen die großen Hersteller von elektronischen Musikinstrumenten einen Standard bzw. ein Datenübertragungs-Protokoll für verschiedene Eingabegeräte, genannt Musical Instrument Digital Interface. Und so gibt es MIDI-Keyboards, MIDI-Mischpulte, MIDI-Handschuhe

etc. In Pd gibt es Objekte zum Empfang wie zum Senden von MIDI-Daten. Man kann sie an ein Instrument senden, das mit diesen Daten wiederum Klang abspielen kann. Jedoch sind auch in den meisten Computer-Soundkarten MIDI-Klänge implementiert, die mit diesen Befehlen abgespielt werden können.

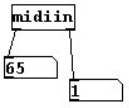
Das MIDI-Protokoll stellt keine Klänge dar, sondern besteht aus Befehlen zur Ansteuerung des Patches oder anderer Instrumente. Dazu werden Befehle übermittelt, wie z. B. „Note-on“ („Schalte Ton an“), „Velocity“ („Anschlagsstärke“) und „Note-off“ („Schalte Ton aus“). Neben diesen elementaren Befehlen stellt MIDI weitere, teilweise sehr spezielle Befehle zur Verfügung, die beispielsweise dazu verwendet werden, andere Klänge zu laden oder geladene Klänge mittels Steuerdaten, wie sie von Schaltern, Knöpfen oder Drehreglern erzeugt werden können, zu beeinflussen.

Jeder normierte MIDI-Befehl (mit Ausnahme systemexklusiver Daten, kurz SysEx genannt) trägt neben seiner Befehlskennung und den Befehlsdaten auch eine Kanalnummer. Die Kanalnummer ist 4 Bits groß, es lassen sich dadurch 2^4, also 16 Kanäle ansteuern. Je nach Software sind die Kanäle 0 bis 15 oder 1 bis 16 durchnummeriert, wobei die Nummerierung von 1 bis 16 üblicher ist.

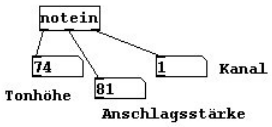
Da MIDI ein serielles Protokoll und die Datenrate der MIDI-Schnittstellen für heutige Verhältnisse recht gering ist, ergeben sich beim Abspielen vieler Noten häufig Timing-Probleme, vor allem beim Einsatz von Sequenzer-Programmen. Schon das Anschlagen eines Akkords mit mehreren Noten kann zu hörbaren Verzögerungen führen, denn MIDI kann die Noten nie zeitgleich durch die Leitung schicken, sondern nur nacheinander.

Für die folgenden Beispiele ist MIDI-Hardware erforderlich. Wir stellen in Pd diese Geräte unter **Media # MIDIsettings** ... ein.

Am grundlegendsten ist „midiin“. Jede MIDI-Eingabe wird darin angezeigt, links der Wert und rechts die Kanal-Nummer.

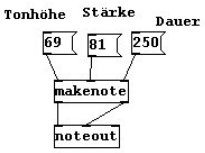


Haben wir ein MIDI-Keyboard oder ein Eingabegerät mit bestimmten Tonhöhen, erhalten wir mit „notein“ folgende Werte:

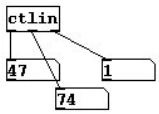


Links erscheint die MIDI-Nummer der Tonhöhe, in der Mitte die Stärke des Anschlags, rechts die Kanal-Nummer.

Umgekehrt können wir diese Informationen an ein Instrument schicken; haben wir nur ein Gerät, müssen wir keine Kanal-Nummer angeben. Wir verwenden „noteout“, und dazu „makenote“, das unsere Angaben ähnlich wie „pack“ zusammenfasst:



Ebenso gibt es die Control-Werte, mit „ctlin“ und „ctlout“. Betrachten wir „ctlin“:

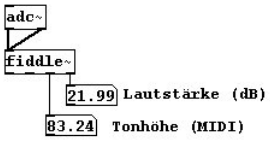


Der linke Output ist der Wert, der mittlere die Nummer des Controllers, der rechte der Kanal. Mit einem Argument können wir den mittleren Wert angeben und so gleich einen bestimmen Controller anwählen.

Alle weiteren MIDI-Sender und -Receiver funktionieren ähnlich. Zu ihnen zählen „pgmin“, „bendin“, „touchin“, „sysexin“.

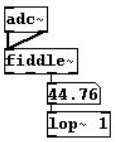
4.3.1.3 Steuerung mit Signalen

Klangeinaben über das Mikrofon können wir nicht nur als Klang, sondern auch als Steuerungsdaten verwenden. Wie schon unter 3.8.3.1 kennengelernt, erhalten wir mit „fiddle~“ aus dem eingehenden Klang die Informationen Amplitude und Frequenzen der Teiltöne:



Diese Zahlen – Pure Data arbeitet pur mit Daten – können wir nun wiederum auf Parameter eines Patches anwenden.

Ein Problem hierbei ist, dass die Informationen aus „fiddle~“ recht chaotisch sind. Dafür gibt es noch einen Trick zur Filterung:



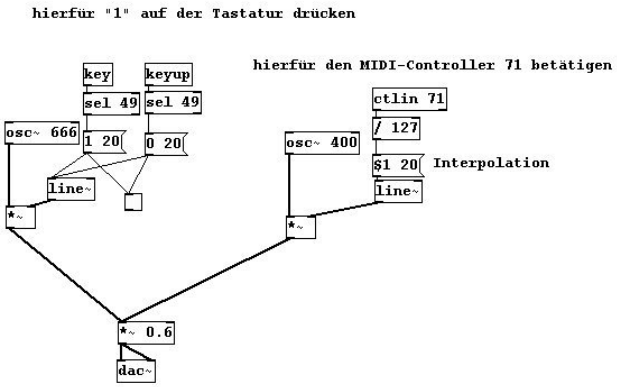
Der Lowpassfilter kann auch einen Kontrolldateninput erhalten. Und in diesem Fall werden nur relativ langsame Veränderungen durchgelassen.

4.3.2 Anwendungen

4.3.2.1 Patches live spielen

Mit Hilfe der oben beschriebenen Eingabegeräte (Eingabemethoden) können die Paramter eines Patches nun extern verändert werden:

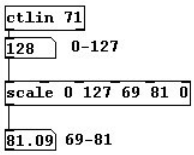
patches/4-3-2-1-patch-play.pd



Es liegt auf der Hand, dass die verschiedenen Geräte bestimmte Funktionen haben: Eine Taste steht für Ein/Aus, ein Drehregler für allmähliche Änderungen.

Für Controller mit einer Zahlreihe, wie Drehregler oder Schieber, empfiehlt sich noch eine Interpolation (da sie bei MIDI zum Beispiel nur 128 Werte haben).

In Pd-extended (allerdings nur, wenn GEM nicht geladen wurde) befindet sich ein sehr nützliches External das in diesem Fall eingesetzt werden kann: „scale“. Wenn wir zum Beispiel Frequenzen zwischen 69 und 81 (MIDI-Nummern) verändern und hierzu einen MIDI-Regler gebrauchen wollen, der aber Zahlen von 0 bis 127 erzeugt, können wir schreiben:



Das letzte Argument steht für linear (0) oder exponentiell (1).

4.3.2.2 Weitere Aufgabenstellungen

a) Verwenden Sie für die Algorithmen unter 4.1.2.1 statt Oszillatoren externe MIDI-Klänge.

b) Bedienen Sie Parameter von allen möglichen Patches aus Kapitel 3 mit Eingabegeräten.

4.3.3 Appendix

4.3.3.1 Andere HIDs

Neben der normalen Tastatur und Maus und MIDI-Geräten wächst die Zahl anderer Eingabegeräte noch beständig. So werden für Computerspiele Joysticks und ähnliches gebraucht, für Zeichnungen Tablets bis hin zu Bewegungssensoren. Gegenwärtig (Juni 2008) gibt es noch kein allgemeines Objekt in Pd, das die Informationen dieser Geräte wiedergibt. Es können nur folgende Externals genannt werden:

„joystick“, das Informationen von Joysticks empfangen kann; „wintablet“ für Wacom Tablets unter Windows; „hid“ unter Linux und MacOSX, das viele Eingabegeräte registriert. Seit einiger Zeit gibt es auch das Arduino-Board, an das man analoge Geräte anschließen kann, deren Informationen es digitalisiert. Mit zusätzlicher Software können diese Informationen dann auch in Pd empfangen werden.

4.3.3.2 Video-Input

Es gibt für Pd die Video-Erweiterung GEM, mit der auch aus eingehendem Video-Material – ob von einer vorab aufgenommenen Datei oder live von einer Webcam – Zahlen gewonnen werden können, die sich dann natürlich wiederum als Parameter von Klängen nutzen lassen.

4.3.4 Für besonders Interessierte

4.3.4.1 Instrument Design

Mit der Kunst des kompositorischen Gebrauchs externer Eingabegeräte habe ich mich theoretisch in einem Vortrag bei den Lagerhaus Lectures Freiburg befasst, auf den ich an dieser Stelle hinweise:

http://www.kreidler-net.de/theorie/instrument-design.htm

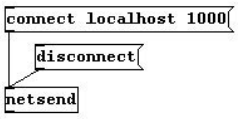
4.4 Netzwerk

Im Folgenden geht es um die Verbindung mehrerer Computer untereinander, die nicht nur die direkte Klangsteuerung zum Ziel hat, sondern auch die Kommunikation mit anderen Menschen an anderen Rechnern, während man zusammen ein Stück spielt.

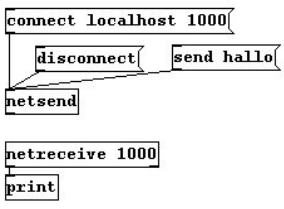
4.4.1 Netsend / Netreceive

Ein Pd-Patch kann mit einem Pd-Patch eines anderen Computers Daten austauschen.

Zunächst müssen wir via Netzwerkkabel mit einem anderen Rechner verbunden sein, auf dem ebenfalls Pd läuft. Mit „netsend“ verbinden wir uns mit einem anderen Rechner. Wir geben als Message „connect [name] [Port-Nummer]“; statt des Namens geht auch die I.P.-Adresse des anderen Rechners. Mit „disconnect“ beenden wir eine Verbindung.



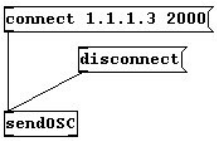
Sind wir verbunden, können wir mit „send“ und angehängten Symbolen Nachrichten an den anderen Rechner schicken. Der andere Rechner empfängt die Daten mit „netreceive [Port-Nummer]“.



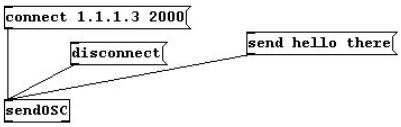
4.4.2 OSC

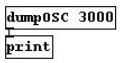
In Pd-extended stehen auch die OSC-Objekte zur Verfügung, mit denen mit vielen anderen Computerprogrammen in einem Netzwerk Daten ausgetauscht werden können. OSC steht für *open sound control*. Ihre Funktionsweise ist fast identisch mit „netsend“ und „netreceive“.

Wir müssen via Netzwerkkabel mit einem anderen Rechner verbunden sein, auf dem ebenfalls OSC läuft. Mit „sendOSC“ (Groß-/Kleinschreibung beachten!) verbinden wir uns mit einem anderen Rechner. Wir geben als Message „connect“, dazu die I.P.-Adresse des anderen Rechners sowie die Portnummer. Mit „disconnect“ beenden wir eine Verbindung.



Sind wir verbunden, können wir mit „send“ und angehängten Symbolen Nachrichten an den anderen Rechner schicken, sofern dort auch ein Programm mit OSC läuft. Der andere Rechner empfängt die Daten mit „dumpOSC“ und als Argument den Port des Senders.





Kapitel 5. Sonstiges

5.1 Arbeit erleichtern

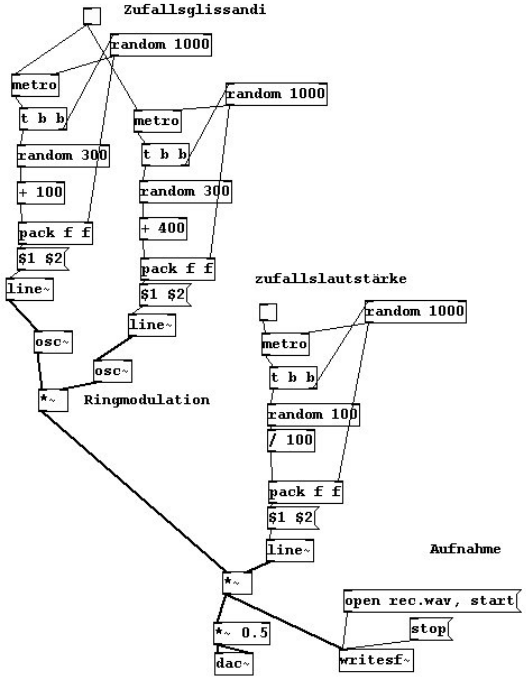
5.1.1 Theorie

5.1.1.1 Unterpatches

Wie man Subpatches in Pd anlegt, haben wir bereits unter 2.2.4.4 gelernt. Jetzt soll nur noch einmal verdeutlicht werden, wie sie sinnvoll zu gebrauchen sind.

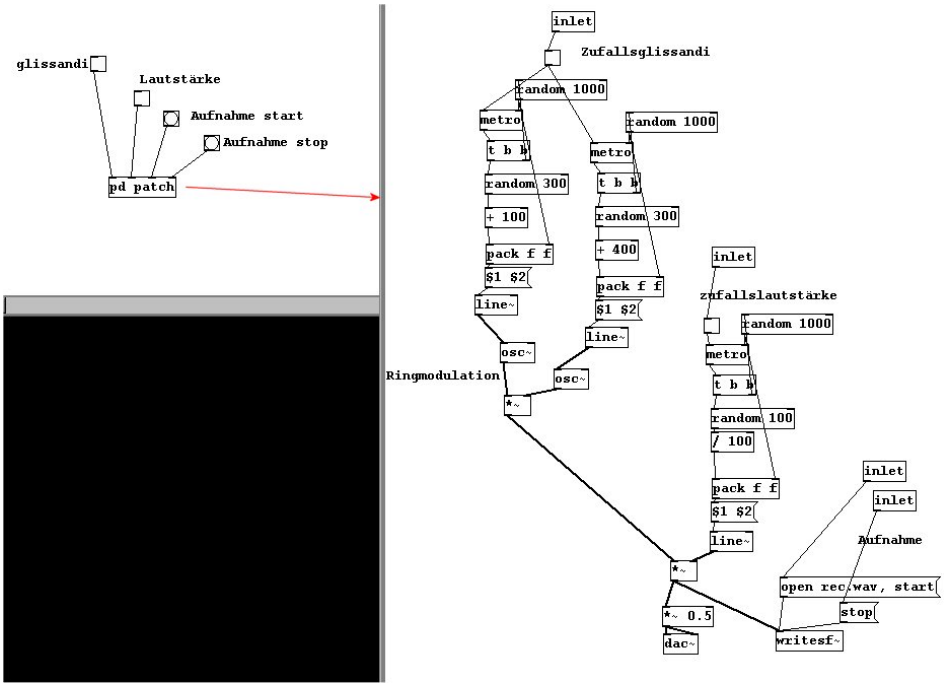
Haben wir einen Patch wie diesen ...

patches/5-1-1-1-subpatch1.pd



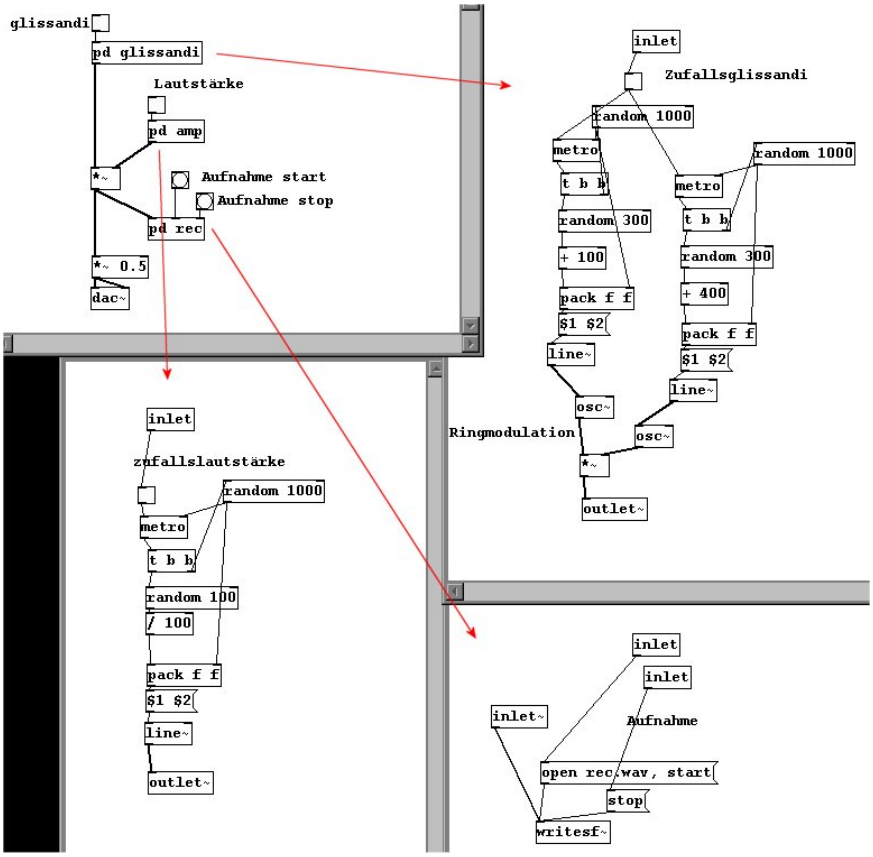
... können für Ordnung sorgen und alles, worauf wir nicht unmittelbar zugreifen, in einen Subpatch auslagern:

patches/5-1-1-1-subpatch2.pd



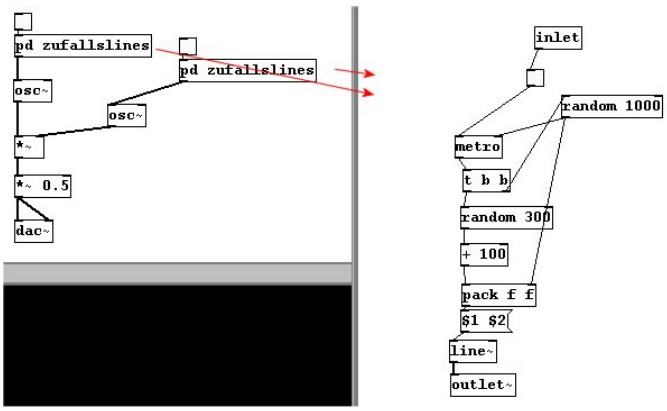
Sinnvoll ist aber auch, je nach Inhalt mehrere Subpatches anzulegen, wenn man später eine bestimmte Sache bearbeiten will.

patches/5-1-1-1-subpatch3.pd



So können wir uns etwa für einen Patch einen bestimmten Algorithmus bauen, der dann an verschiedenen Orten verwendet wird:

patches/5-1-1-1-modul.pd

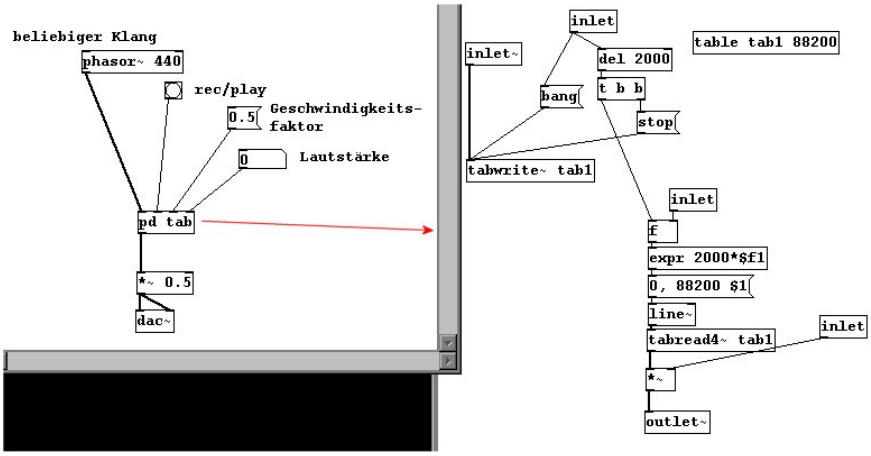


Man spricht in diesem Fall auch von einem „Modul“.

5.1.1.2 Abstraktionen

Haben wir nun einen universell anwendbaren Subpatch, zum Beispiel den folgenden, der zwei Sekunden in einen Array aufnimmt und dann mit einer bestimmten Geschwindigkeit und variablen Lautstärke abspielt:

patches/5-1-1-2-abstraktion1.pd

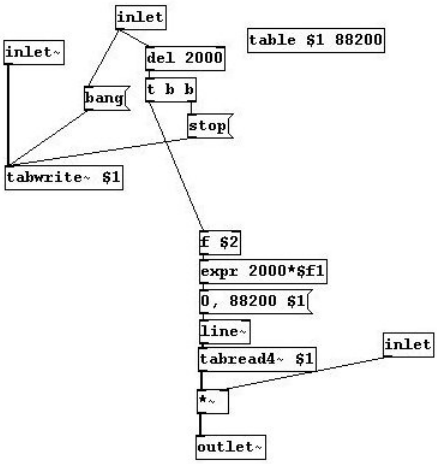


... und möchten ihn an verschiedenen Orten verwenden, haben wir das Problem, dass nach jedem Duplizieren erst einmal der Array darin umbenannt werden muss und ein neuer Input (für die Geschwindigkeit) gegeben werden muss. Stattdessen können wir aber auch eine Abstraktion daraus machen; dies ist ein Patch, der als separate Datei abgespeichert wird und sich dann vielfach mit Variablen aufrufen lässt.

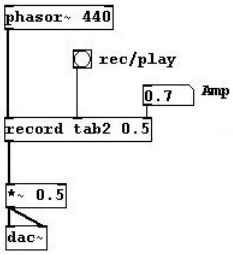
Wir nehmen den Subpatch von eben und speichern ihn unter dem Namen „record.pd“ ab. Dann beginnen wir einen gänzlich neuen Patch, aber speichern diesen im selben Verzeichnis wie „record.pd“ unter dem Namen „main“. Anschließend schreiben wir als Objekt „record“:

„record“ ist eigentlich kein Objekt von Pd. Ist allerdings ein Patch mit diesem Namen (plus der Endung ".pd") in dem selben Verzeichnis vorhanden, wird ein Objekt erstellt, das diesen Patch wie ein wie ein Subpatch beinhaltet.

Der Vorteil dabei besteht in den Variablen. Laden wir wieder den „record.pd“-Patch (**File** # **Open**). Darin können wir in Objekte Variablen der Form „$1“, „$2“ etc. schreiben. Setzen wir zum Beispiel Variablen für den Arraynamen und für die Geschwindigkeit:

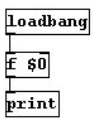


Nun können wir im „main.pd“-patch wieder unser neues Objekt „record“ schreiben, und zwar mit zwei Argumenten für die beiden Variablen, also als erstes Argument einen Namen für den Array und als zweites die Abspielgeschwindigkeit.

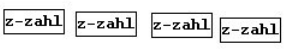


Die Lautstärke, die wir weiterhin im Mainpatch ändern wollen, setzen wir genau wie bei einem Subpatch als Inlet.

Nun bleibt aber noch das Problem, dass wir, wenn wir in dem Mainpatch mehrfach die Abstraktion „record“ laden, jedes Mal dem Array darin einen eigenen Namen geben müssen. Es gibt dafür eine spezielle Option in Pd, für einen bestimmten Patch eine individuelle Zufallszahl quasi als Namen zu generieren. Diese wird mit $0 erzeugt. Wir machen die Abstraktion „z-zahl.pd“:

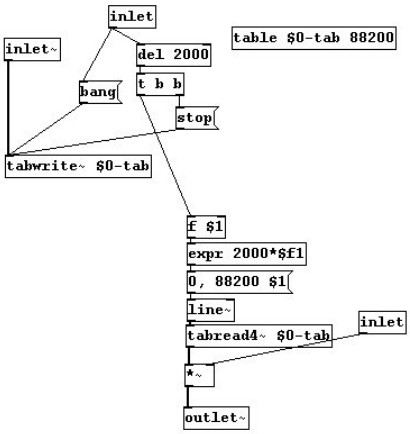


Es wird eine Zufallszahl mit Offset 1000 erzeugt, von der aus dann hochgezählt wird. Jedes Mal wenn wir nun die Abstraktion aufrufen, wird eine andere Zahl erzeugt:

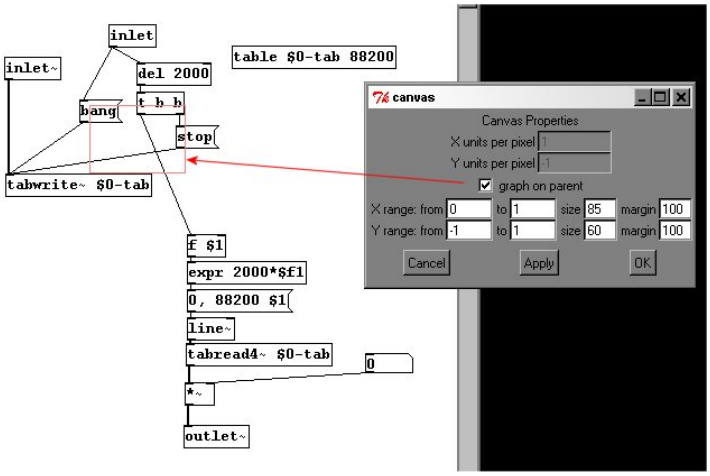



```
print: 1021
print: 1022
print: 1023
print: 1024
```

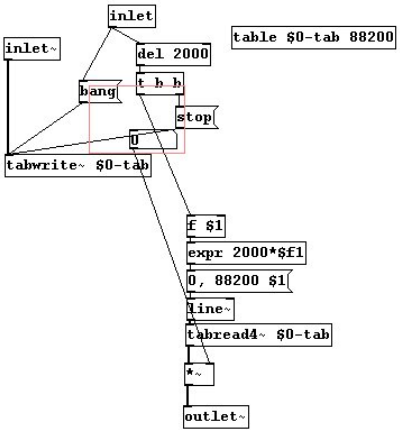
Damit geben wir nun dem Array einen eigenen Namen. Konventionell geschieht das in der Form: $0-[Arrayname]. Ansonsten brauchen wir nur noch eine Variable ($1) für die Geschwindigkeit. Die Lautstärke soll weiterhin variabel bleiben.



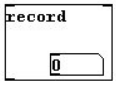
Zuletzt können wir noch grafische Elemente einer Abstraktion mit der Funktion "Graph on Parent" in das spätere Objekt holen. In dem Patch „record“ erstellen wir für die Lautstärke statt des Inlets eine Nummern-Box, dann klicken wir mit der rechten Maustaste irgendwo auf die weiße Fläche, # **Properties**. Dort haken wir „graph on parent“ und dann „apply“ an:



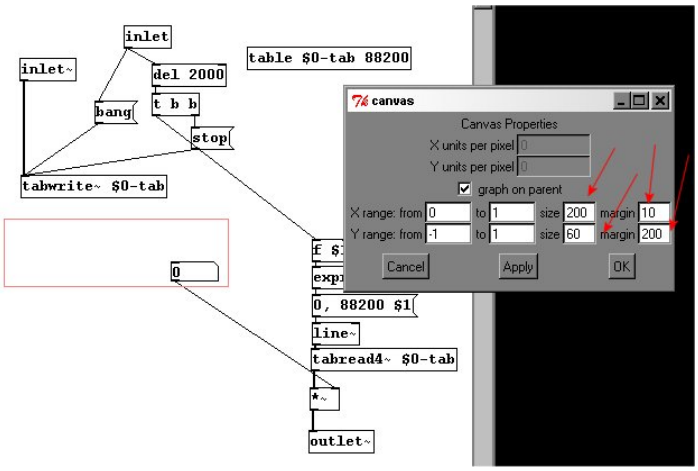
Wir sehen darauf ein rotes Rechteck. Alle GUI-Objekte, die vollständig innerhalb dieses roten Feldes sind, werden dann in dem späteren Objekt angezeigt. Wir bewegen die Nummern-Box für die Lautstärke in die Box, speichern „record.pd“ und schreiben anschließend das Objekt in einem neuen Patch:



Im neuen Patch:

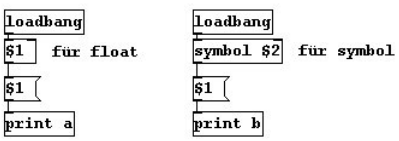


Wir können die Größe und Position dieses roten Feldes hier ändern:



Graph on Parent funktioniert auch mit Subpatches.

Zweierlei gilt es zu beachten: Man kann den Inhalt einer Abstraktion vom Mainpatch aus sehen und verändern (einfach im Mainpatch auf das Objekt klicken). Die Veränderung darin wird aber nicht gespeichert! Nur wenn man die Datei der Abstraktion selbst als normalen Patch in Pd öffnet, kann man ihn auch ändern und speichern. Außerdem können in einer Abstraktion die $-Variablen nur in Objekten geschrieben werden. In Message-Boxen bezeichnet $1, wie sonst auch, nur einen Input in die Message-Box (siehe 2.2.2.1.4). Wir helfen uns beispielsweise so:



Und wir müssen natürlich, wenn wir den Mainpatch kopieren und irgendwo andernorts (in einem anderen Verzeichnis, auf einem anderen Computer) verwenden möchten, auch den „record“-patch mit

kopieren. Wir können Abstraktionn aber auch immer verfügbar machen, wenn wir sie im Verzeichnis „extra“ innerhalb des Pd-Verzeichnisses speichern. Alle darin liegenden Patches sind immer als Abstraktionen verfügbar. Ebenso kann man aber auch ein eigenes Verzeichnis mit Abstraktionen anlegen, das standardmäßig geladen werden soll. Wir stellen einen solchen Verzeichnispfad in **File # Path** ein.

5.1.1.3 Pd erweitern

Viele nützliche Objekte, die nicht in der ursprünglichen Version von Pd enthalten sind, wurden mittlerweile von vielen Programmierern weltweit erstellt. Man nennt diese Objekte „Externals“. Mehrere Externals sind wiederum zu „Libraries“ (dt. Bibliotheken) zusammengefasst, wie zum Beispiel die zexy-library oder die maxlib-library. Man muss sie in den Ordner „Extra“ und überdies in den Ladevorgang integrieren (**File # Startup**). Mittlerweile gibt es auch „Pd extended“, das bereits weitere, zur ursprünglichen Version hinzugefügte Bibliotheken enthält.

Außerdem lässt sich Pd durch Zusatzprogramme ergänzen. Das bekannteste ist GEM, mit dem man, statt wie sonst mit Klang, mit Video-Daten und -Dateien arbeiten kann. Darüber hinaus lassen sich beispielsweise mit open sound control Daten aus anderen Programmen, die ebenfalls über einen OSC-Anschluss verfügen, in Pd integrieren. Die Pd-Versionen einiger Programmierer sind vor lauter Erweiterungen fast nicht mehr wiederzuerkennen.

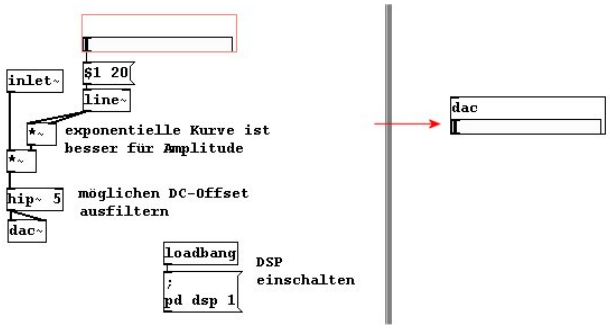
5.1.2 Anwendungen

5.1.2.1 Customize your Pd

Wie bereits im letzten Kapitel angedeutet, kann man Pd stark individualisieren. Erstellen wir uns einige nützliche Abstraktionen.

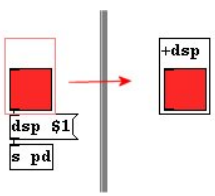
Für den normalen Programmiergebrauch etwa ein „dac~“ mit eingebautem Lautstärkeschieber, nun genannt „dac“:

patches/dac.pd



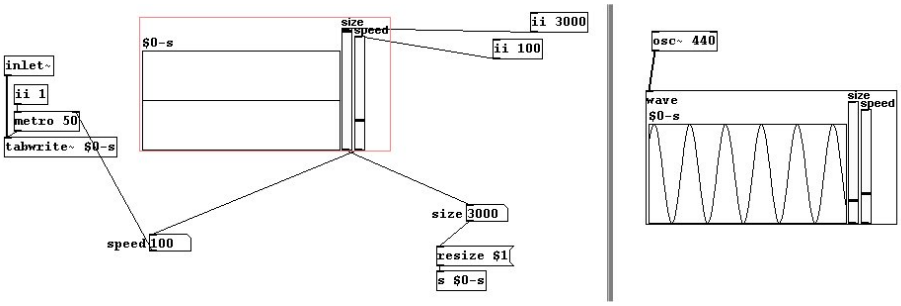
Einen DSP-Schalter, genannt „+dsp“:

patches/+dsp.pd



Eine Darstellung für die Wellenform:

patches/wave.pd



5.1.2.2 Weitere Aufgabenstellungen

Als Abstraktionen:

a) Bauen Sie den DSP-Schalter in die „dac“-Abstraktion ein, ebenso einen Schalter, der auf stumm schaltet und wieder zurück.

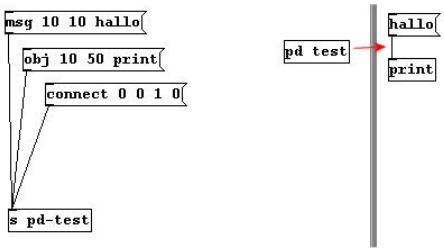
b) Bauen Sie einen Kompressor.

c) Bauen Sie eine Darstellung des eingehenden Spektrums.

5.1.3 Appendix

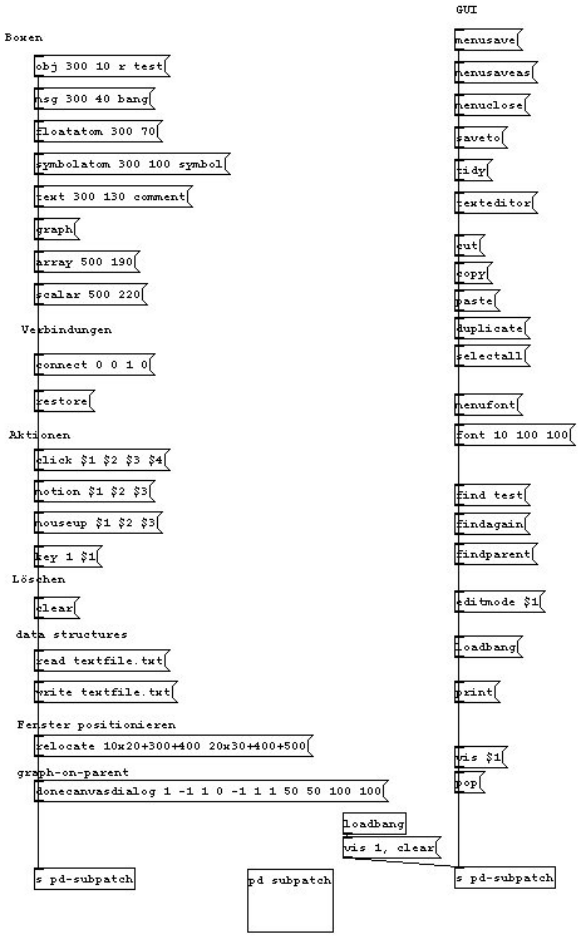
5.1.3.1 Ein Patch automatisch erstellen

Wir haben einen Patch mit einem Subpatch namens „test“. Im Hauptpatch senden wir diesem Subpatch durch „s pd-test“ nun folgende Messages, in der Reihenfolge von oben nach unten:



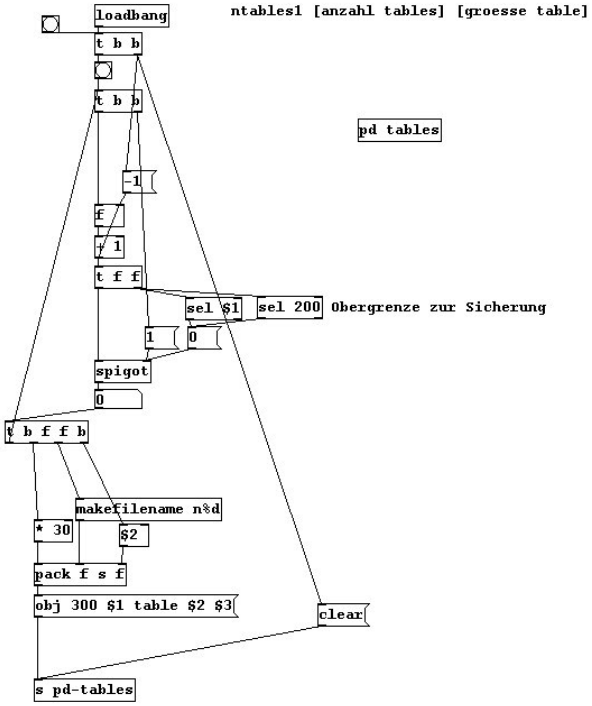
„obj“ erstellt ein Objekt; wir geben die x/y-Koordinaten sowie den Namen des Objekts mit Argumenten dahinter, ebenso die Message-Box. Die Verbindung funktioniert folgendermaßen: „connect [Objekt-Nummer] [Outlet-Nummer] [Objekt-Nummer] [Inlet-Nummer]“. Die Objekte werden in der Reihenfolge ihrer Erzeugung nummeriert, die Outlets und Inlets von links nach rechts. Alles jeweils bei 0 beginnen. Mit der Message „clear“ löschen wir den Inhalt des Subpatches.

Hier eine Übersicht über alle Befehle. Manche funktionieren gegenwärtig (Juni 2008) allerdings nicht:

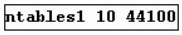


Nun können wir – wenn auch auf etwas komplizierte Weise – beispielsweise eine bestimmte Anzahl von Arrays mit einer Abstraktion erzeugen, was anders in Pd nicht zu realisieren ist (vgl. 3.2.3.1):

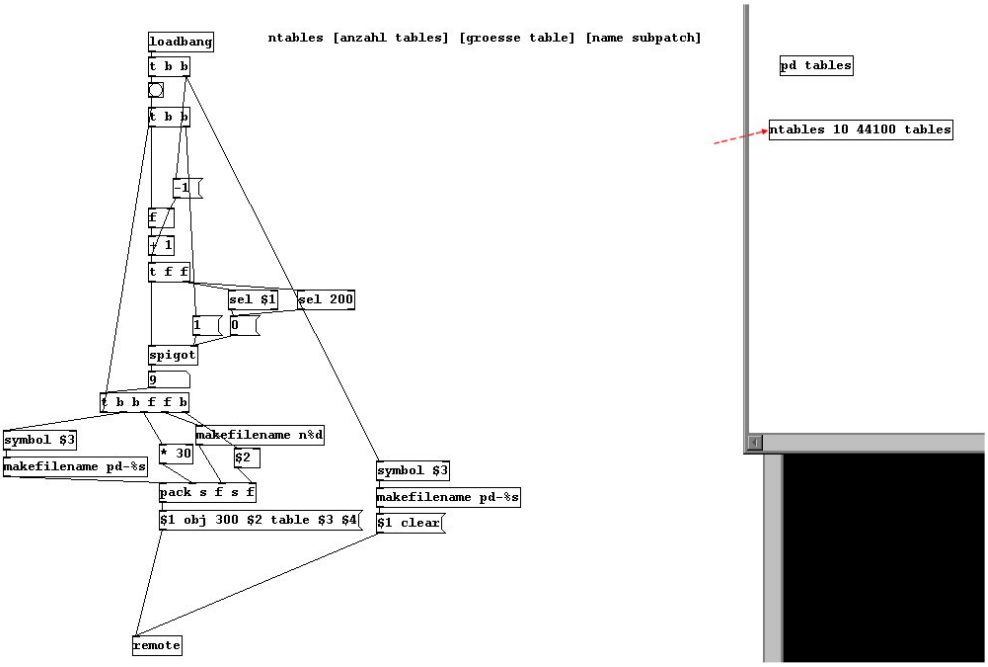
patches/ntables.pd



Mit dieser Abstraktion beispielsweise erzeugen wir in dem Subpatch zehn Arrays mit der Größe 44100:



Und auf diese Weise können wir den Subpatch in unserem Hauptpatch anlegen:



5.1.4 Für besonders Interessierte

5.1.4.1 Eigene Objekte schreiben

...kann man natürlich auch. Pd ist ja eigentlich nur eine Oberfläche für eine Programmierung in der Programmiersprache C. Mit dieser können eigene Objekte („Externals“) geschrieben werden. Hierzu gibt es folgende Anleitung: http://iem.at/pd/externals-HOWTO/

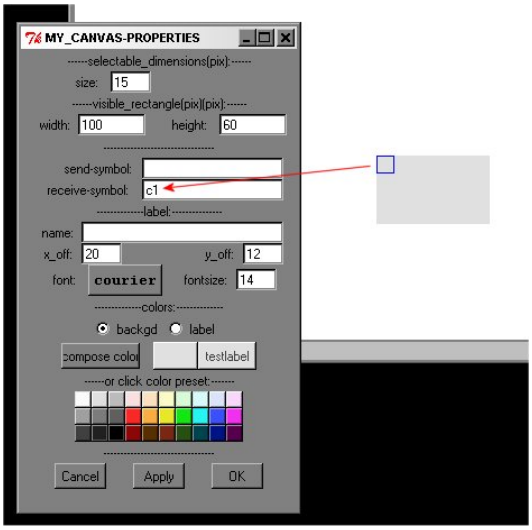
5.2 Visuelles

5.2.1 Theorie

5.2.1.1 Pd ist visuell und somit visuell programmierbar

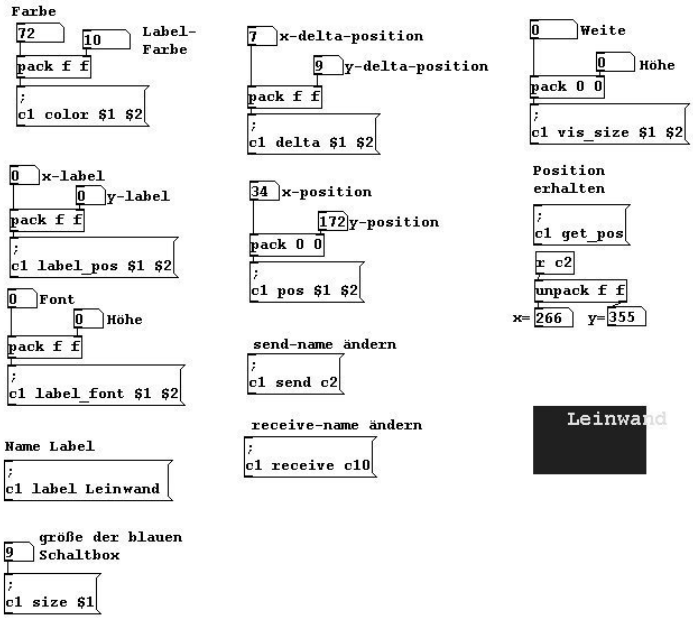
Das Handwerkszeug zur visuellen Darstellung haben wir schon unter 2.2.4.3 gelernt. Hinzu kommt jetzt noch die Steuerung einer Canvas.

Erstellen wir eine Canvas (**Put** # **canvas**) und geben ihr in den Properties (Rechtsklick auf die Box links oben in der Canvas) das receive-Symbol „c1“:



Nun können wir der Canvas Nachrichten senden:

patches/5-2-1-1-canvas.pd



5.2.2 Anwendungen

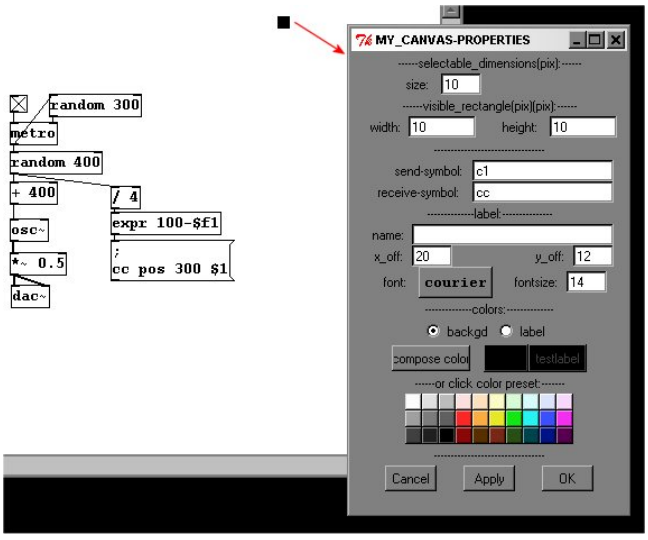
5.2.2.1 Schaltfläche und Unterpatches

Dies sollte aus dem Vorhergegangenen schon klar geworden sein: Ein sauberer Patch besteht aus Schaltfläche und Unterpatches (vgl. 3.4.2.4 und 5.1.1.1). In diesem Tutorial habe ich meist darauf verzichtet, da Sie das jeweils Erläuterte auf einer Grafik besser überblicken können.

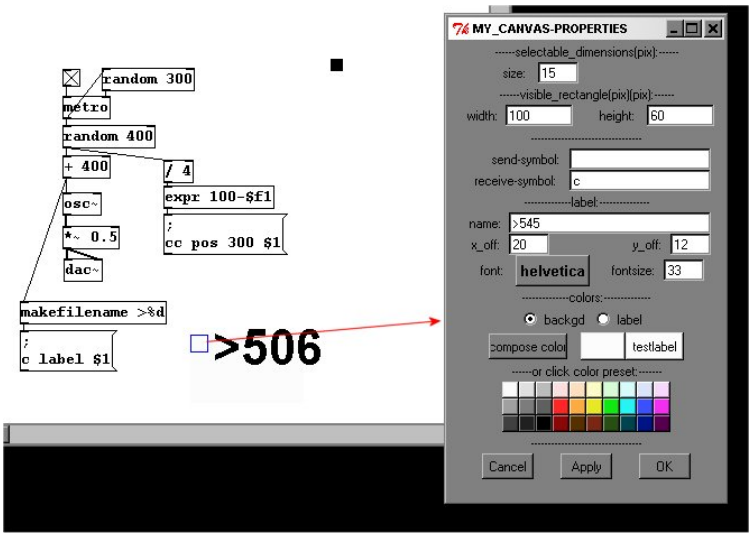
5.2.2.2 Canvases als Anzeige

Canvases dienen als farbliche Elemente, um einen Patch klarer zu unterteilen (siehe 3.4.2.4). Zur Anzeige von Funktionen lassen sie sich aber auch variabel einsetzen. Zum Beispiel können wir uns die Ergebnisse eines Zufallsgenerators anzeigen lassen:

patches/5-2-2-2-canvas-anzeige.pd



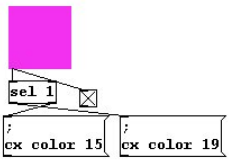
Wir können auch eine Zahl vergrößert darstellen (allerdings muss die Zahl grundsätzlich durch ein Symbol ergänzt werden, in diesem Beispiel „>“):



Die Möglichkeiten lassen sich fast beliebig erweitern, bis hin zur Erstellung ganzer grafischer Partituren mit Canvases.

5.2.2.3 Canvases als erweitertes GUI

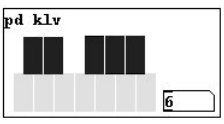
Canvases können über andere GUI-Objekte gelegt werden; die Reihenfolge der Erzeugung ist dabei entscheidend. So kann man eigene Toggles erstellen:



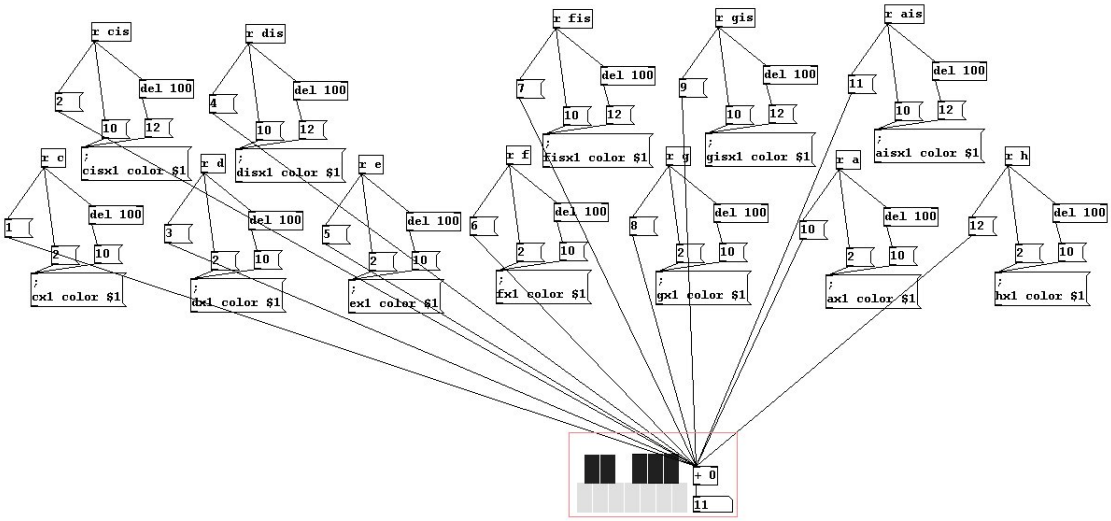
Es wurde zuerst ein Toggle mit der Größe 55 erstellt, von dem die Verbindung zum „sel 1“ weggeht. Dann wurde ein Canvas der Größe 55 und ein receive-Symbol „cx“ erstellt und genau über den Toggle gesetzt (wichtig ist die Reihenfolge des Erstellens, sonst verschwindet die Canvas hinter dem Toggle). Das „sel 1“ und den Rest könnte man auch in einem Subpatch separieren und von dem Toggle intern ein „send“ einstellen.

Hier sind wiederum den Möglichkeiten keine Grenzen gesetzt. So lässt sich zum Beispiel eine Klavieroktave umsetzen (selbstverständlich gibt es begnadetere Pd-Grafikkünstler als mich):

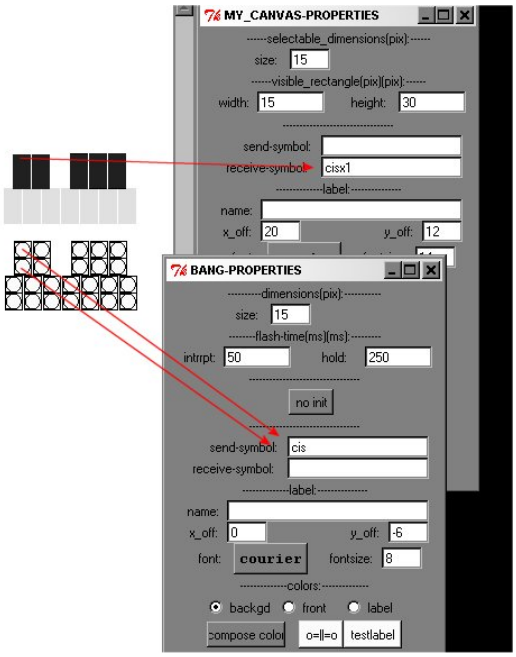
patches/5-2-2-3-klavier-anzeige.pd



Der Subpatch enthält:



Hinter dem Klavier stecken eine Reihe von Bangs:



5.2.2.4 Weitere Aufgabenstellungen

a) Stellen Sie eine Stoppuhr schön dar.

b) Stellen Sie ein laufendes 5/8-Metrum schön dar, also einen optischen Klicktrack.

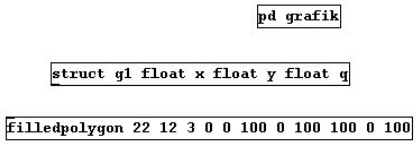
5.2.3 Appendix

5.2.3.1 Data Structures

Eine eigene Abteilung für Grafik in Pd sind die Data Structures.

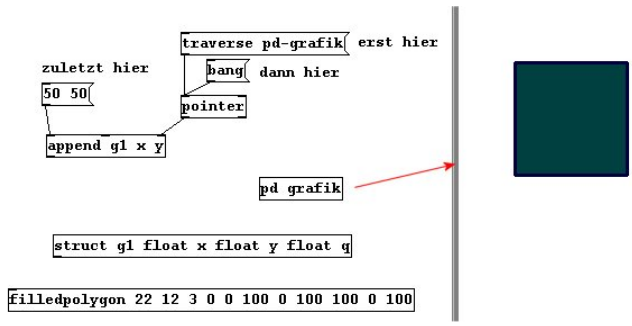
In einem Subpatch können wir grafische Elemente platzieren. Dazu erstellen wir zunächst den Subpatch „grafik“ und definieren für diesen Subpatch Variablen und eine Grafik. Wir nennen dies eine „Vorlage“. Sie erhält die Variablen mit „struct“; als Argument geben wir den Namen der Vorlage (hier „g1“) und dann immer jeweils Paare aus Typ und Name, in unserem Fall float x float y float q. „float“ ist der Typ (also eine Komma-Zahl), x, y, und q sind frei gewählte Namen.

Die Grafik können wir mit den vorhandenen Objekten „drawcurve“, „drawpolygon“, „filledcurve“ und „filledpolygon“ definieren. Nehmen wir für unser Beispiel „filledpolygon“. Ein Polygon ist ein Mehreck. Als erstes Argument gilt die Farbe innen, dann die Farbe des Randes, die Breite des Randes und dann Paare von Koordinaten-Punkten, angefangen links oben und weiter im Uhrzeigersinn.



Zum Erstellen der Grafik brauchen wir nun „append“. Es erhält als erstes Argument den Vorlagennamen und dann mögliche Variablen – x und y sollten immer dabei sein – außerdem als Input rechts zunächst die Stelle im Subpatch, an der die Grafik platziert werden soll. Wir müssen uns vorstellen, dass die grafischen Elemente im Subpatch wie eine Liste nacheinander aufgereiht sind. „append“ muss zunächst den Platz in der Liste wissen, den ihm das Objekt „pointer“ benennt. „pointer“ geben wir die Message „traverse pd-grafik“; damit geht „pointer“ ganz an den Anfang der Liste. Mit der Message „bang“ wird dieser Listenplatz nun ausgegeben (an „append“). Daraufhin geben wir „append“ im linken Inlet Daten für die Variablen von „append“.

patches/5-2-3-1-data-structures1.pd



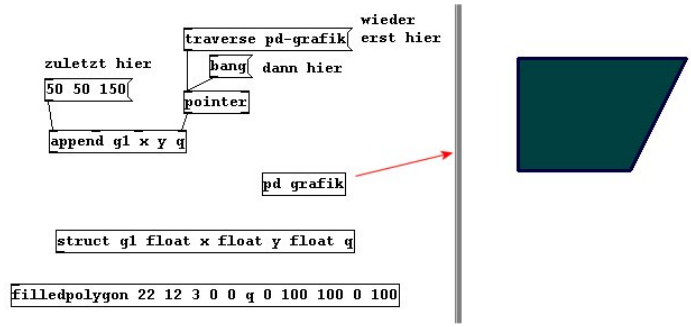
Um es noch einmal zu verdeutlichen: „x“ und „y“ sind spezielle Namen für ein grafisches Element bei den data structures. Damit wird die absolute Position festgelegt, 50/50. Wir haben nun ein „filledpolygon“ erzeugt mit der Innenfarbe 22, der Randfarbe 12, der Randbreite 3 und mit 4 Punkten: links oben, mit Abstand 0/0 von der absoluten Position, rechts oben mit Abstand 100/0 von der absoluten Position, rechts unten mit 100/100 und links unten mit 0/100. Würden wir die letzten zwei Zahlen einfach aus „filledpolygon“ entfernen, hätten wir nur noch ein Dreieck. Ebenso kann man eine Fläche mit beliebig vielen Ecken erzeugen.

Mit „traverse pd-grafik“ sind wir quasi ganz an den Anfang des Subpatches „grafik“ gegangen; mit „bang“ wurde dieser Platz an „append“ gegeben und append legt dann an dieser Stelle, wenn wir ihm Daten für die Variablen geben, die Grafik an.

Wir können nun aber auch an dieser Grafik etwas ändern. Wir haben ja für die Vorlage „g1“ noch die Variable „q“ (ein float) bestimmt. Geben wir nun für das filledpolygon ein: „filledpolygon 22 12 3 0

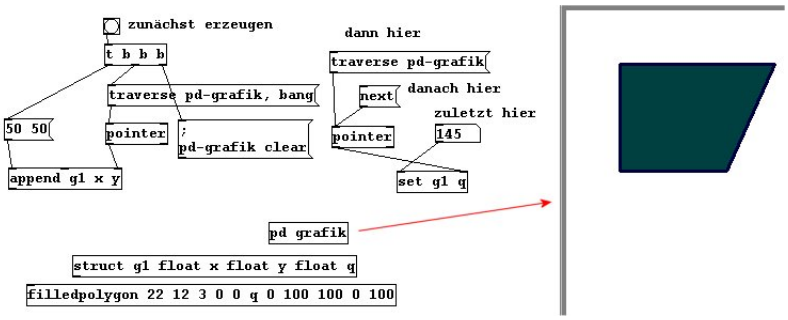
0 q 0 100 100 0 100“. Ein Punkt hat nun eine Variable. Daraufhin veschwindet der Punkt zunächst und wir haben nur noch ein Dreieck. Diesen variablen Punkt können wir mit „append“ einstellen:

patches/5-2-3-1-data-structures2.pd



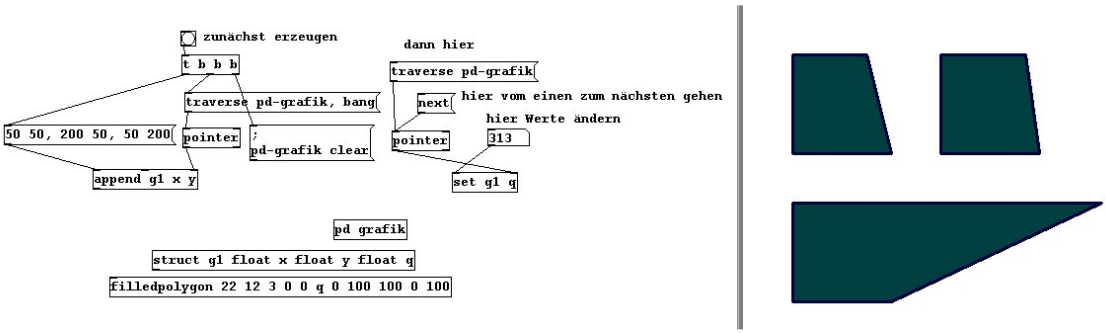
... aber auch mit „set“. Dies muss ebenso von dem pointer die Information erhalten, an welcher Stelle im Subpatch etwas geändert werden soll. In dem Fall ist es ein Schritt nach dem Beginn. Wir gehen mit „traverse pd-grafik“ ganz an den Anfang. Der Anfang ist unbesetzt; mit „next“ gehen wir an die sodann gesetzte Grafik. Nun können wir im linken Input für q einen Wert eingeben.

patches/5-2-3-1-data-structures3.pd



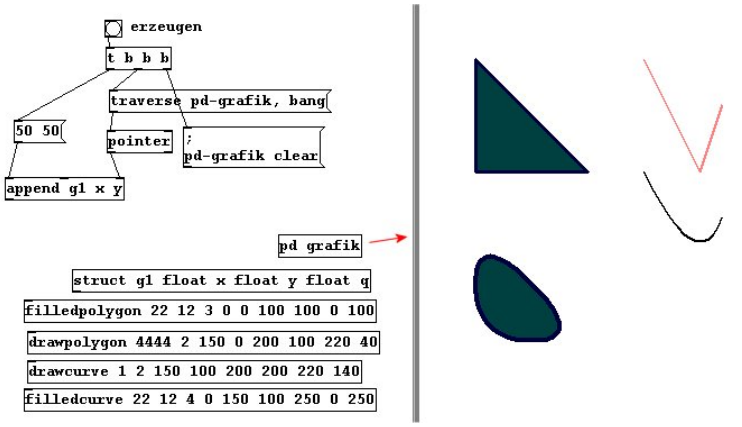
Mehrere Grafiken legen wir einfach mit mehreren Messages für „append“ an. Nun geht man mit „next“ von einer zur nächsten.

patches/5-2-3-1-data-structures4.pd



Zu den anderen grafischen Elementen: „drawpolygon“ ist einfach nur eine Linie, die mit Ecken verläuft; das erste Argument mit der Innenfarbe fällt weg. Ebenso verhält sich „drawcurve“, bei dem die Ecken des Strichs abgerundet sind. „filledcurve“ wiederum schließt die Fläche und hat als erstes Argument die Innenfarbe.

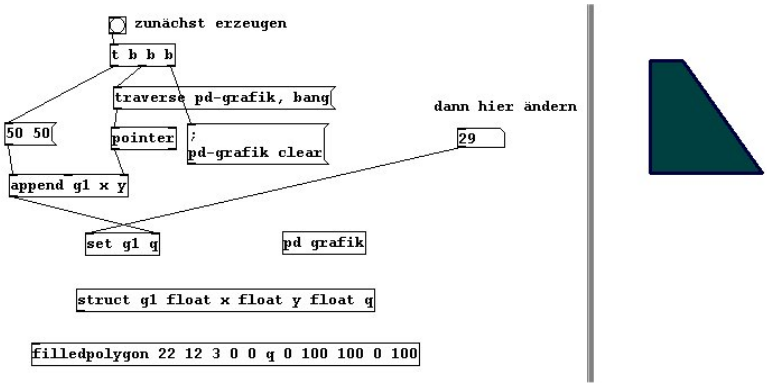
patches/5-2-3-1-data-structures5.pd



Dazu sei noch ergänzt: Die (Eck)Punkte einer Grafik, die Variablen haben, kann man mit der Maus verändern; der Cursor ändert sein Erscheinungsbild an der entsprechenden Stelle und der Punkt lässt sich bei gedrückter Maustaste bewegen.

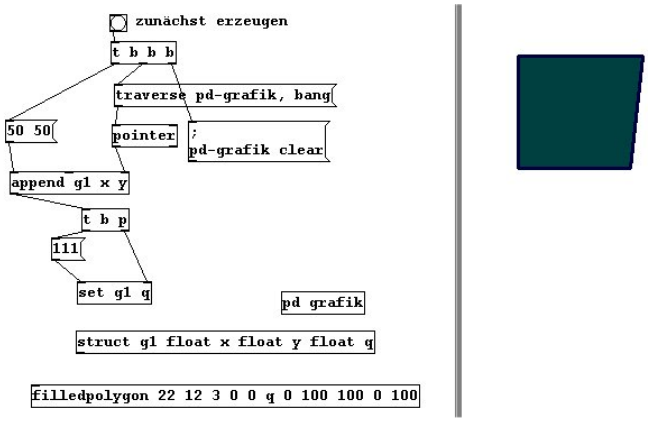
Setzt man mit „append“ ein grafisches Objekt, gibt es als Output einen neuen Pointer-Verweis genau auf dieses Objekt (wie „next“):

patches/5-2-3-1-data-structures6.pd



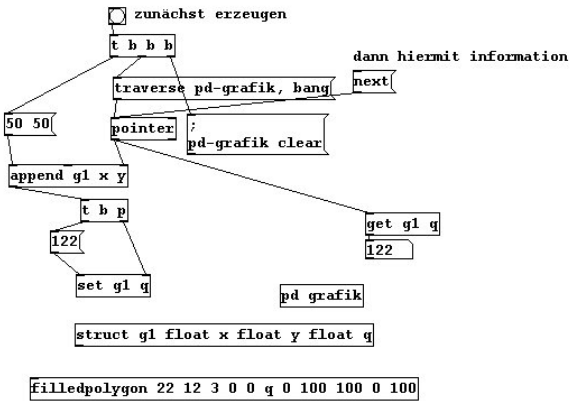
Ein pointer-Verweis ist übrigens eine eigene Art von Information, zum Beispiel für „trigger“:

patches/5-2-3-1-data-structures7.pd



Mit „get“ können wir nun auch die Daten von grafischen Elemente empfangen, angeschlossen an den Pointer:

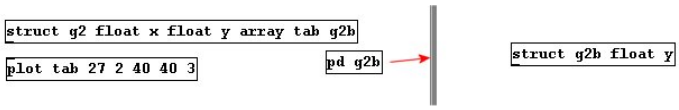
patches/5-2-3-1-data-structures8.pd



Zuletzt kann man auch einen (grafischen) Array mit data structures erzeugen. Der Array wird in „struct“ definiert, und zwar durch einen Namen und eine weitere zugewiesene Vorlage. Mit „plot“ geben wir diesem Array Farbe, Breite, Startpunkt (x/y) und Punkteabstand.

```
struct g2 float x float y array tab g2b
plot tab 27 2 40 40 3
```

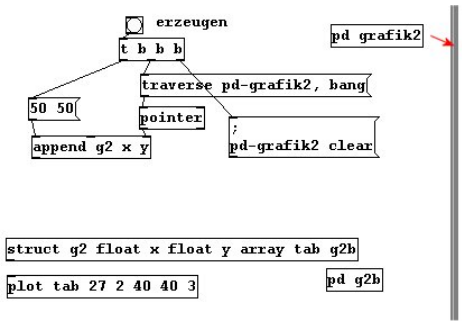
In einem anderen Subpatch muss dazu noch die weitere Vorlage, die die Variablen des Arrays bestimmt, enthalten sein:



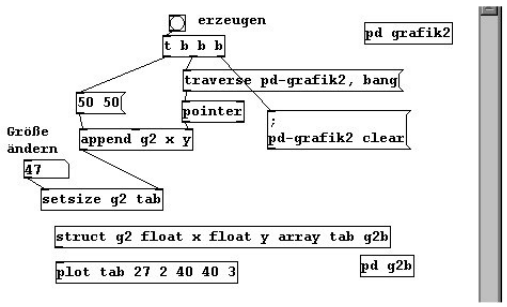
Die Variable „y“ wird automatisch als Höhe des Arrays verstanden. Diese Variable ist notwendig, um den Array richtig zu erstellen.

Nun erzeugen wir den Array:

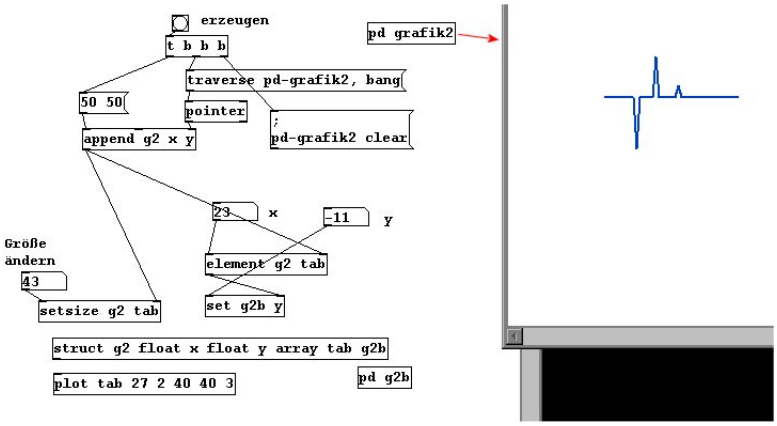
patches/5-2-3-1-data-structures9.pd



Mit „setsize“ ändern wir die Größe:



Und über einen Umweg ändern wir die Variable „y“: Mit „element“ erhalten wir den Zugang.



Eventuell wird sich die Gestalt der data structures in Zukunft noch ändern (Stand Juni 2008). Des Weiteren sind noch einige Spezialfunktionen für die data structures in der originalen Pd-Dokumentation zu sehen.

5.2.4 Für besonders Interessierte

5.2.4.1 GEM

Wie schon vorab sei auch in diesem Zusammenhang erwähnt, dass es für Pd ein Zusatzprogramm zur Videoerzeugung namens GEM gibt.

http://gem.iem.at

Nachwort

Nach dem Studium und der Erprobung der vorgestellten Techniken ist Ihre Fantasie hoffentlich zur Genüge angeregt, um alle möglichen Objekte miteinander zu kombinieren – denn genau darauf ist Pd ausgerichtet.

Dem weiteren Studium sind natürlich keine Grenzen gesetzt. Etliche andere Bücher über digitale Signalverarbeitung, gerade das Buch „Theory and Techniques of Electronic Music“ vom Pd-Hauptautor Miller Puckette, warten darauf, gelesen zu werden. Natürlich sind nun auch tiefere Kenntnisse von Akustik, Aufnahmetechnik und Programmierung angebracht. Auch der Umgang mit einem Sequencer-Programm zur einfachen Anordnung von Klängen empfiehlt sich beim Programmieren von Klang. Am Wichtigsten bleiben aber freilich die künstlerische Schaffensfreude und die ästhetische Reflexion.

Bei weiteren Fragen zu Pd steht Ihnen immer die Pd-Community mit der "Pd-list" zur Verfügung; allerdings antworten die freundlichen Pd-Programmierer nur auf Englisch. Der Wissenstand dieses Buches entspricht dem von Pd 0.39, Ende 2007. So bleibt zu hoffen, dass es nicht allzu bald überholt sein wird.

Johannes Kreidler

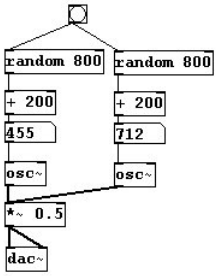
Anhang A. Lösungen

Hier finden Sie Lösungsvorschläge zu den weiteren Aufgabenstellungen am Ende der jeweiligen Kapitel. Freilich gibt es häufig noch andere richtige Ergebnisse.

2.2.1.2.8

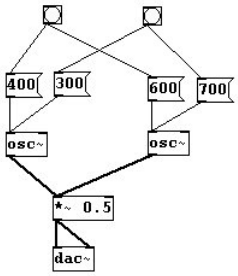
a) Zwei Zufallsmelodien gleichzeitig:

patches/a-1-zwei-zufallsmelodien.pd



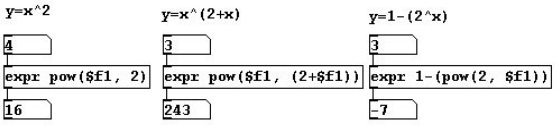
b) Mit zwei Bangs zwei verschiedene (beliebige) Intervalle anwählen:

patches/a-2-zwei-intervalle.pd



c) Mit „expr“ Exponentialfunktionen berechnen, z.B. $y = x^2$ oder $y = x^{(2 + x)}$ oder $y = 1 - (2^x)$:

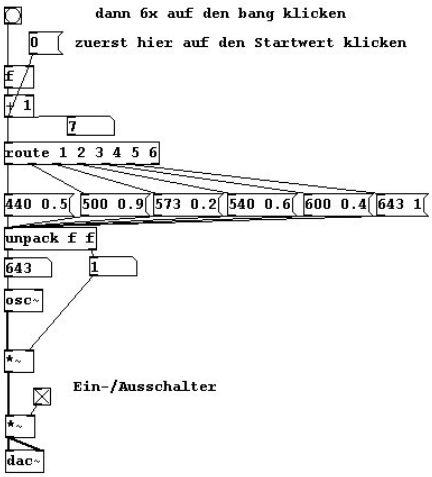
patches/a-3-exponentialfunktionen.pd



2.2.2.2.6

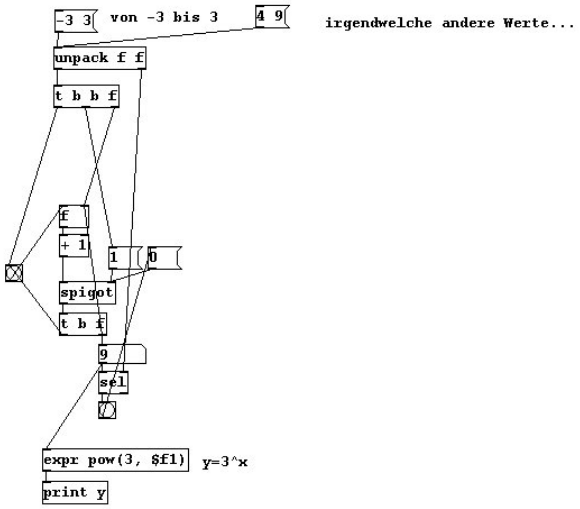
a) Eine Sequenz von Listen mit Tonhöhe und Lautstärke:

patches/a-4-listensequenz.pd



b) Eine Funktion, bei der wir mit einer Liste von zwei Zahlen, die den Start- und Endwert des x-Bereiches angeben, einen Ausschnitt berechnen können; wenn wir z. B. die Werte der Funktion $y = 3^x$ für den Bereich von $x = -2$ bis $x = 4$ ausrechnen möchten:

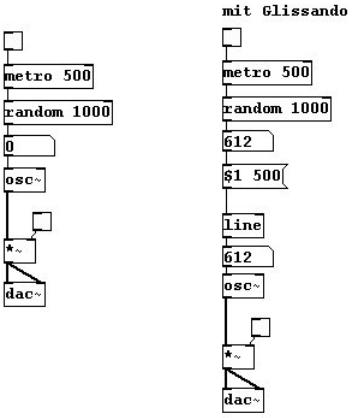
patches/a-5-funktionsausschnitt.pd



2.2.3.2.9

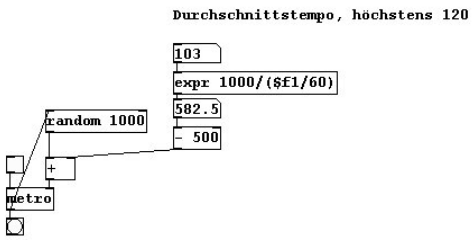
a) Eine Zufallsmelodie, die alle halbe Sekunde zum nächsten Ton springt (alternativ: glissandiert):

patches/a-6-zufallsmelodie500.pd



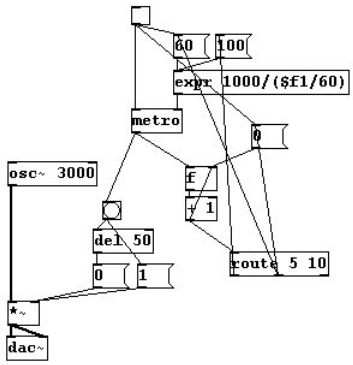
b) Ein Metronom mit unregelmäßigen Zufallsrhythmen (durchschnittliches Tempo verstellbar):

patches/a-7-unregelmetro.pd



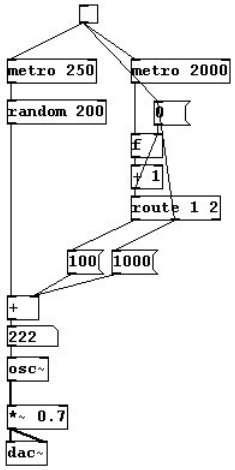
c) Ein Metronom, das immer abwechselnd fünf Schläge im Tempo Viertel = 60 und fünf Schläge im Tempo Viertel = 100 macht:

patches/a-8-zweimetro.pd



d) Eine Melodie, die abwechselnd (im Wechsel von je zwei Sekunden) eher hoch oder eher tief ist:

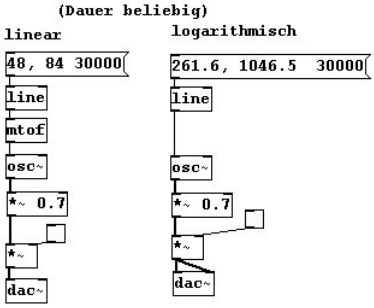
patches/a-9-hochtiefmelodie.pd



3.1.1.2.2

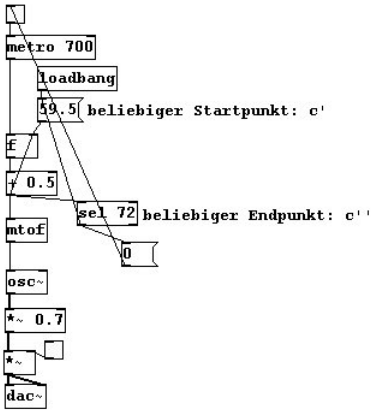
a) Ein (für unser Ohr) lineares und ein logarithmisches Glissando von c bis c'''.

patches/a-10-linloggliss.pd



b) Eine Tonleiter in Vierteltönen:

patches/a-11-viertelton.pd

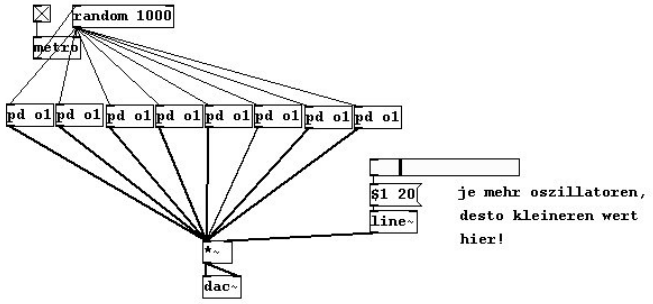


3.1.2.2.5

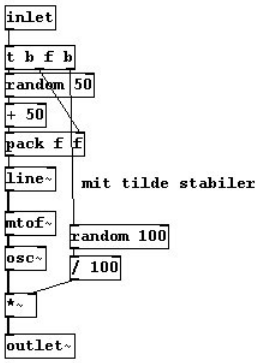
a) (random-)Glissando-Akkorde, zusätzlich noch mit random-Lautstärkeänderung jedes einzelnen Tones:

patches/a-12-randakk.pd

Dieser Hauptpatch:

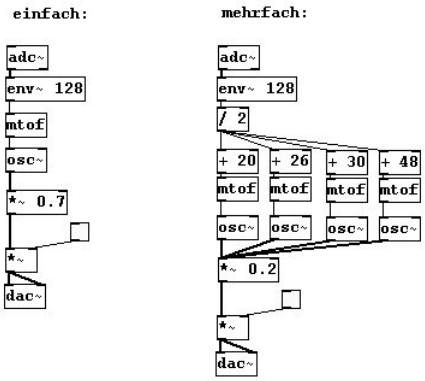


Mit diesem Subpatch „o1“:



b) Die Lautstärke des Mikrofoneingangs soll die Tonhöhe eines Oszillators steuern (oder dann auch mehrere mit verschiedenem Frequenz-Offset):

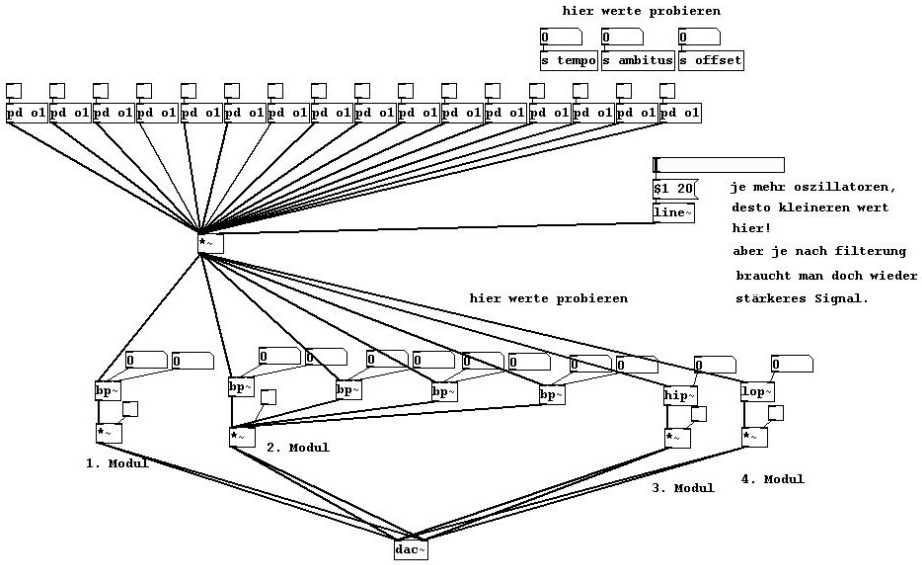
patches/a-13-adcampcont.pd



3.3.2.3

Mit den Filterungen des "Glissandoorchesters" (3.1.2.2.4) experimentieren:

patches/a-14-orchesterfilter.pd



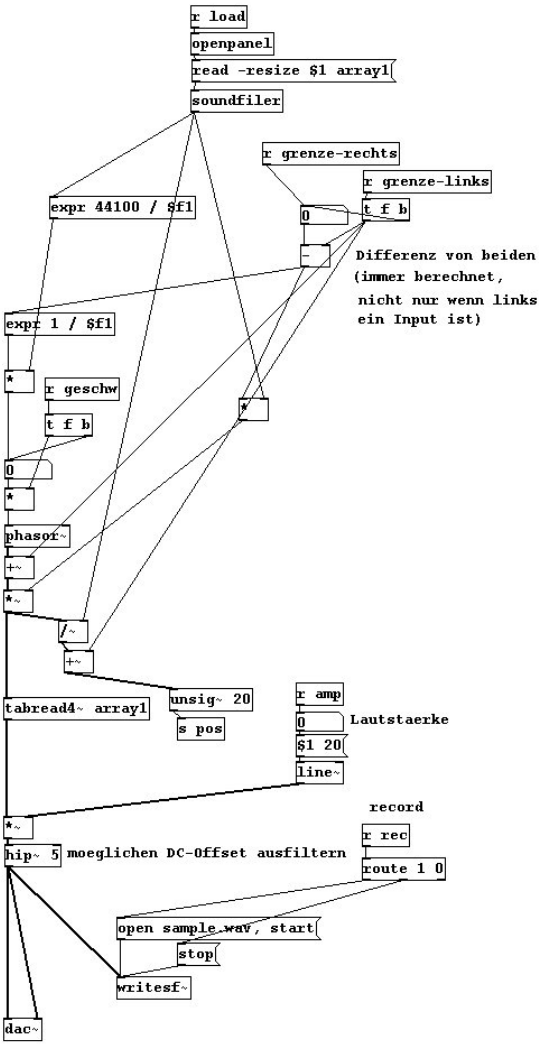
3.4.2.11

a) In den Sample-Player eine Aufnahmefunktion einbauen:

patches/a-15-aufnahmesampler.pd

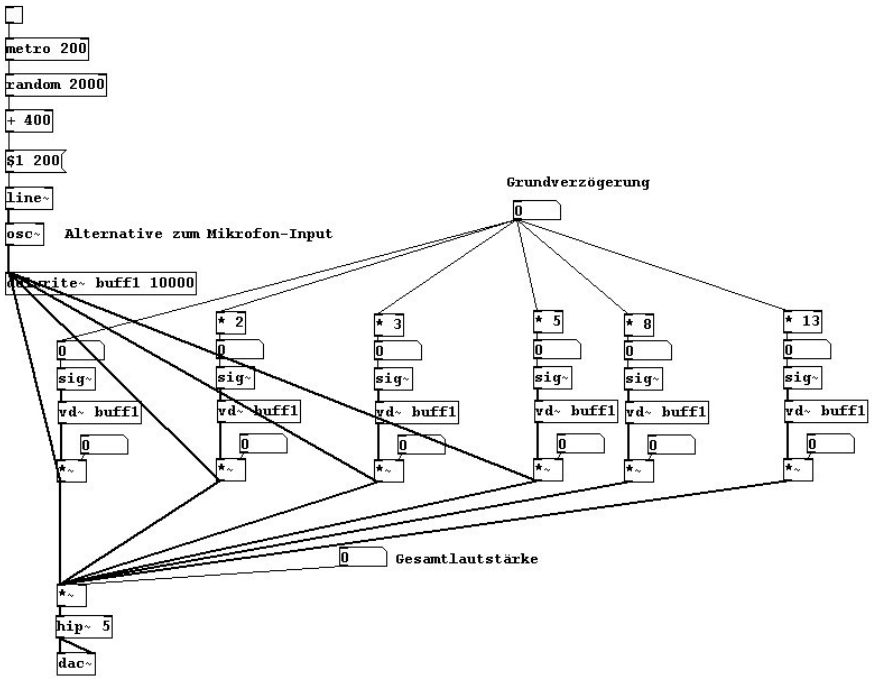
In den Hauptpatch von 3.4.2.4 ist nun noch ein Toggle integriert, der an „rec“ sendet.

Im Subpatch:



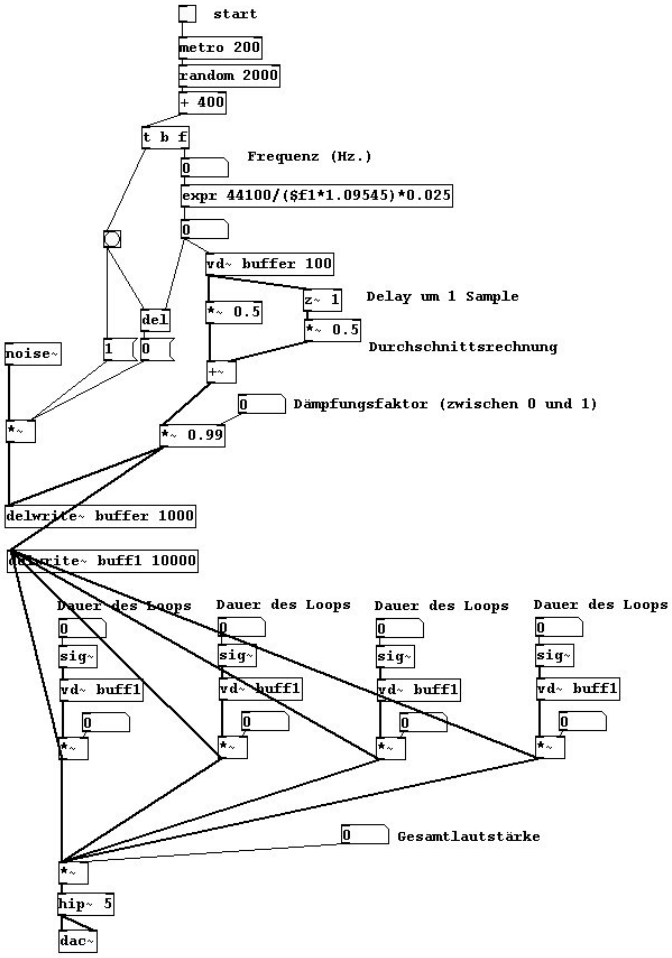
b) Einen Hall bzw. eine Textur bilden mit verschiedenen Delay-Zeiten des Eingangssignals, z. B. mit Vielfachen der Fibonacci-Reihe (die nächste Zahl ist immer die Summe der zwei vorangegangenen, also 1 2 3 5 8 13):

patches/a-16-fibodelay.pd



c) Mit verschiedenen Karplus-Strong-Klängen Texturen von variabler Dichte schaffen:

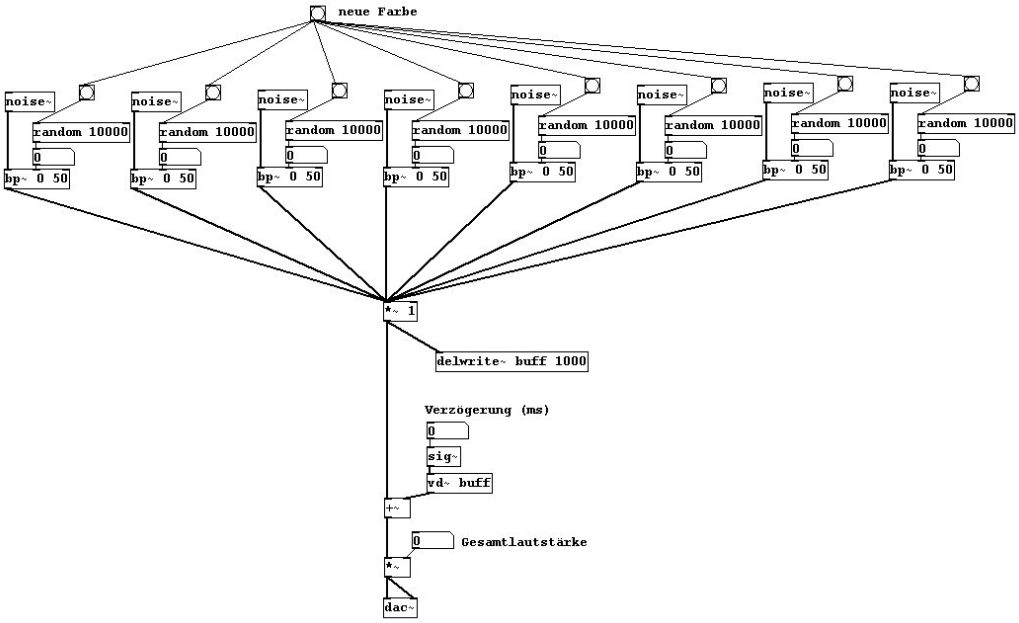
patches/a-17-karplus-text.pd



d) Den Kammfilter auf andere in den vorigen Kapiteln vorgestellte Klänge anwenden:

patches/a-18-kammfilteranwendung.pd

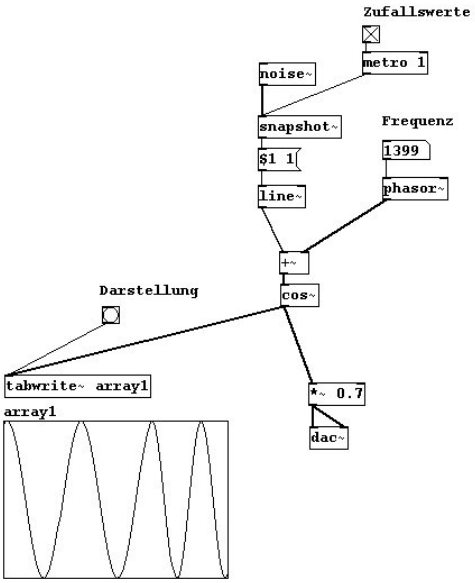
z. B. mit den „Filterfarben“ (3.3.2.1) (kleinen Wert bei Verzögerung einstellen, z. B. 15 ms):



3.5.2.4

Eine sich ständig ändernde Welle:

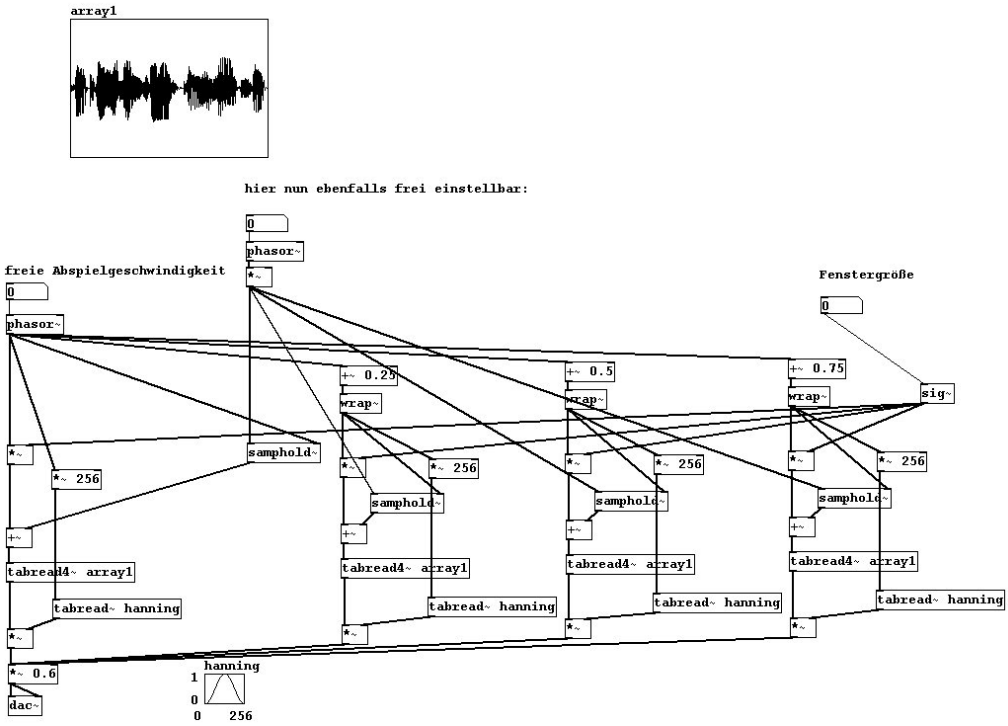
patches/a-19-wellenwandel.pd



3.7.2.3

Vier Leser, Fenstergröße variabel. Qualitäten ausprobieren!

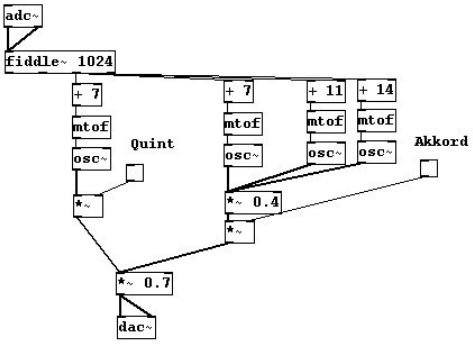
patches/a-20-vier-leser.pd



3.8.3.5

Statt dem einfachen 'Nachzeichnen' nun eine parallele Stimme im Quintabstand, oder einen Akkord parallel:

patches/a-21-followers.pd



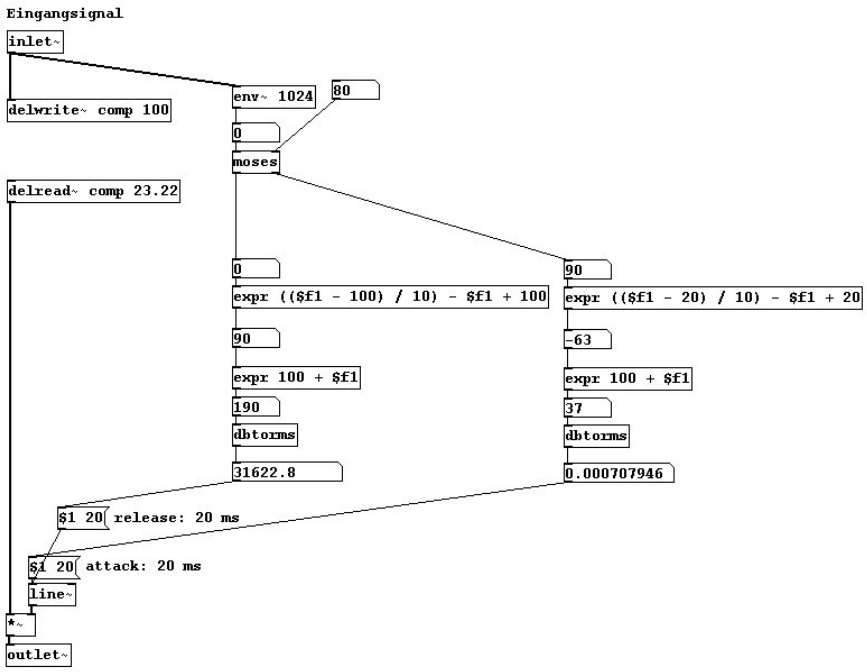
3.9.2.2

a) Ein „Expander“ – schwache Amplitudenunterschiede sollen starke Unterschiede werden:

Hierfür braucht man beim Kompressor-Patch (3.9.1.2) als Referenz einfach einen Wert über dem des Eingangssignals.

b) Eine Lautstärkeninvertierung: leises wird laut, lautes wird leise.

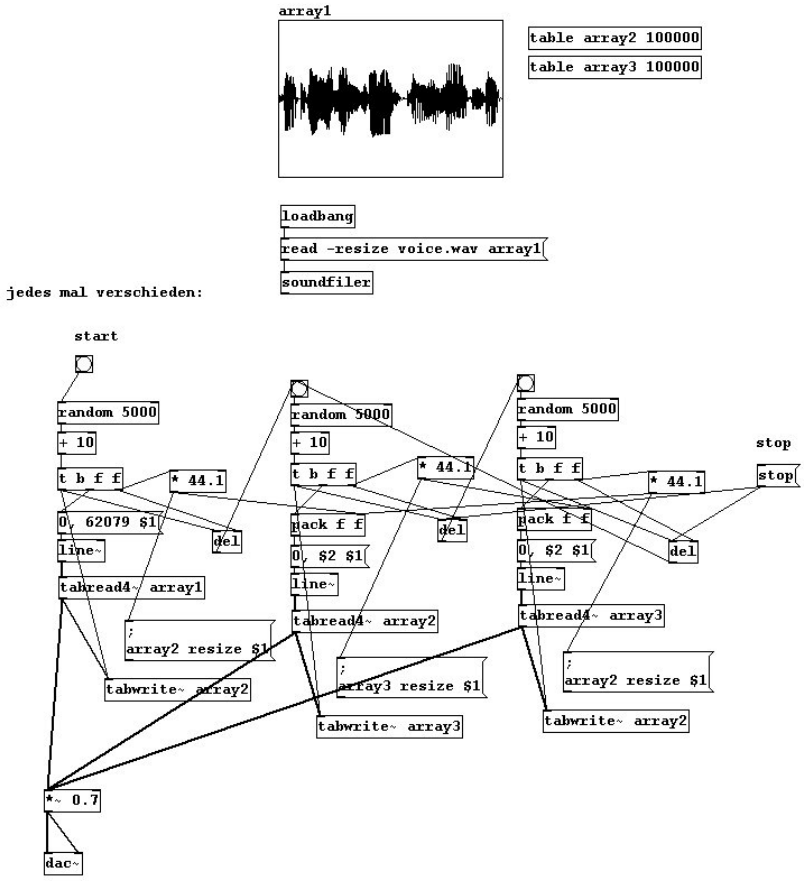
patches/a-22-ampinvert.pd



4.1.2.3

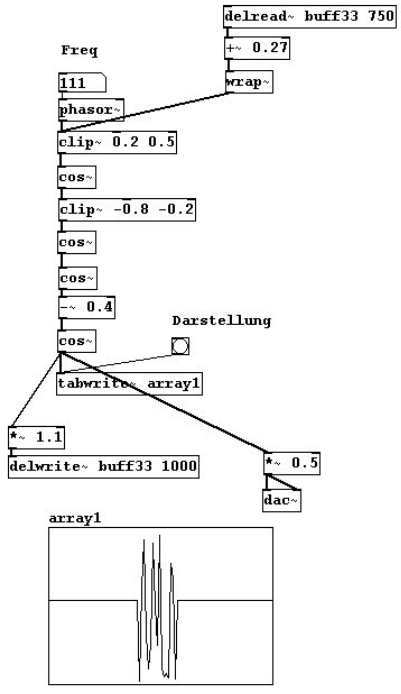
a) Nehmen Sie ein Sample auf, das in falscher Geschwindigkeit abgespielt wird, spielen Sie das Ergebnis wieder falsch ab, nehmen es auf und so weiter. Probieren Sie aus, das Sample in immer gleicher Weise falsch abzuspielen oder aber bei jedem Durchgang anders.

patches/a-23-sample-falsch.pd



b) Ein Waveshaping-Algorithmus, dessen Ergebnis mit Delay wieder oben eingespeist wird:

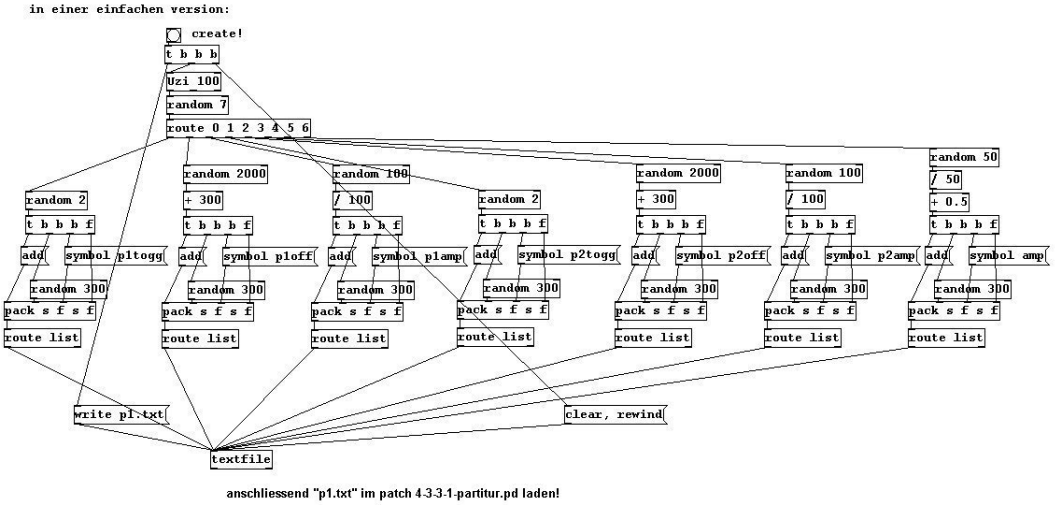
patches/a-24-waveshape-feedback.pd



4.2.2.2

Zufallsalgorithmen in eine Textdatei schreiben, mit der man dann per „qlist“ den Patch von 4.2.2.1 (in verschiedenen Geschwindigkeiten) spielen kann:

patches/a-25-textcreate.pd

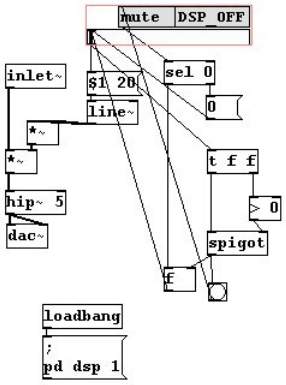


5.1.2.2

Als Abstraktionen:

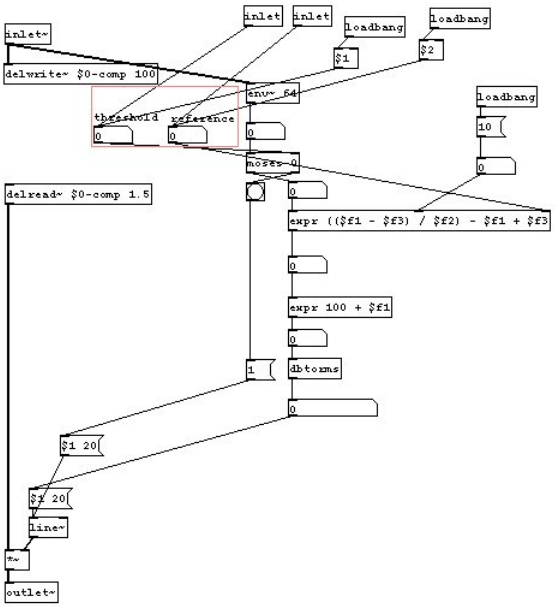
a) Den DSP-Schalter in die „dac“-Abstraktion einbauen, ebenso einen Schalter, der auf Stumm schaltet und wieder zurück:

patches/a-26-dac-erweitert.pd



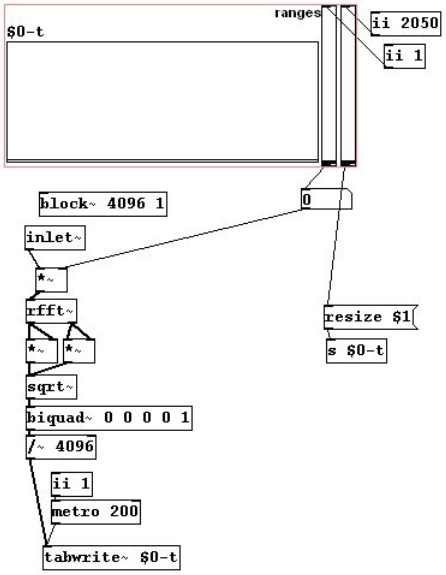
b) Einen Kompressor:

patches/a-27-compress~.pd



c) Eine Darstellung des eingehenden Spektrums:

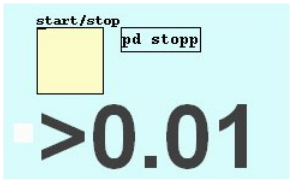
patches/a-28-spectrum.pd



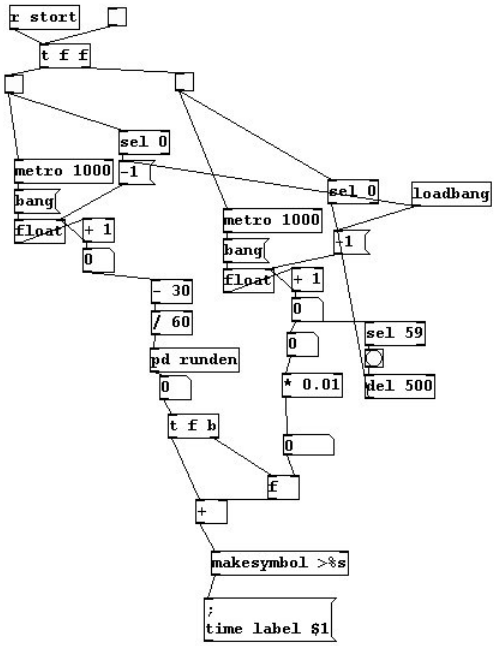
5.2.2.4

a) Eine Stoppuhr schön visualisieren:

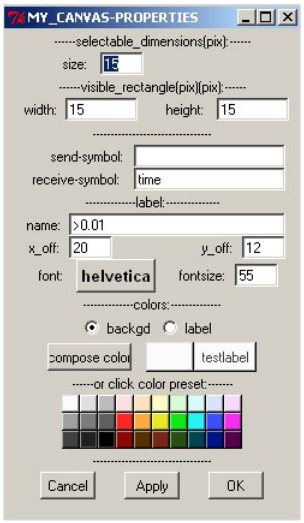
patches/a-29-stopp.pd



... mit dem Subpatch "stopp":



... und den folgenden Canvas-Einstellungen:



Im Toggle ist eingestellt, dass er zu „stort“ schickt.

b) Ein laufendes 5/8-Metrum schön visualisieren (ein optischer Klicktrack): siehe Patch-Datei.

patches/a-30-opt-track.pd