# Table of Contents

**Pure Data** ..... **1**

**What is real-time graphical programming?** ..... **2**
Graphical Programming ..... 2
Real Time ..... 2

**Installing on Mac OS X** ..... **3**

**Installing on Microsoft Windows** ..... **6**

**Installing Pure Data on Ubuntu** ..... **16**
Installing libflac7 and libjasper ..... 16
Installing Pure Data ..... 18

**Installing Pure Data on Debian** ..... **21**

**Configuring Pure Data** ..... **25**
Basic Configuration ..... 25
Audio drivers ..... 25
MIDI drivers (Linux only) ..... 26
Audio Settings ..... 26
Sample rate ..... 26
Delay (msec) ..... 26
Input Device ..... 27
Output Device ..... 27
MIDI Settings ..... 27
Test Audio and MIDI ..... 28
Advanced Configuration ..... 28
Startup Flags ..... 30
Path ..... 31
Platform-Specific Configuration Tools ..... 32

**Starting Pure Data** ..... **35**
Starting PD with the Clickable Icon ..... 35
Starting Pd via Command Line ..... 35
Linux (from xterm) ..... 35
Mac OSX (from Terminal.app) ..... 35
Windows (from the DOS shell or Command Prompt) ..... 35
Starting Pd from a Script ..... 36
Windows ..... 37
Linux and OS X ..... 37
Advanced Scripting for Starting Pd ..... 37

**The Interface** ..... **39**
The main PD window ..... 39
Starting a New Patch ..... 40
Interface Differences in Pure Data ..... 41
Linux ..... 41
Mac OS X ..... 41
Placing, Connecting and Moving Objects in the Patch ..... 42
Edit Mode and Play Mode ..... 45
Messages, Symbols and Comments ..... 47
GUI Objects ..... 48
GUI Object Properties ..... 49

# Table of Contents

**The Interface**

Arrays and graphs....50
Graph....51
A Note on using GUI Objects....51

**Troubleshooting....52**

I don't hear any sound!....52
There are clicks, glitches or crackles in the test tone!....52
The test tone sounds distorted!....52
I'm not seeing any audio input!....52
I don't see any MIDI input!....52
I get the message "... couldn't create" when I type an object's name and there's a dashed line around my object!....52
I get the message "... couldn't create" when I open a patch and there's a dashed line around my object!....53
I get the message "error: signal outlet connect to nonsignal inlet (ignored)" when I open a patch....53
I get the message "error: can't connect signal outlet to control inlet" and I cannot connect two objects together!....53
I get the message"error: DSP loop detected (some tilde objects not scheduled)" when I click "Audio ON", and the sound is not working!....53
I get the message "error: stack overflow" when I connect two Dataflow Objects together!....54
I get the error message "connecting stream socket: Network is unreachable" when I start Pd!....54

**What is digital audio? ....56**

Frequency and Gain....56
Sampling Rate and Bit Depth....56
Speed and Pitch Control....56
Volume Control, Mixing and Clipping....56
The Nyquist Number and Foldover/Aliasing....57
Block Size....57
It's All Just Numbers....57

**Building a Simple Synthesizer....58**

Introduction....58
Downloads ....58
Oscillators....58
Sine Wave Oscillator....59
Sawtooth Wave Oscillator....59
Square Wave Oscillator....59
Other Waveforms....60
Frequency....60
MIDI and Frequency....61
Additive Synthesis....62
Low Frequency Oscillators & Modulation....63
Amplitude Modulation....63
Ring Modulation....64
Frequency Modulation....65
Pulse Width Modulation....66
Math & Logic Operations....67
Filters....68
The Envelope Generator....71
The Amplifier....76
Using a Slider....76
Using [line~], [vline~] and [tabread4~]....77

# Table of Contents

**Building a Simple Synthesizer**
Controlling the Synthesizer....78
Input from the Computer Keyboard....79
Input from a MIDI Keyboard....79
Building a 16-Step Sequencer....80
Hot and Cold....81
Storing and Retrieving MIDI Note Values....82
The Finished 16-Step Sequencer Patch....82
A Four Stage Filtered Additive Synthesizer....83
The Input Stage....84
The Oscillator Stage....84
The Filter Stage....85
The Amp Stage....85
Subpatches....85

**Streaming Audio with PureData....89**
Creating the mp3cast object....89
Add an osc~....89
mp3cast~ Settings....89
Start the Stream....91
Streaming from The Mic....91
Disconnect....92

**Dataflow tutorials....93**
Messages....93
Math....93
Three little bits - temperature, order and depth....94
Inlets: hot and cold....94
Order of connecting and [trigger]....96
Depth first message passing....98
Invisible connections, crossing borders, reusing code....100
Send, receive, throw and catch....100
Subpatches....102
Abstractions....104
Dollarsigns....106
Pretty interfaces and two-dimensional data....108
Graph on parent....108
Arrays + graphs = tables....111

**Glossary....116**
Glossary Terms....116
Abstraction....116
ADC....116
ADSR....116
Aliasing....116
Argument....116
Array....116
ASIO....117
Attack....117
Audio Driver....117
Bang....117
Bit Depth....117
Buffer....117
Canvas....117

# Table of Contents

**Glossary**

Clipping....117
Cold and Hot....118
Comment....118
Creation Argument....118
Cutoff Frequency....118
DAC....118
Decay....118
Decibel....118
Delay....118
Distortion....119
Dollar Sign....119
Dynamic Range....119
Edit Mode....119
Envelope....119
External....119
External Library....120
Filter....120
Feedback....120
Float or Floating Point....120
Foldover....120
Frequency....120
Gain....120
Glitch....121
Graph....121
Graph on Parent....121
GUI element....121
Hot and Cold....121
Hradio....121
Hslider....121
Herz or Hz....121
Inlet....121
Integer....122
JACK....122
Latency....122
Linear....122
Logarithmic....122
Loudness....122
Message....122
MIDI....122
MME....123
Monophonic....123
Noise Floor....123
Note....123
Number....123
Nyquist Number....123
Object....123
Octave....124
Oscillator....124
OSS....124
Outlet....124
Pass Band....124
Patch....124
Path....124

# Table of Contents

**Glossary**

Pitch....124
Play Mode....125
Polyphonic....125
Portaudio....125
Property....125
Radio....125
Real-time....125
Release....125
Resonance....125
Sample....125
Sampler....126
Sampling Rate....126
Sequencer....126
Self-noise....126
Send and Receive....126
Shell....126
Slider....126
Startup Flag....127
Stop Band....127
Subpatch....127
Sustain....127
Symbol....127
Synthesizer....127
Table....127
Toggle....127
Truncate....127
Variable....128
Vector Based Graphics....128
Velocity....128
Voices....128
Vradio....128
Vslider....128
VU....128
White noise....128
Word Length....128
Working Directory....128

Core Pure Data....129

IEMLIB....132
ZEXY....135
MAXLIB....138
PDP....139
PiDiP....142
GEM....143

**PD Links....148**

Pure Data Software....148
Externals....148
Linux Distributions with PD....148
Tutorials & Examples....148
Getting Help....149

# Table of Contents

**License**.....150

**Authors**.....151

**General Public License**.....154

# Pure Data

**Pure Data** (or **PD**) is a **real-time graphical programming environment** for **audio**, **video**, and **graphical processing**. Because all of these types of media are handled as data in the program, many fascinating opportunities for cross-synthesis between them exist. Sound can be used to manipulate video, which could then be streamed over the internet to another computer which might analyze that video and use it to control a motor-driven installation. PD is commonly used for live music performance, VeeJaying, sound effects composition, interfacing with sensors, cameras and robots or even interacting with websites.

The core of Pd is written and maintained by Miller S. Puckette (http://crca.ucsd.edu/~msp/) and includes the work of many developers (http://www.puredata.org/), making the whole package very much a community effort.

The community of users and programmers around PD have created additional functions (called "externals" or "external libraries") which are used for a wide variety of other purposes, such as video processing, the playback and streaming of MP3s or Quicktime video, the manipulation and display of 3-dimensional objects and the modeling of virtual physical objects.

PD runs on **Linux**, **Windows** and **Mac OS X**, and there is a wide range of external libraries available which give PD additional features.

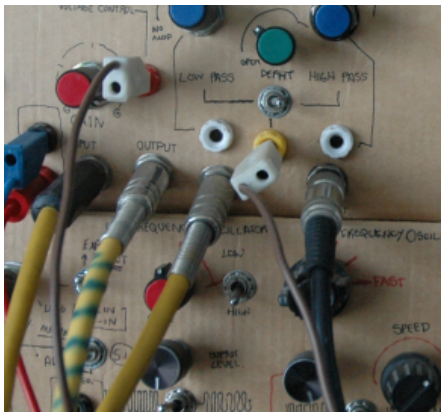
# What is real-time graphical programming?

Traditionally, computer programmers used text-based **programming languages** to write applications. The programmer would write lines of code into a file, and then run it afterwards to see the results. While this way of programming is very efficient for skilled programmers, many sound or visual artists as well as other non-programmers find this a difficult and non-intuitive method of creating things.

## Graphical Programming

**Pure Data**, on the other hand, is a **graphical programming environment**. What this means is that the lines of code, which describe the functions of a program and how they interact, have been replaced with visual objects which can be manipulated on-screen. Users of Pure Data can create new programs (**patches**) by placing functions (**objects**) on the screen. They can change the way these objects behave by sending them **messages** and by connecting them together in different ways by drawing lines between them.

This visual metaphor borrows much from the history of 20th Century electronic music, where sounds were created and transformed by small electronic devices which were connected together via **patch cables**.




The sounds that were heard were the result of the types of devices the composer used and the way in which she or he connected them together. Nowadays, much of this electronic hardware has been replaced by computer software capable of making the same sounds, and many more.

## Real Time

The real advantage of Pure Data is that it works in "real-time". That means that changes can be made in the program even as it is running, and the user can see or hear the results immediately. This makes it a powerful tool for artists who would like to make sound or video in a live performance situation.

# Installing on Mac OS X

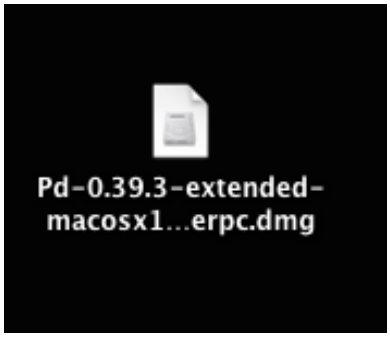
**Software name :** Pure Data Extended
**Homepage :** http://puredata.info
**Software version used for this installation :** Pd-Extended 0.39.3
**Operating System use for this installation :** Mac OS 10.4.11
**Recommended Hardware :** Any Mac running OS X

To begin the installation visit the download page for **Pure Data** ( http://puredata.info/downloads ) :

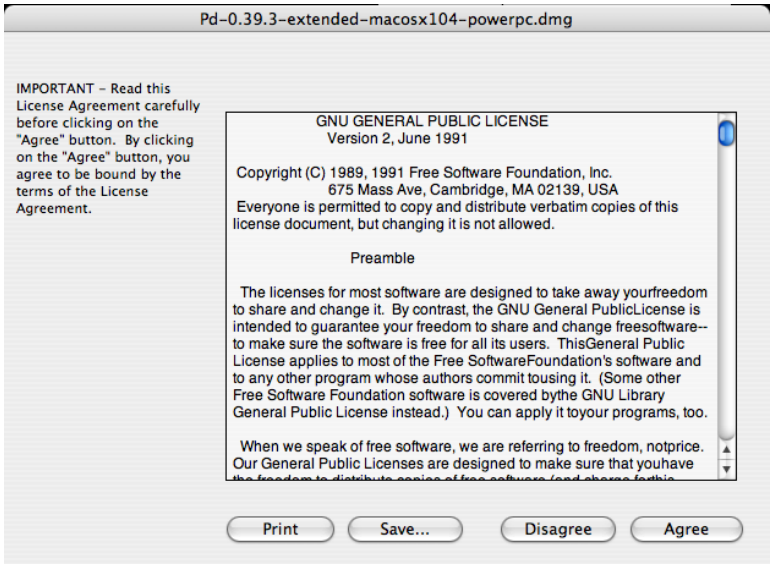

You can download either Miller Puckette's version of Pure Data, or Pure Data Extended. Miller's version of Pure Data is called "pd-vanilla" because it does not contain any external libraries or any of the features developed by the Pure Data community which are included in Pure Data Extended. We will use Pure Data Extended for this manual, so chose your installer from the "pd-extended" section of this webpage.

Since there is not a "Universal Installer" for Pure Data Extended, you will want to select the Mac OS X installer that best suits your computer. Use the one labelled "Mac OS X 10.4 i386" for the newer, Intel-processor equipped Mac computers running Mac OS 10.4. Use the "Mac OS X 10.4 PowerPC" installer if you have a Powermac or PowerBook with a G4 or G5 processor running Mac OS 10.4 "Tiger", and use the "Mac OS X 10.3" installer if you are still running "Panther" or an even older version of Mac OS X.

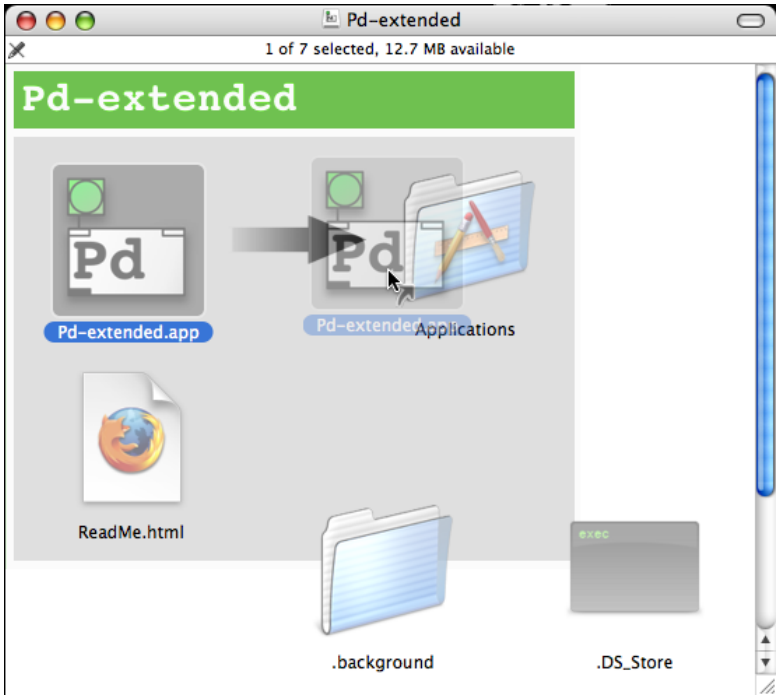
Once you've downloaded the right installer, you'll have a **.dmg (Disk Image)** on your harddrive.



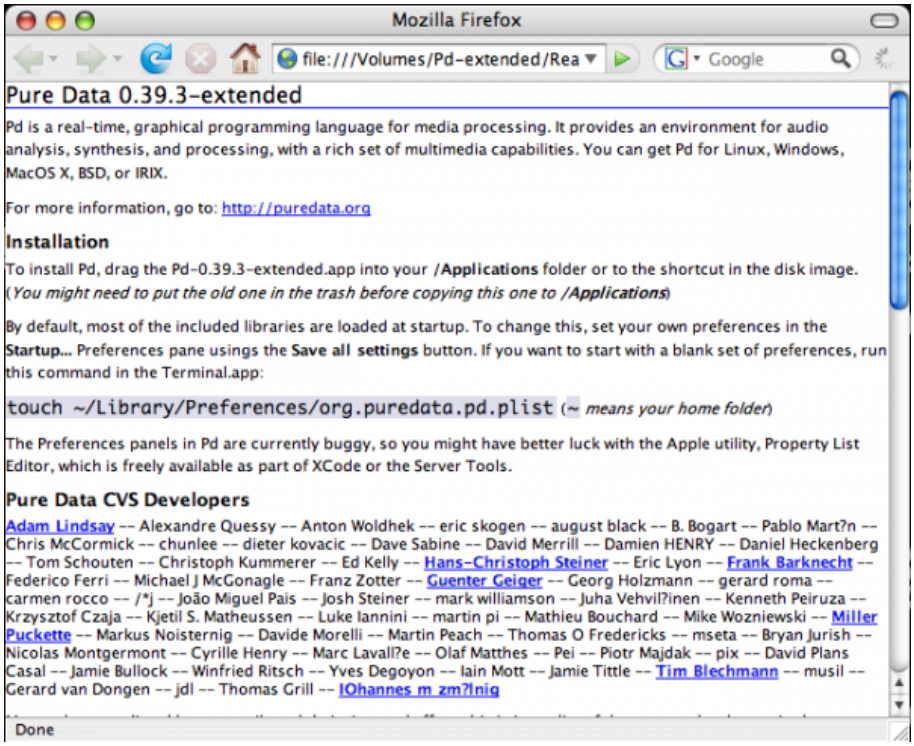

Double click to open and mount it, and you will have a chance to read and accept the License Agreement.




Once you click **"Agree"**, the Disk Image will mount and automatically open. Then simply drag the Pd-extended.app to the provided shortcut to your **Applications** folder (or to another location of your choice.) This will copy Pd-extended to your harddrive.

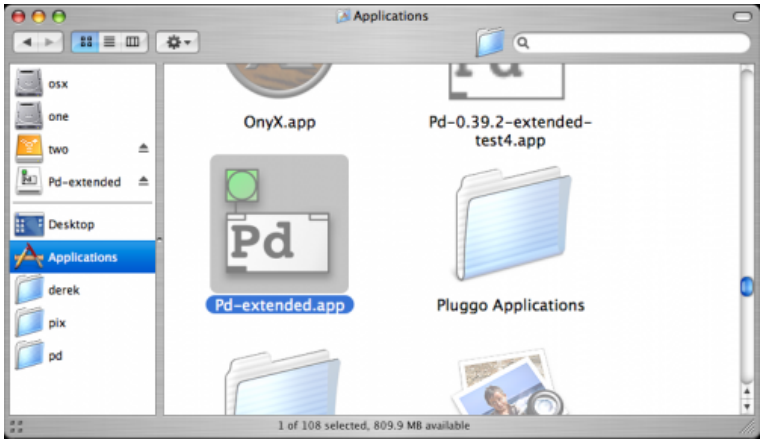



After that, make sure to check the **"ReadMe"** file for important installation information.




As indicated, the Pd-extended.app is setup by default to load most of the included external libraries. If you want to change the libraries which are loaded at startup time, or any of the other startup settings, please notice the instructions here in the "ReadMe", and be sure to read the chapter **"Configuring Pure Data"** in this manual.

From here, you can open up your **"Applications"** folder in the **Finder**, and start PD by clicking the "Pd-extended.app" icon found there.



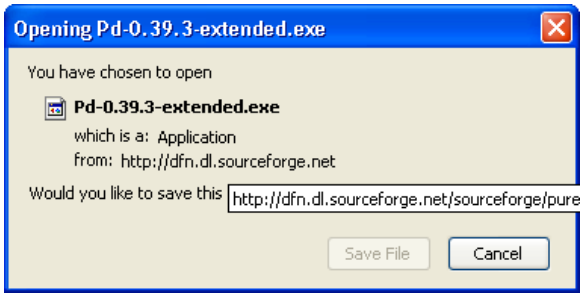
# Installing on Microsoft Windows

**Software name :** Pure Data Extended
**Homepage :** http://puredata.info
**Software version used for this installation :** Pd-Extended 0.39-3
**Operating System use for this installation :** Microsoft Windows (XP)
**Recommended Hardware :** 300 Mhz processor (CPU) minimum

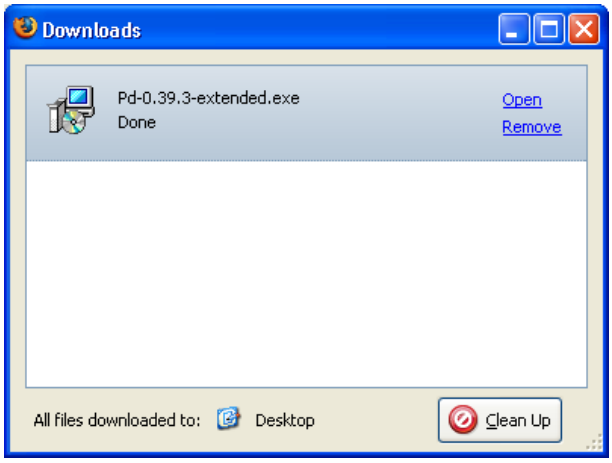
To begin the installation visit the download page for **Pure Data** ( http://puredata.info/downloads ) :


You can download either Miller Puckette's version of Pure Data, or Pure Data Extended. Miller's version of Pure Data is called "pd-vanilla" because it does not contain any external libraries or any of the features developed by the Pure Data community which are included in Pure Data Extended. We will use Pure Data Extended for this manual, so chose your installer from the "pd-extended" section of this webpage.

In the first group of links under "pd-extended"'click on the link marked "Microsoft Windows (2000/XP/Vista)" and you should see something like this (this example using **Firefox**) :



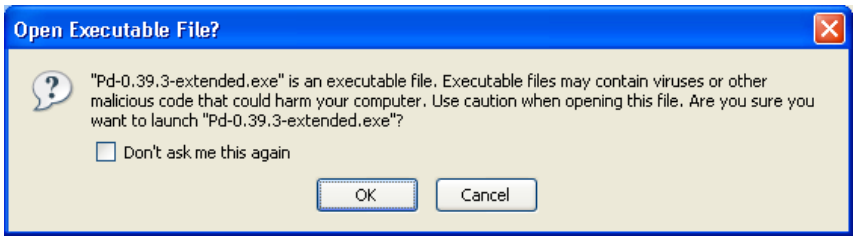
Press "OK" and the download should proceed, leaving you (hopefully) with a dialog box that informs you the download is complete. If you are using Firefox then the dialog may look something like this:



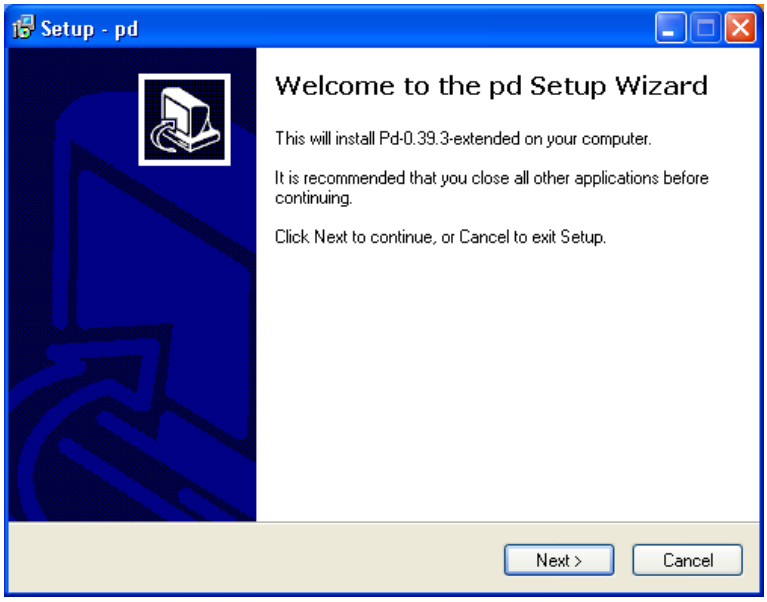

Now you can either browse your computer to look for the installer icon which will look something like this :


you can double click on this icon to start the installation process. Alternatively, you may wish to click ***Open*** in the download dialog :

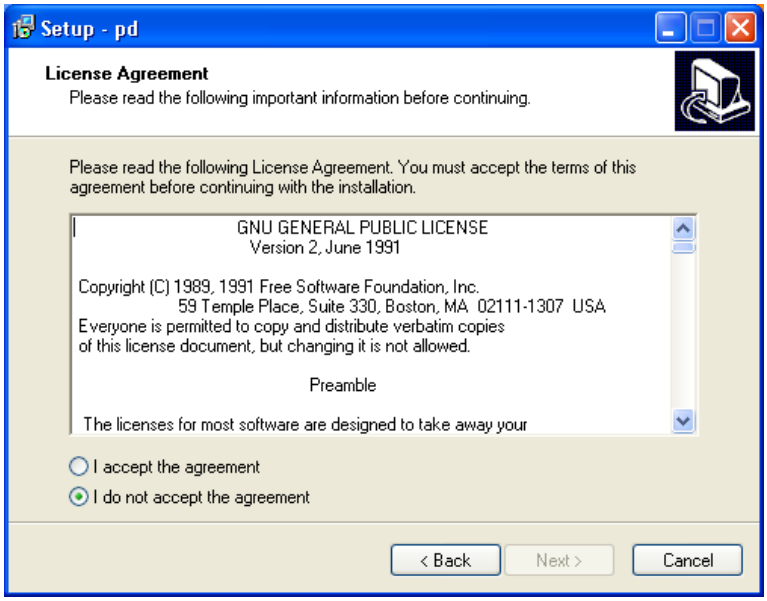

If you choose to do it this way then you may see the following window :



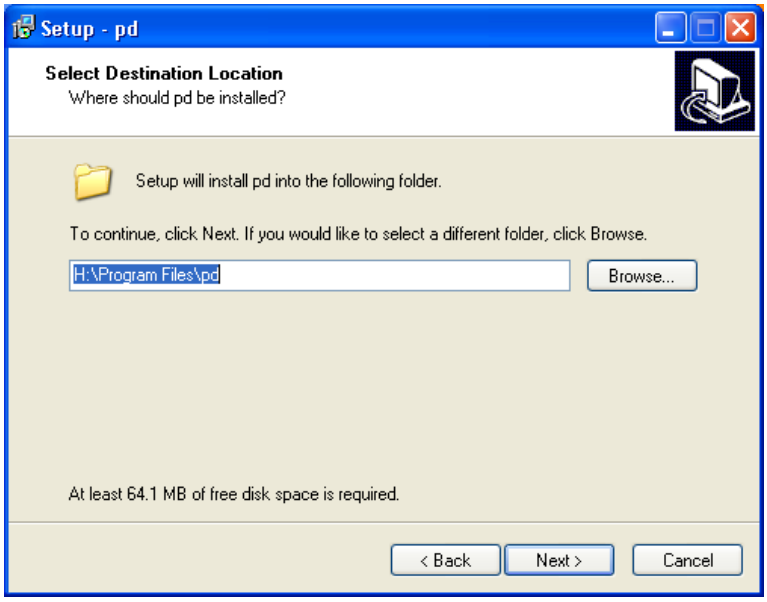

if you see this click "OK" and continue. Either of the steps above should put you in the same place, which is this :



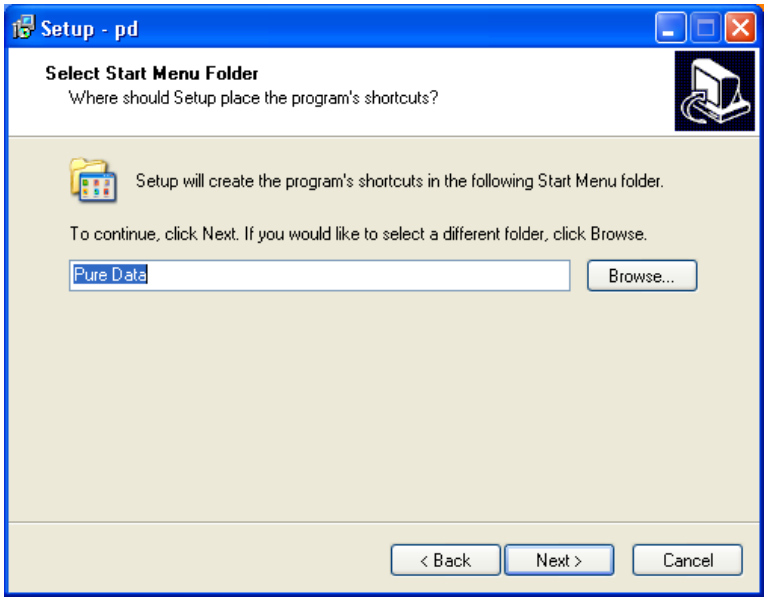

now press "Next >" and the installation process will begin. You will see this screen :



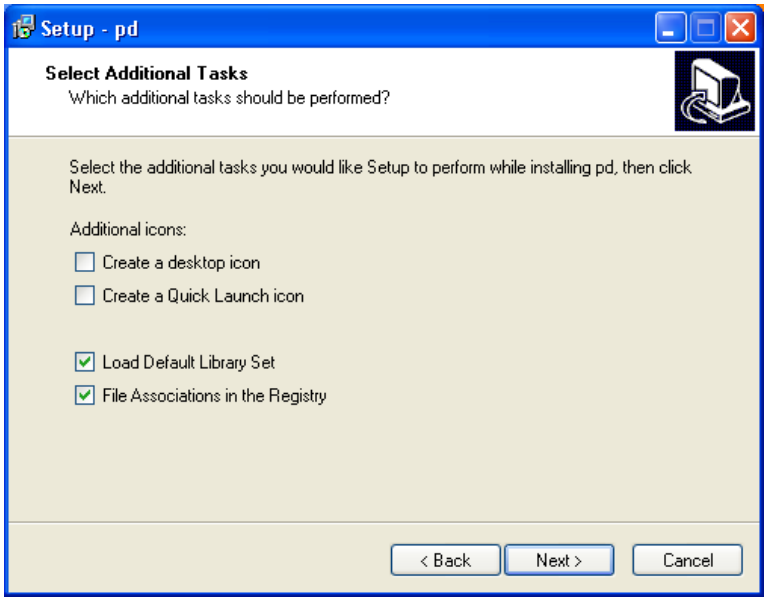

This is the standard license page. If you don't agree with the license you can't install the software. So, my recommendation is - click on the green button next to 'I accept the agreement' and then press 'Next >'. You will see the following :



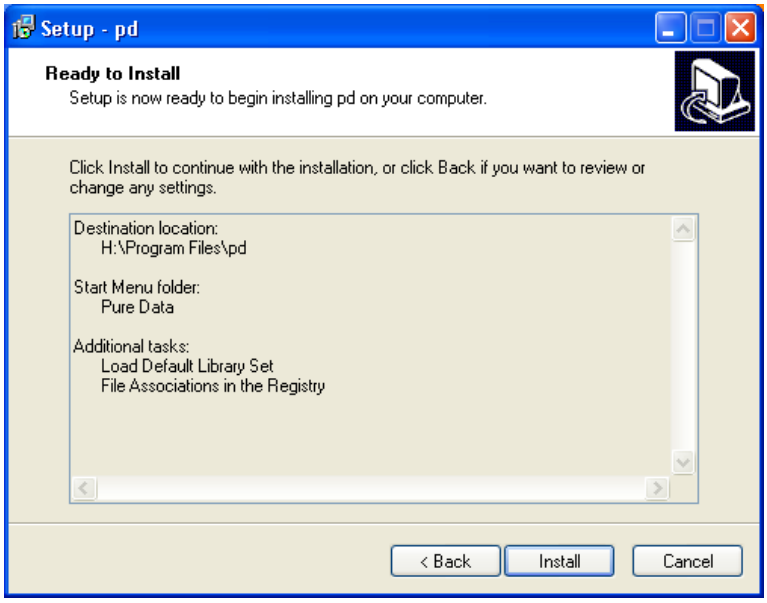

The above assists you in deciding where to install Pure Data Extended. Unless you have a good reason to, leave the default settings as they are. If you have a good reason, and know what you are doing, you can press 'Browse' and choose another place to install Pure Data Extended on your computer. If you decide to change the defaults, or keep them, youy must then press 'Next >' to continue :



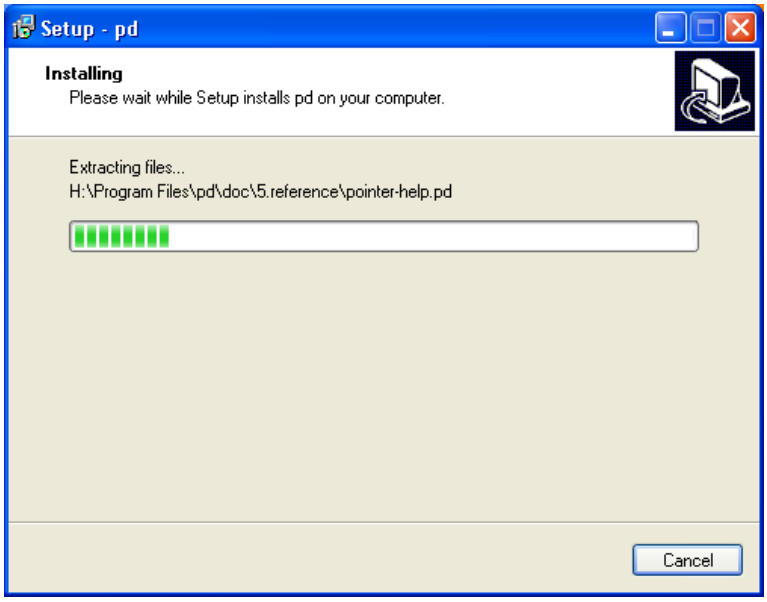

The above screen is merely choosing what to call the installation in the Windows 'Start Menu', Just leave it as it is and press 'Next >'.



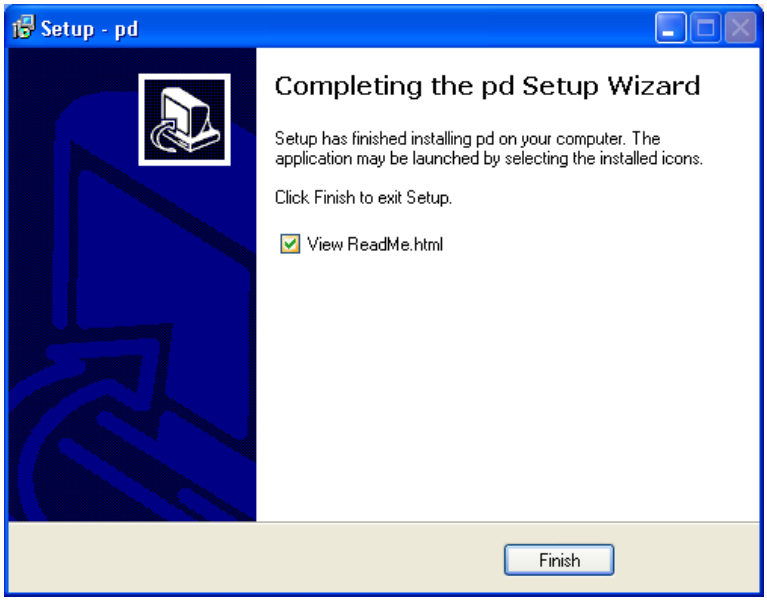

You really don't want to uncheck the last two boxes as they are necessary for the installation. The first two choices are merely cosmetic and effect the 'shortcut' icons. It doesn't matter if you check these or leave them as they are. When you are ready press 'Next>'.




The above is the summary window. Press 'Install' and the installation will commence. It might take some time depending on how quick your computer is. While you wait the installer will present you with progress bars :




Then when the installation is complete you will see a final screen :

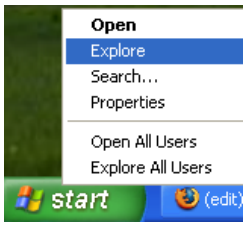



If you click 'Finish' your browser will open the (rather unattractive) Read Me page :

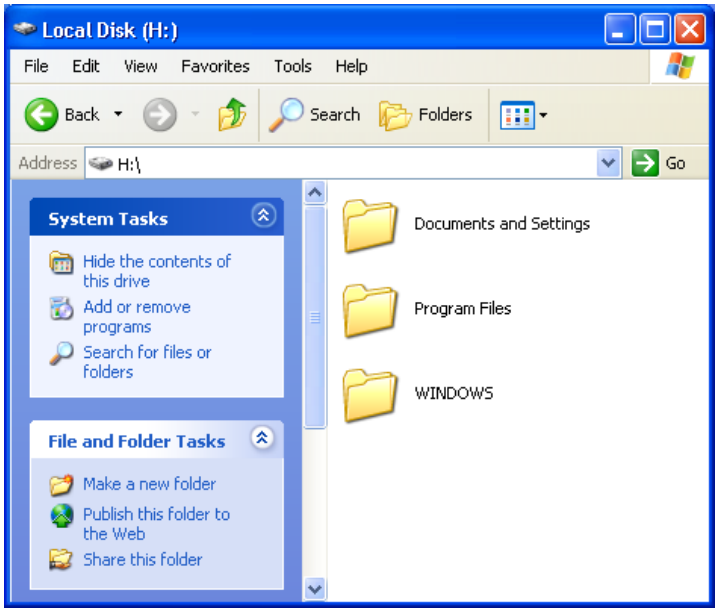

It is rather uncompelling material but it does have one useful hint...

```
"To make sure that all of the included libraries are loaded when Pd runs, double-click C:\Program
```

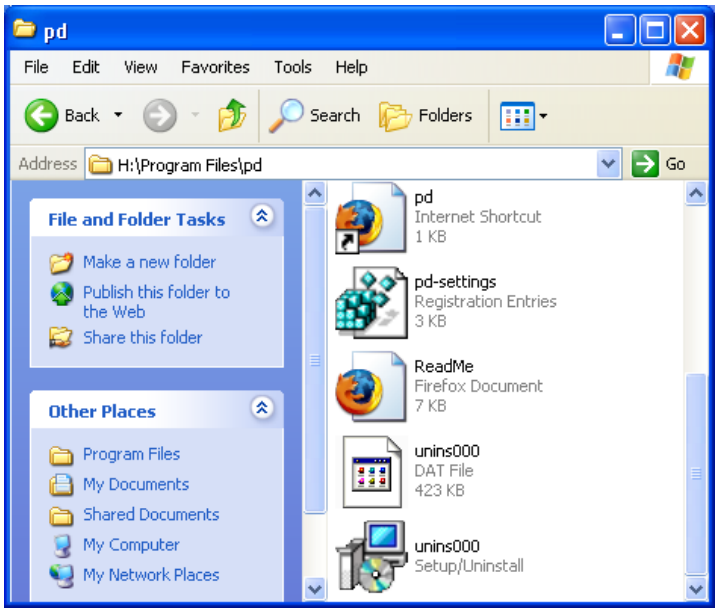
This is rather important, so you need to open the 'Program Files' in your file browser. Usually you can right-click on the Windows Start Menu to open a file browser :



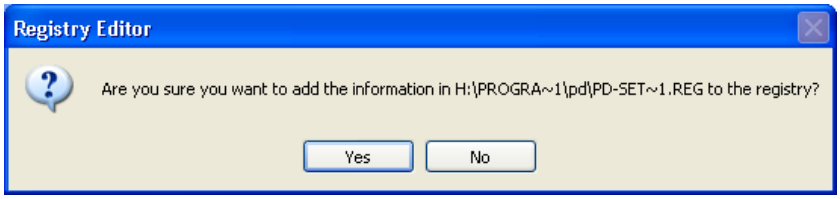

Then you will see something like this:




Double-click on 'Program Files' and the the directory called 'pd', in this window you should see a file called 'pd-settings':



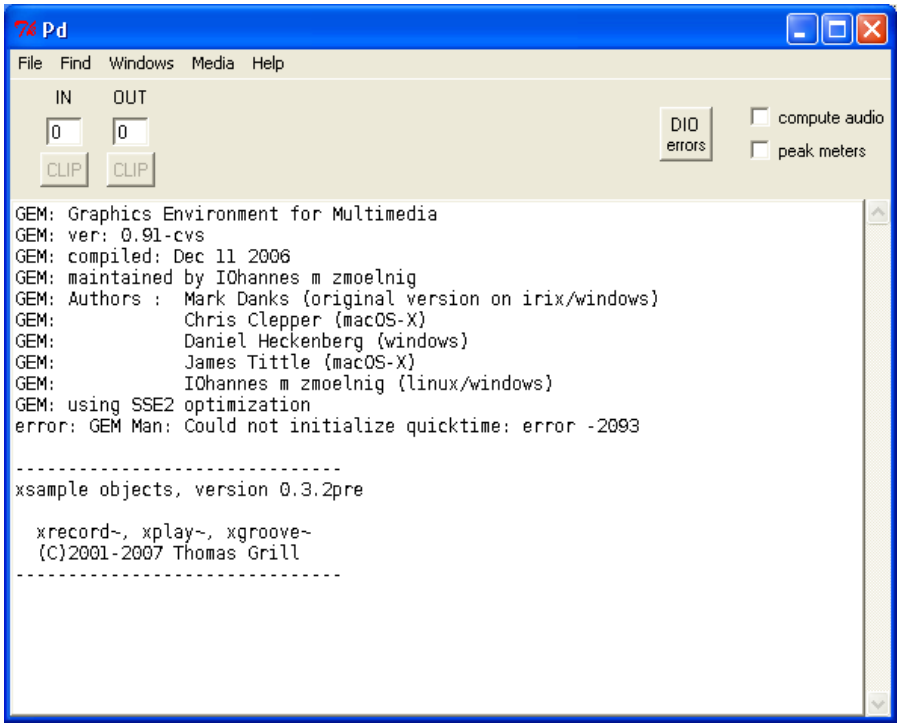

Double-click on this file and you will see the following :




Press 'Yes' :


Then press 'OK' and that window will disappear. Now you probably want to actually open Pure Data. Click on the Windows Start Menu and slide across to 'All Programs' and 'Pure Data', then finally again to the 'Pure Data' icon :


Release the mouse button and Pure Data should open :


Pd
File Find Windows Media Help
IN
OUT
0
0
CLIP
CLIP
DIO errors
compute audio
peak meters
GEM: Graphics Environment for Multimedia
GEM: ver: 0.91-cvs
GEM: compiled: Dec 11 2006
GEM: maintained by IOhannes m zmoelnig
GEM: Authors : Mark Danks (original version on irix/windows)
GEM: Chris Clepper (macOS-X)
GEM: Daniel Heckenberg (windows)
GEM: James Tittle (macOS-X)
GEM: IOhannes m zmoelnig (linux/windows)
GEM: using SSE2 optimization
error: GEM Man: Could not initialize quicktime: error -2093
xsample objects, version 0.3.2pre
xrecord~, xplay~, xgroove~
(C)2001-2007 Thomas Grill

# Installing Pure Data on Ubuntu

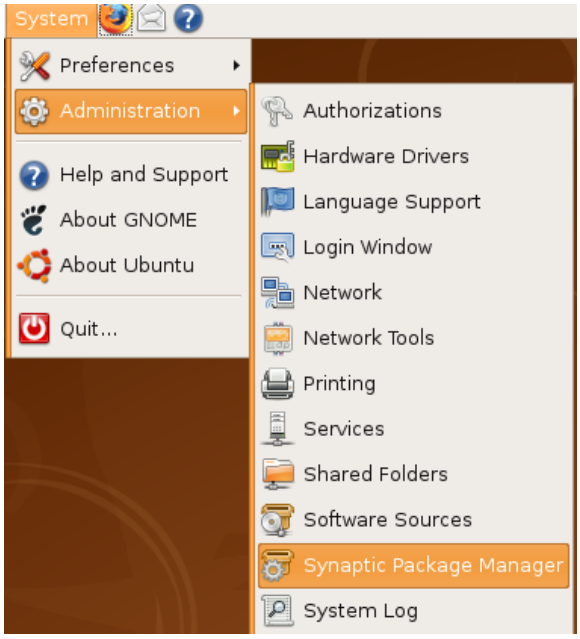
**Software name :** Pure Data Extended
**Homepage :** http://puredata.info
**Software version used for this installation :** Pd-Extended 0.39-3
**Operating System use for this installation :** Ubuntu 8.04 (tested also on 7.10)
**Recommended Hardware :** 300 Mhz processor (CPU) minimum Â–

Installation on Ubuntu Gutsy (7.10) and Ubuntu Hardy (8.04) is the same process. It is made a little tricky because Pure Data Extended requires some software that is not normally part of these operating systems but *is* included in an older version of Ubuntu. So we must indulge a short work around to get Pure Data Extended working correctly. Thankfully it is quick and simple.

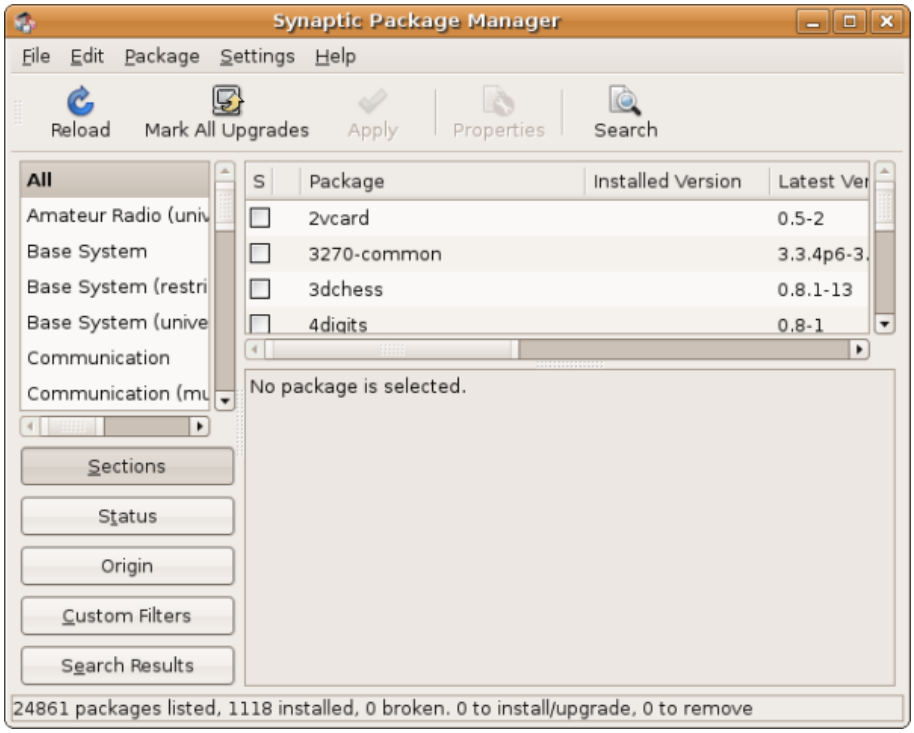
## Installing libflac7 and libjasper

Pure Data Extended requires two software 'libraries' from an older version of Ubuntu - **libflac7** and **libjasper**

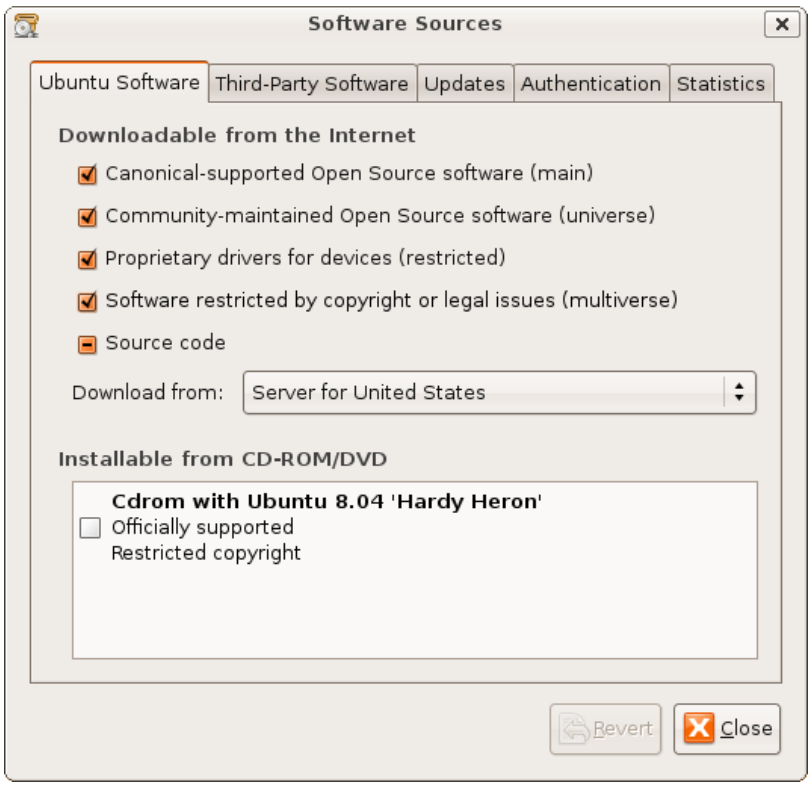
To prepare Ubuntu to install them when you install Pure Data Extended, you first need to open the **Synaptic Package Manager** :




You will be asked for a password. Enter in your *adminstrator* password (not your user password) and you will see Synaptic open.




Now we need to add the older software repositories too install these 2 software libraries. Click on **Settings** and then **Repositories** and you will see the **Synaptic Repository Manager** :




Now click on the second tab entitled **Third-Party Software**. It is here that you will now need to enter information about these two repositories:

```
deb http://archive.ubuntu.com/ubuntu/ feisty main restricted
deb-src http://archive.ubuntu.com/ubuntu/ feisty main restricted
```

You need to add them one at a time by clicking on **+ Add** and typing one of the above lines into the text field provided and then press **Add Source**. Then do the same for the next line.

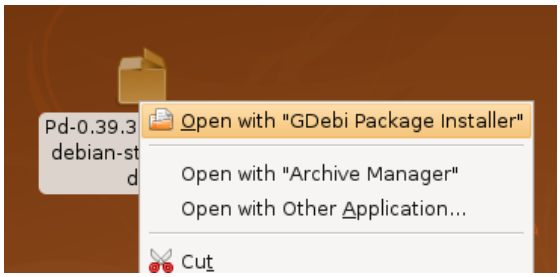
Now close the repository manager window and you will be asked to reload the repository information because it has changed. This can be done by pushing the blue **Reload** button on the Synaptic interface. Then quit the Synaptic Package Manager.

# Installing Pure Data

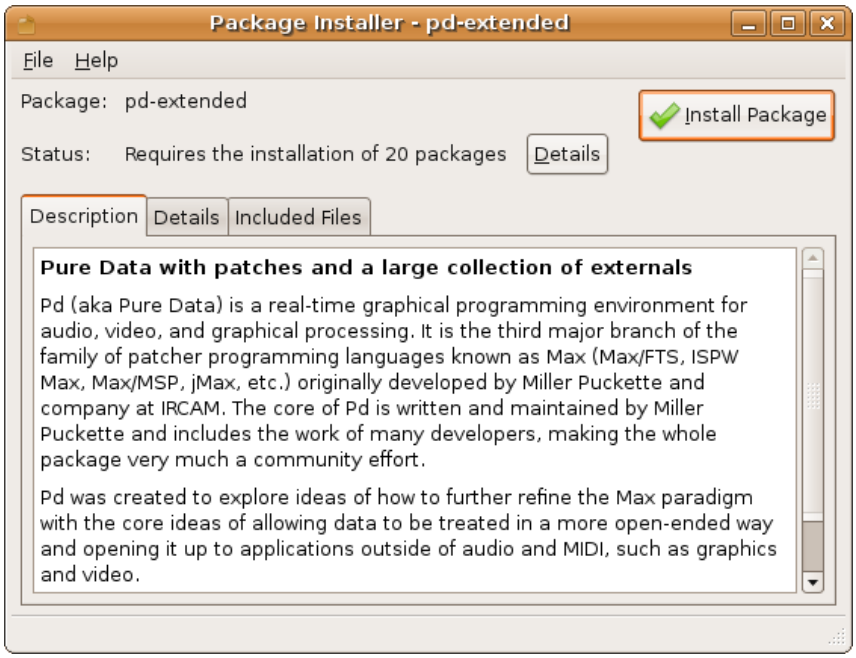
Now download the Pure Data Extended package. Visit the download page ( http://puredata.info/downloads ) :


You can download either Miller Puckette's version of Pure Data, or Pure Data Extended. Miller's version of Pure Data is called "pd-vanilla" because it does not contain any external libraries or any of the features developed by the Pure Data community which are included in Pure Data Extended. We will use Pure Data Extended for this manual, so chose your installer from the "pd-extended" section of this webpage.

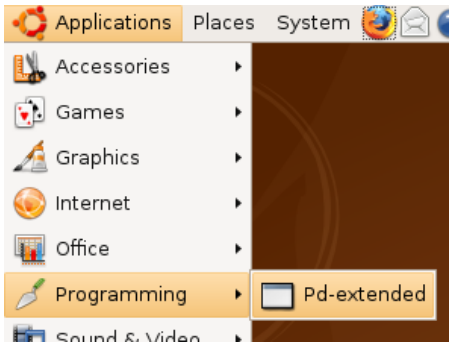
In the very first section click on the link "Debian and Ubuntu (intel i386 processor)", this will forward you to a download page. Don't do anything else, the download should start automatically. When the file has downloaded browse to the files and right click on it and choose 'Open with "GDebi Package Installer"'

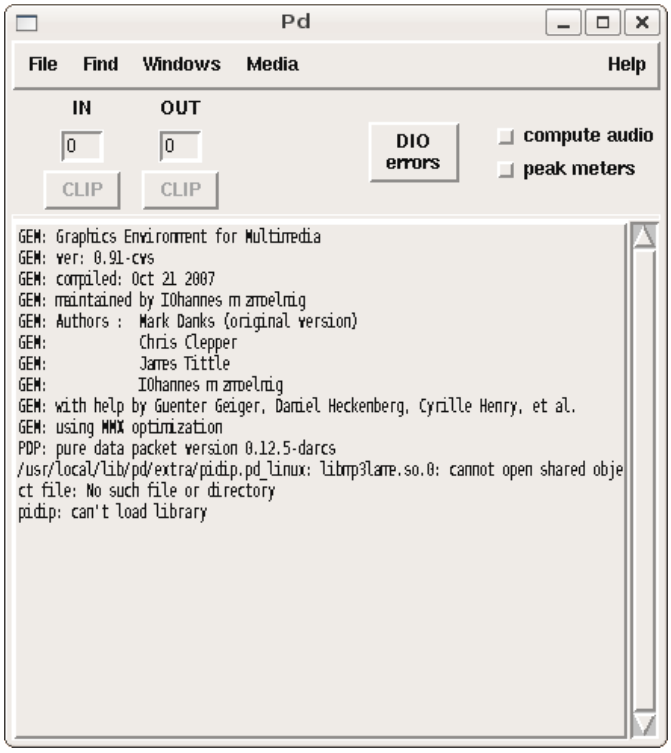


The package installer will open :




Now press **Install Package** - you will be asked to enter your password, and then Pure Data Extended will be installed. When the process is finished close GDebi and open Pure Data Extended:





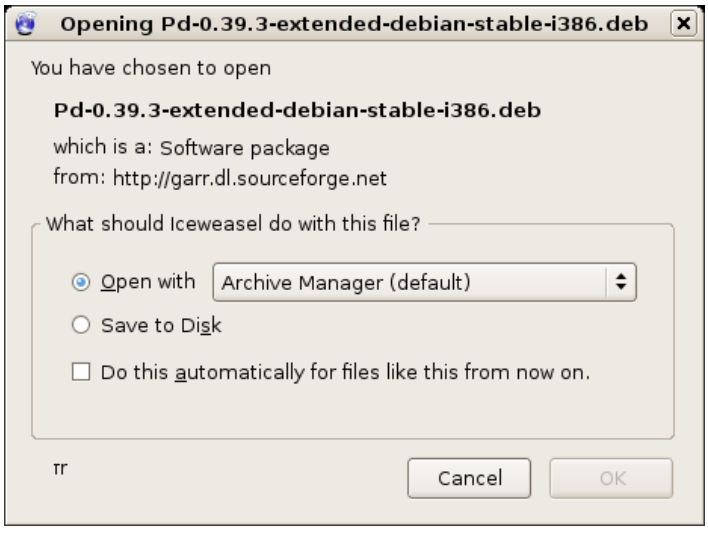

Now it is important to open the Synaptic Package Manager again and disable the two new repositories so they don't cause issues with future software installations.

# Installing Pure Data on Debian

**Software name :** Pure Data Extended
**Homepage :** http://puredata.info
**Software version used for this installation :** Pd-Extended 0.39-3
**Operating System use for this installation :** Debian Linux (4.0 rc3 stable)
**Recommended Hardware :** 300 Mhz processor (CPU) minimum

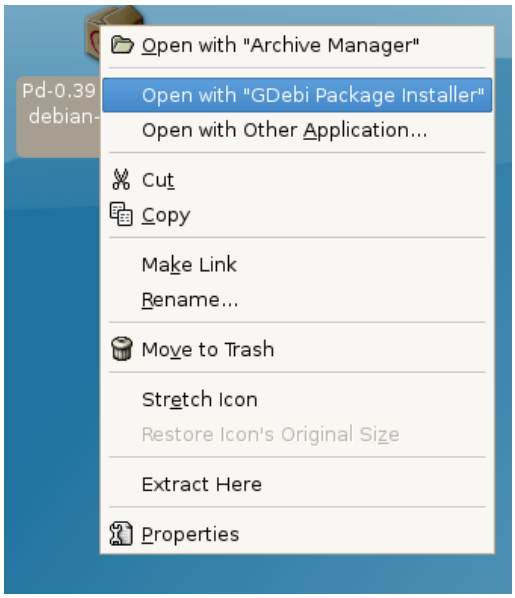
To install Pure Data Extended, first visit the download page ( http://puredata.info/downloads ) :


In the very first section click on the link "Debian and Ubuntu (intel i386 processor)", this will forward you to a download page. Don't do anything else, the download should start automatically. If you used the default Debian web browser (Ice Weasel) you will see the following :

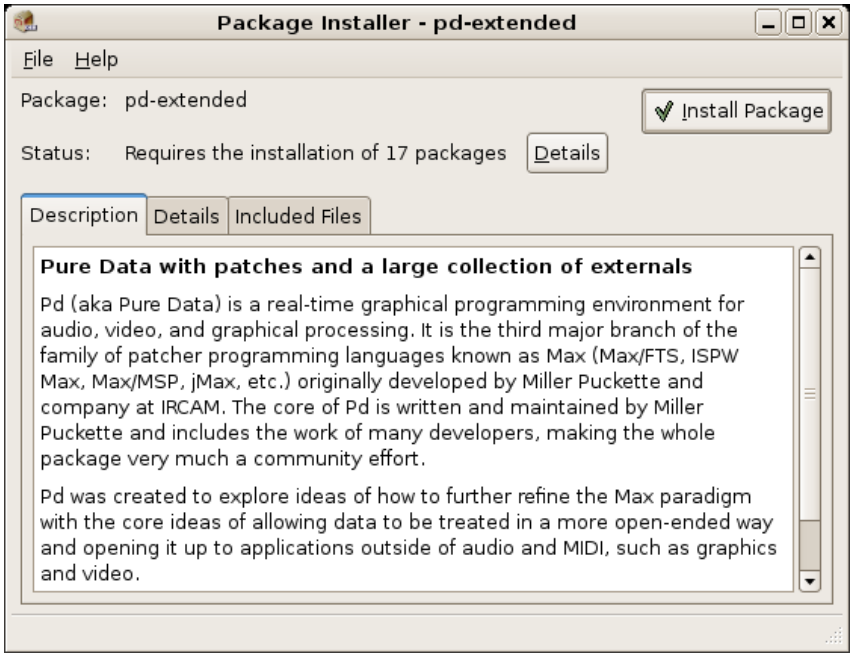


Don't use the archive manager, instead choose 'Save to Disk' and press 'OK'. When your file has downloaded you must browse to it. The default download location is the Desktop, on my Desktop I see this :


Right-click on this icon and choose 'Open with "GDebi Package Installer"':

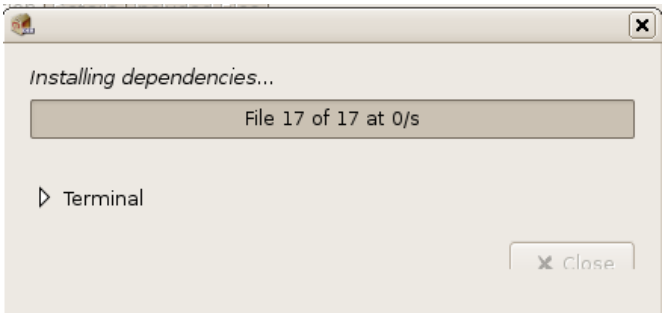



This will show something like this :

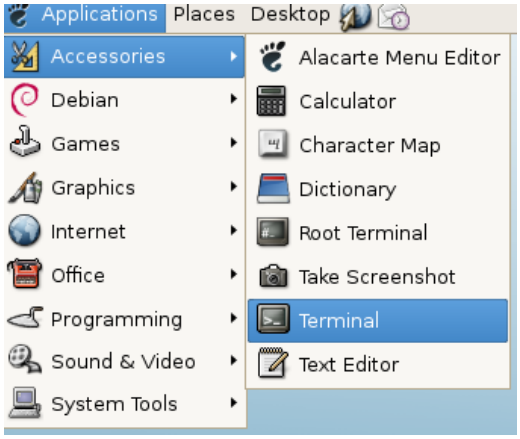



This is the general package (software) installer for Debian. Just click "Install Package" and you will be asked for the administrator ('root') password for your computer :


Enter the password and the installation process will start :

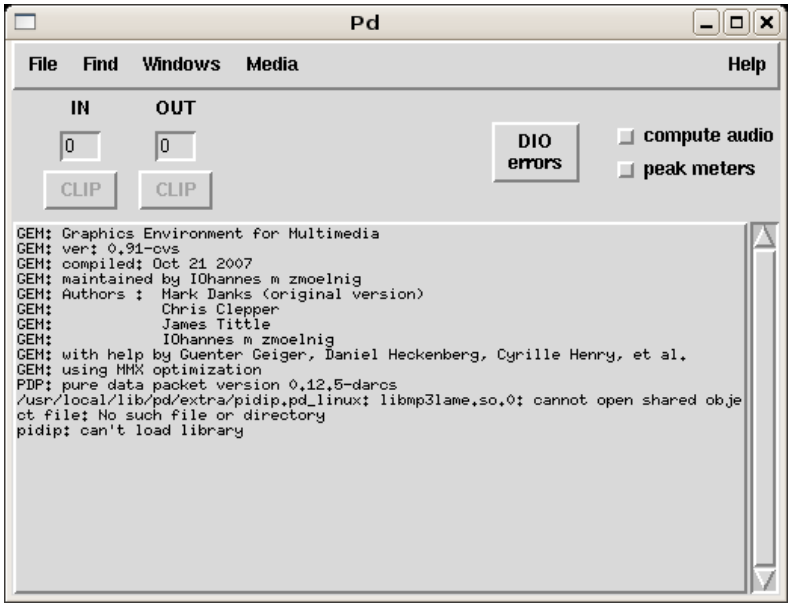



When the process has completed just open a terminal :




Type in the terminal 'pd' and press return :


and now Pure Data should appear :



# Configuring Pure Data

Pd-Extended has done a lot to make installing and setting up Pure Data easier than ever before. But every computer system is different, and each Pd user will have different needs. This section shows how to configure the most basic parts of Pd, including the soundcard and MIDI devices, as well as some advanced configuration options for those wishing to customize their installation.

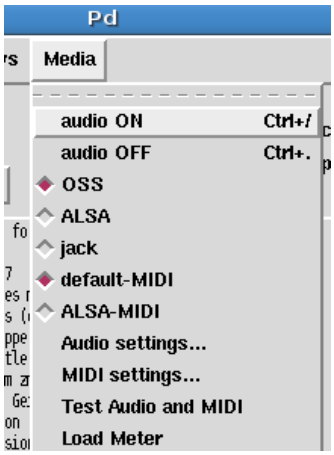
## Basic Configuration

The first thing we'll want to do once Pd is running is make sure that the audio is configured correctly. This includes choosing the correct drivers, the correct soundcard and the proper latency for your system to be both responsive and glitch-free. Also, if you have any MIDI devices (such as keyboards or fader boxes), you can set Pd up to use those as well. After that, you can test the audio and MIDI to make sure it is working properly.

### Audio drivers

Pd can use a variety of audio **drivers** to connect to the soundcard. So our first step is to chose the correct ones. This can be done via the **"Media"** menu:

**OSX :** Media -> portaudio/jack
**Linux :** Media -> OSS/ALSA/jack
**Windows :** Media -> ASIO (via portaudio)




This part of the menu should list the available audio drivers on your system, and allow you to switch between them. The drivers you have depend on your operating system, and what drivers you have installed on that operating system. Keep in mind you may not have all of these installed on your computer:

**Linux**

- OSS
- ALSA
- jack

**OS X**

- portaudio
- jack

**Windows**

- MMIO
- ASIO

Linux users are encouraged to investigate **JACK (Jack Audio Connection Kit)**, an audio server which allows different audio applications to be connected with virtual "cables" in your computer. JACK, and it's Graphical User Interface **QJackctl**, should be available from whatever Linux distribution you happen to be running.

Many OS X users have also reported that audio runs smoother and with less CPU load when using **JackOSX**, an implementation of the JACK server and user interface for the Mac OS. JackOSX can be found at http://jackosx.com/

And Windows users may find configuring their ASIO soundcards much easier by using **ASIO4ALL**, which can be downloaded from http://www.asio4all.com/

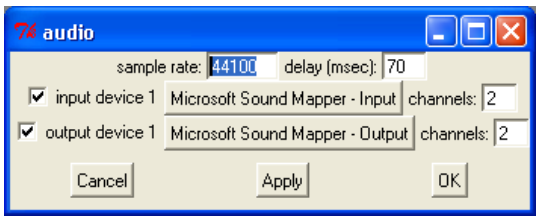
## MIDI drivers (Linux only)

**Linux :** Media -> default-MIDI/ALSA-MIDI

This menu which allows you to switch between the built-in Pd MIDI drivers and the ALSA MIDI drivers, if they are installed. If the ALSA MIDI drivers are used, then JACK users can use the **QJackctl** application (available in most Linux distributions) to connect external MIDI devices and other MIDI applications running on the same computer to Pd.

## Audio Settings

**OSX :** Pd-extended -> Preferences -> Audio Settings
**Linux & Windows :** Media -> Audio Settings




This is one of the most important configuration menus in Pd. Here you can change the **sample rate**, **delay**, **input** and **output** devices as well as the number of **channels** they use.

### Sample rate

The sampling rate for CD quality audio is 44,100 Hz. Most computer soundcards run at this sampling rate, or at 48,000 Hz, by default. Choose the rate that matches the rate of your soundcard or audio drivers here.

### Delay (msec)

Your computer needs a certain amount of time to process all the information coming out of Pd and send it to the soundcard for playback. Likewise, when you are recording, Pd needs a

certain amount of time to gather all the information coming from the soundcard. The term for this delay is called **latency**, and it measures the amount of time between the moment when you tell Pd to do something (for example by playing a note on a keyboard), and when you hear the result of that action. A shorter latency means you will hear the results quicker, giving the impression of a more responsive system which musicians tend to appreciate. However, with a shorter latency you run a greater risk of getting an interruption or 'glitch' in the audio. This is because the computer does not have enough time to "think about" the sound before sending it to the soundcard. A longer latency means less chances of glitches, but at the cost of a slower response time. It is up to you to find the best balance for your own needs, but the default latency in Pd is 50 milliseconds. You can increase or decrease the latency of Pd by entering a value in milliseconds in this box.

### Input Device

Choose the soundcard you wish to use with Pd and the number of channels you want to use. In the case of a normal, stereo soundcard you would enter the number 2. For a multichannel soundcard, you may choose some or all of the channels. Make sure this is checked if you would like to record sound into Pd.

### Output Device

Choose the same soundcard as you selected for the Input Device, and a matching number of channels as you selected for the Input Device as well. Although it may be possible to use different soundcards and unmatched numbers of channels for input and output on some systems, this can also cause problems for Pd, so experiment first. Make sure the checkbox next to the device is checked.

## MIDI Settings

**OSX :** Pd -extended -> Preferences -> MIDI Settings
**Linux & Windows :** Media -> MIDI Settings


On Linux, you have a choice of using the built-in MIDI drivers, or the ALSA-MIDI drivers if they are installed. If you are using the built-in drivers, you should be able to choose which devices to Pd will send and receive MIDI messages with. You may also select "use multiple devices" if you have several applications or devices using MIDI. This method is rather complex, because you must set up the devices by number using your startup flags and you will not be able to change them while Pd is running. Using the ALSA-MIDI drivers is easier to manage, and therefore recommended.

When using the ALSA MIDI drivers on Linux, you can tell Pd the number of In and Out Ports to use here. These are connections which other MIDI applications or devices can use to connect to and from Pd. To connect devices or applications, you can use ALSA MIDI with the JACK audio drivers and the **Qjackctl** if you have them installed. In Qjackctl, you will see a tab for MIDI, and be able to connect the inputs and outputs of MIDI devices and applications by clicking on them.

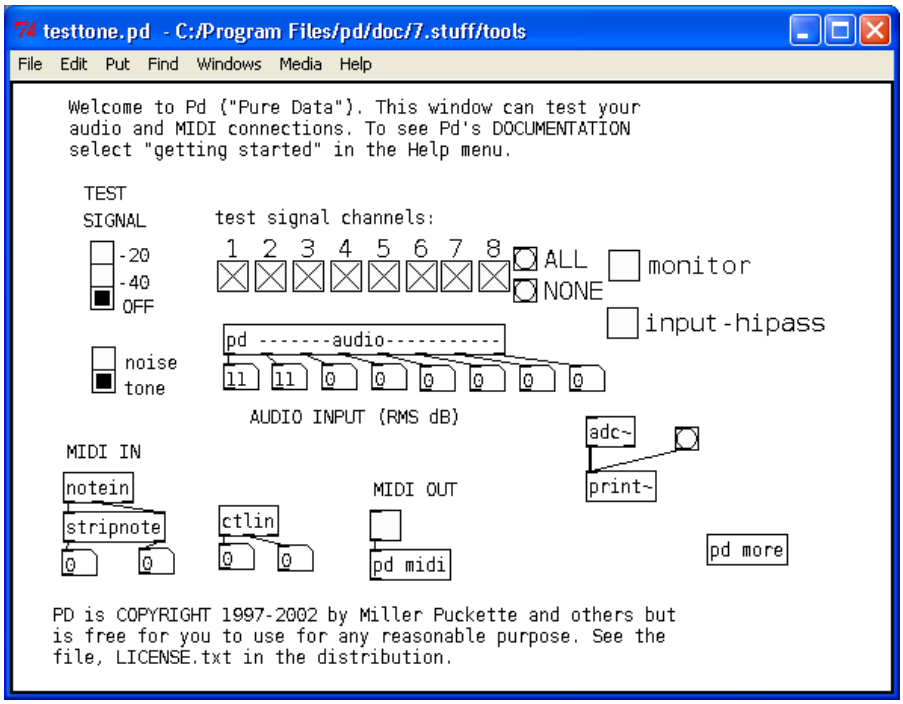
On Mac OS X, to use MIDI you must first open the "Audio MIDI Setup.app", which is located in your

Applications/Utilities folder. Once this application is open, and you have connected your external MIDI devices (if any), you should be able to see your MIDI devices in this window. Minimize the "Audio MIDI Setup.app" and return to Pd and this "MIDI Settings" menu. Now you will be able to choose which devices with which Pd will send and receive MIDI messages. You may also select "use multiple devices" if you have several applications or devices using MIDI.

## Test Audio and MIDI

**OSX, Linux & Windows :** Media -> Test Audio and MIDI

To make sure that you've configured your audio and MIDI correctly, Pd includes a patch to test your setup. If you open "Test Audio and MIDI", you will see this window:




First, click one of the radio buttons marked either "-20" or "-40" under "TEST SIGNAL". If your audio is set up correctly, you will hear a test tone and you will see some of the number boxes above "AUDIO INPUT" changing to measure any incoming audio signal from the line in or microphone of your computer. If you have any external MIDI devices or a piece of MIDI software connected to Pd, you can test the connection by sending MIDI data to Pd and watching to see if the number boxes connected to [notein] and [ctlin] change.

# Advanced Configuration

Since Pd-Extended is installed with most of the settings, search paths and external libraries already configured, many users won't have to worry about configuring these parts of Pure Data at all. Advanced users, however, may be interested in customizing these settings. The settings which can be changed in Pure Data are the same as those available when starting from the command line:

```
audio configuration flags:
  -r <n>           -- specify sample rate
  -audioindev ...  -- audio in devices; e.g., "1,3" for first and third
  -audiooutdev ... -- audio out devices (same)
  -audiodev ...    -- specify input and output together
  -inchannels ...  -- audio input channels (by device, like "2" or "16,8")
  -outchannels ... -- number of audio out channels (same)
  -channels ...    -- specify both input and output channels
  -audiobuf <n>    -- specify size of audio buffer in msec
```

```
  -blocksize <n>   -- specify audio I/O block size in sample frames
  -sleepgrain <n>  -- specify number of milliseconds to sleep when idle
  -nodac           -- suppress audio output
  -noadc           -- suppress audio input
  -noaudio         -- suppress audio input and output (-nosound is synonym)
  -listdev         -- list audio and MIDI devices
  -oss             -- use OSS audio API
  -32bit           ----- allow 32 bit OSS audio (for RME Hammerfall)
  -alsa            -- use ALSA audio API
  -alsaadd <name>  -- add an ALSA device name to list
  -jack            -- use JACK audio API
  -pa              -- use Portaudio API
  -asio            -- use ASIO drivers and API
  -mmio            -- use MMIO drivers and API
MIDI configuration flags:
  -midiindev ...   -- midi in device list; e.g., "1,3" for first and third
  -midioutdev ...  -- midi out device list, same format
  -mididev ...     -- specify -midioutdev and -midiindev together
  -nomidiin        -- suppress MIDI input
  -nomidiout       -- suppress MIDI output
  -nomidi          -- suppress MIDI input and output
  -alsamidi        -- use ALSA midi API
other flags:
  -path <path>     -- add to file search path
  -nostdpath       -- don't search standard ("extra") directory
  -stdpath         -- search standard directory (true by default)
  -helppath <path> -- add to help file search path
  -open <file>     -- open file(s) on startup
  -lib <file>      -- load object library(s)
  -font-size <n>     -- specify default font size in points
  -font-face <name>  -- specify default font
  -font-weight <name>-- specify default font weight (normal or bold)
  -verbose         -- extra printout on startup and when searching for files
  -version         -- don't run Pd; just print out which version it is
  -d <n>           -- specify debug level
  -noloadbang      -- suppress all loadbangs
  -stderr          -- send printout to standard error instead of GUI
  -nogui           -- suppress starting the GUI
  -guiport <n>     -- connect to pre-existing GUI over port <n>
  -guicmd "cmd..." -- start alternatve GUI program (e.g., remote via ssh)
  -send "msg..."   -- send a message at startup, after patches are loaded
  -noprefs         -- suppress loading preferences on startup
  -rt or -realtime -- use real-time priority
  -nrt             -- don't use real-time priority
  -nosleep         -- spin, don't sleep (may lower latency on multi-CPUs)
```

All of the Audio and MIDI configuration flags in this list are set using the menus described above. Note that not all settings are available on all platforms (for example, there are no -asio or -mme options on Mac OS X or Linux, nor the -alsa, -oss, -pa or -jack settings on Windows, etc...)

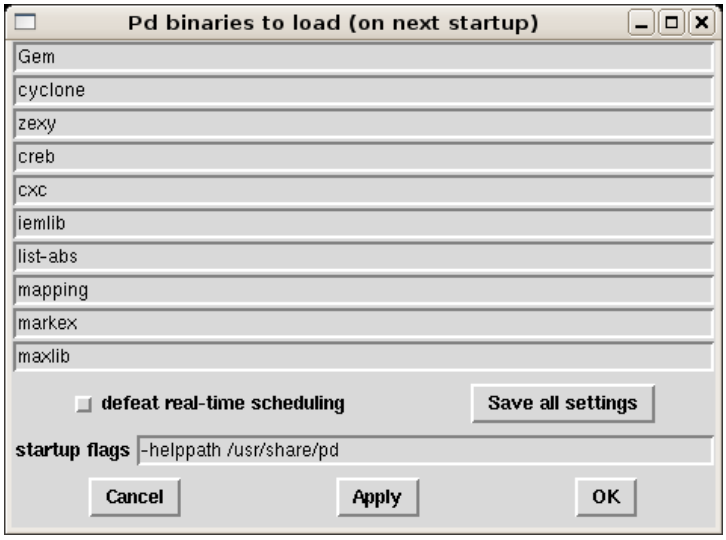
The next most-important configuration options have to do with the external libraries which Pd loads at startup time (and thus which objects you will be able to use), as well as the locations in your file system where Pd can search for these externals and for other resources the program uses to run.

Pure Data uses a system called **pdsettings** to store all these options and use them every time Pd starts up. The pdettings can be configured through various menus in the application, as we saw with the audio and MIDI settings. But they can also be configured by other tools, which are specific to each operating system.

We'll start by looking at the built-in menus for **Startup** and **Path**, and then we'll look at other methods to change the configuration options.

## Startup Flags

**OSX** : Pd-extended -> Preferences -> Startup
**Linux & Windows** : File -> Startup




The things we want to pay attention to in this menu are the externals we load, which are listed as "Pd binaries to load (on next startup)", and whether or not we "defeat real-time scheduling".

Under "Pd binaries to load", you can make a list of the external libraries which you have installed on your system which you would like to be available in Pd. You will then be able to run these externals the next time you start Pd. Because you are using the **Pd-extended** distribution, this section should be completed for you with a list of the externals which come with the distribution.

If you would like to add more libraries to the ones listed, the simplest way is to add them to an existing line of the Startup menu, like so:

```
Gem:my_new_lib
```

And then click "Save all settings" and "OK". However, Pd-Extended is still a program which is under development, and this method has been noted to have some problems lately, so you may wish to try the **Platform-Specific Configuration Tools** below.

If you are running Pd on Linux, you may want to experiment with using "real-time scheduling" to improve the audio quality by allowing Pd faster access to the soundcard. On some systems, however, you must run Pd as the administrator of the system (i.e. "root" or "su") to have permission to do this. To use "real-time scheduling", enter the following in your "startup flags"

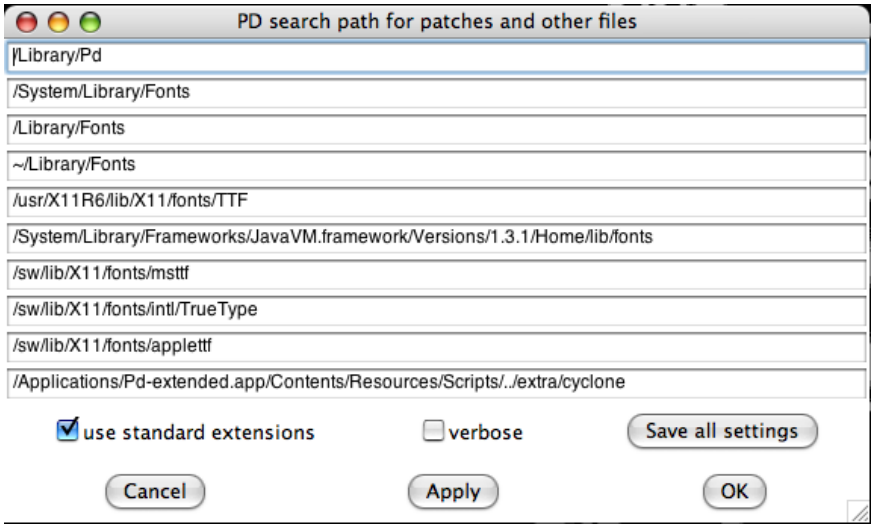
```
-rt
```


But keep in mind that if Pd overloads or locks up your system by using too much of the processer's resources, it can be very difficult to quit the program when using "real-time scheduling".

Users on Mac OS X should **not** use the "real-time scheduling" flag, and should click the box which says "defeat real-time scheduling" for better audio quality.

## Path

**OSX** : Pd-extended -> Preferences -> Path
**Linux & Windows** : File -> Path




Shown here is the Mac OS X menu for setting the Paths. These are the **Search Paths** that Pd will use to locate external libraries, help patches, and other any patches, fonts, soundfiles, videos ar anything else which you may need while working in the program. If you would like to add more directories to the ones listed, the simplest way is to add them to an existing line of the Path menu, like this:

```
/Library/Pd:/home/my_name/my_new_path
```

And then click "Save all settings" and "OK". However, as with the Startup menu, some people have had problems using this method, so you may wish to try the **Platform-Specific Configuration Tools** below.

Quite a bit of this configuration has been taken care of by Pd-Extended already, so let's look at some real-world examples of when you might want to add a path. One situation would be if you want to use an audio file or a collection of audio files in your patch, but you don't want to have to specify the whole location every time it's needed in any object or message.

So, instead of typing

```
/home/beaver/my_soundfiles/spoken/boy/geewhiz.wav
```

or

```
/home/beaver/my_soundfiles/spoken/girl/golly.wav
```

you could add

```
/home/beaver/my_soundfiles/spoken
```

to your Path, and then call these soundfiles by typing:

```
boy/geewhiz.wav
girl/golly.wav
```

Another very common situation is when you would like to use a Pd patch you have saved as an **abstraction** (which essentially treats the saved patch like another Pd object) inside another Pd patch. In this case, you must either have the patch you wish to use as an abstraction saved in the folder as the "parent" patch you wish use it

in, or you must add the folder containing the abstraction to your Path. For example the path:

```
/home/pdfreek/puredata/abstractions/reverb_tools
```

might contain various kinds of reverb abstractions that the user "pdfreek" created to be reused in other patches. For more information about abstractions, please see the **DataFlow Tutorials** chapter.

Finally, if you want to compile your own external Pd libraries, or use ones which you have downloaded from the internet, then you need to place the binary files (which end in **.pd_linux** for Linux, **.pd_darwin** for OS X and **.dll** for Windows) in a folder and add that folder to your path, such as:

```
~/pd/extra
```

where **~/** means your home directory (i.e. */home/"username"* on Linux and */User/"username"* on Mac OS X). Please note that in the case of **name clashes** (where two objects or files have the same name), the one which is loaded last takes precedence over all others. An example of this is the object [counter], which exists in several external libraries, and which has a different function in each one!

## Platform-Specific Configuration Tools

The locations for the pdsettings files in Pd are:

**OS X:** *~/Library/Preferences/org.puredata.pd.plist* (~ means your home folder)
**Windows:** *HKEY_LOCAL_MACHINE -> SOFTWARE -> Pd* (using REGEDIT.EXE/REGEDIT32.EXE)
**Linux**: *~/.pdsettings* (~ means your home folder)

**Linux**

Linux users may edit the file directly via command line applications such as **joe**, **vim**, **pico** or **nano**, or with whatever other text editing application comes with your distribution:

```
$ nano /home/derek/.pdsettings

GNU nano 1.2.4                      File: /home/derek/.pdsettings

audioapi: 5
noaudioin: False
audioindev1: 0 4
noaudioout: False
audiooutdev1: 0 4
audiobuf: 50
rate: 44100
nomidiin: False
midiindev1: 0
nomidiout: False
midioutdev1: 0
path1: /home/derek/pd/rradical/memento
path2: /home/derek/pd/ix_toxy
path3: /home/derek/pd/berlin
path4: /home/derek/pd/rradical/memento/tutorial
path5: /home/derek/workshop_patches
path6: /usr/local/lib/pd/doc/5.reference
path7: /usr/local/lib/pd/extra/xjimmies
npath: 7
standardpath: 1
verbose: 0
loadlib1: pool
loadlib2: iemlib1
loadlib3: iemlib2
loadlib4: iem_mp3
```

```
loadlib5: iem_t3_lib
loadlib6: OSC
loadlib7: zexy
nloadlib: 7
defeatrt: 0
flags: -alsamidi -rt

                                        [ Read 31 lines ]
^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Txt     ^T To Spell
```

Remember that if you add a new *path* or *loadlib*, then you will need to give it a number higher than the last existing one, and you will need to change the *npath* or *nloadlib* to the number of new paths or loadlibs you have added. In the above *pdsettings*, to add the loadlib **pdp**, you would have to add/change the following:

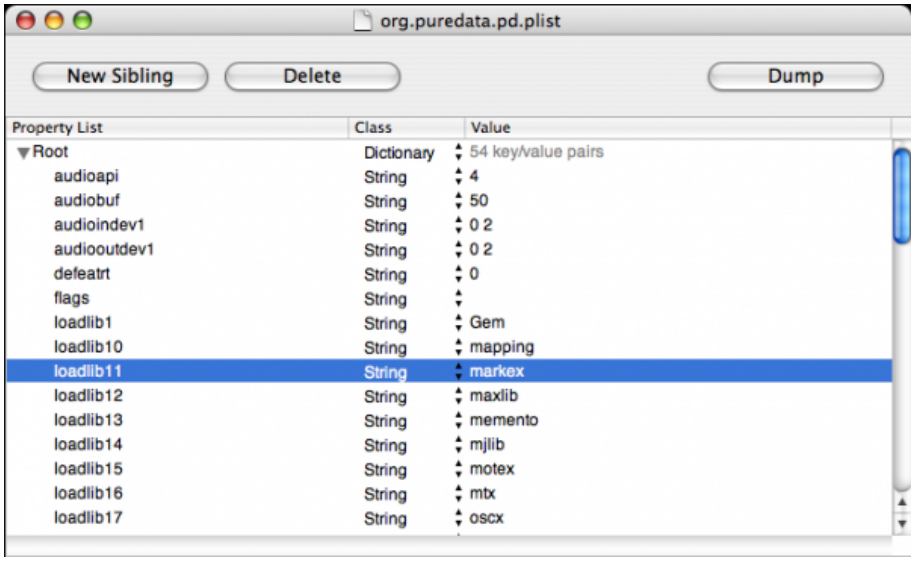
```
loadlib8: pdp
nloadlib: 8
```

### OS X

OS X users may wish to try using the **Property List Editor.app**, which can be installed from the **XCode Tools** or **Server Tools** CDs available for free from Apple:

http://developer.apple.com/tools/xcode/

Here is the Property List Editor, with the *org.puredata.pd.plist* file open:




You can click directly in the *Value* field to change a value, or use the *New Sibling* button to add a new line.

The command line utility **defaults** can also be used. The following line in the terminal lists all the *pdsettings* in *org.puredata.pd.plist*:

```
defaults read org.puredata.pd
```

The following command can be used to write a new line to pdsettings:

```
defaults write org.puredata.pd loadlib30 test
```

and this command can be used to delete one line from pdsettings:

```
defaults delete org.puredata.pd loadlib30
```

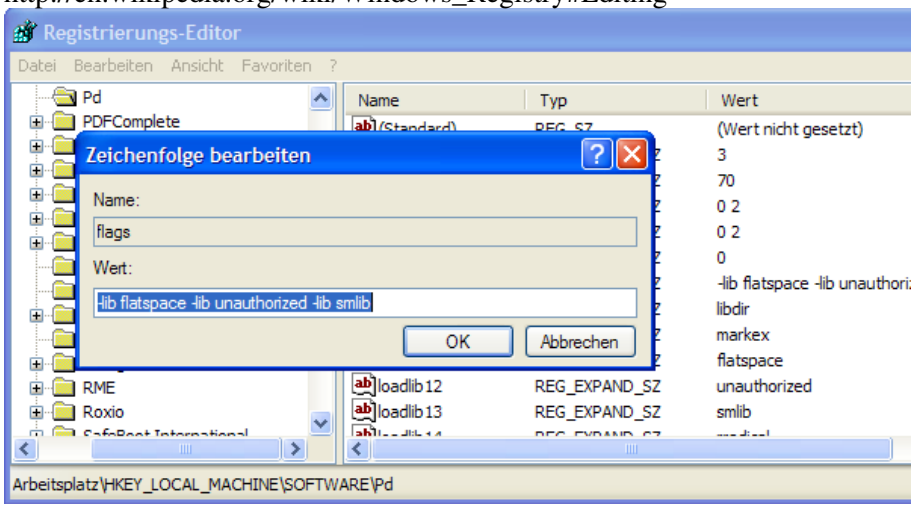
In this case, *loadlib30* represents the next possible line that could be added to load a library (29 libraries are loaded already), and *test* represents a hypothetical library which we add to the startup in the first case using the *write* command, and remove from the startup in the second case by using the *delete* command. For more information about defaults, type:

```
defaults --help
```

**Windows**

Windows users may also use the **REGEDIT** program to edit their *pdsettings*. This program comes with the Windows operating system, and can be located under the name REGEDIT.EXE or REGEDT32.EXE (Windows XP or newer). Please note: manually editing the Windows Registry files using a text editor instead of REGEDIT is generally considered unsafe, since errors here can disrupt the entire operating system! Those interested in more details about the Registry should read:

http://en.wikipedia.org/wiki/Windows_Registry#Editing

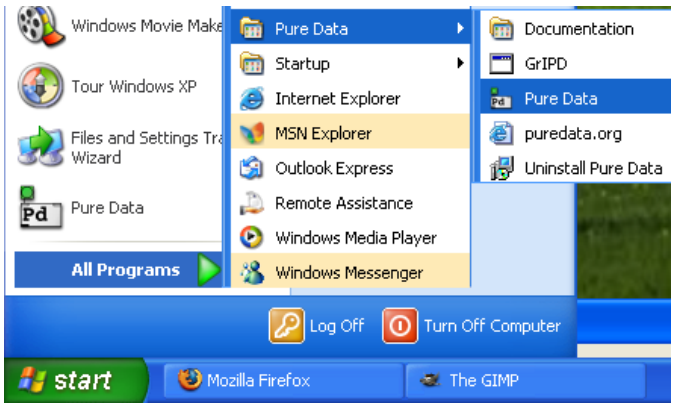


# Starting Pure Data

Now that you have Pd-Extended installed on your computer and configured, let's look at different ways to start it--from simply clicking an icon through starting from the command line and adding different startup flags or using a script to save different sets of startup information.

## Starting PD with the Clickable Icon

There are two ways of starting Pure Data. The way that will be used most commonly on Windows or Mac OS X will be to click on the icon which the installer put in your "My Programs" or "Applications" folder. On Windows, this is "Start -> Pure Data -> Pure Data".




On Linux, your system may also have a menu bar, such as "Programs/Multimedia" or "Programs/Sound" where Pd can be started by clicking the menu item.

## Starting Pd via Command Line

The other way is to open Pd from the terminal or shell via a command line. This is most often done on Linux, but it can be done this way on any platform. To do this, one must know the location of the Pd application on his/her system, which can be different depending on where Pd was installed.

### Linux (from xterm)

```
/usr/local/bin/pd
```

### Mac OSX (from Terminal.app)

```
/Applications/Pd-extended.app/Contents/Resources/bin/pd
```

### Windows (from the DOS shell or Command Prompt)

```
C:\Program Files\pd\bin\pd.exe
```

Why would we want to open Pd by command line? The most common reason would be is if we wanted to use a different set of flags than the default ones. For example, if you were using Pd in a live performance, and you wanted it to open up the same patch whenever you started it in this situation, you might use the command:

```
/usr/local/bin/pd -open /home/pdfreek/liveset.pd
```

Which would start Pd and open the patch *liveset.pd*. You could also add other startup **flags**, such as which soundcard and drivers to use, which external libraries to load or which search paths to add. Flags are additional pieces of information which can alter the configuration of Pd for that particular startup, rather than the *pdsettings* which we looked at in the **ConfiguringPD** chapter, which affect the program every time it starts.

Like almost any program launched by command line, you can add the flag "--help" to see a long list of configuration options, which gives you some idea of the different possibilities for starting up Pd:

```
$ /Applications/Pd-0.39.2-extended-test4.app/Contents/Resources/bin/pd --help
usage: pd [-flags] [file]...
audio configuration flags:
-r <n>           -- specify sample rate
-audioindev ...  -- audio in devices; e.g., "1,3" for first and third
-audiooutdev ... -- audio out devices (same)
-audiodev ...    -- specify input and output together
-inchannels ...  -- audio input channels (by device, like "2" or "16,8")
-outchannels ... -- number of audio out channels (same)
-channels ...    -- specify both input and output channels
-audiobuf <n>    -- specify size of audio buffer in msec
-blocksize <n>   -- specify audio I/O block size in sample frames
-sleepgrain <n>  -- specify number of milliseconds to sleep when idle
-nodac           -- suppress audio output
-noadc           -- suppress audio input
-noaudio         -- suppress audio input and output (-nosound is synonym)
-listdev         -- list audio and MIDI devices
-jack            -- use JACK audio API
-pa              -- use Portaudio API
      (default audio API for this platform:  portaudio)
MIDI configuration flags:
-midiindev ...   -- midi in device list; e.g., "1,3" for first and third
-midioutdev ...  -- midi out device list, same format
-mididev ...     -- specify -midioutdev and -midiindev together
-nomidiin        -- suppress MIDI input
-nomidiout       -- suppress MIDI output
-nomidi          -- suppress MIDI input and output
other flags:
-path <path>     -- add to file search path
-nostdpath       -- don't search standard ("extra") directory
-stdpath         -- search standard directory (true by default)
-helppath <path> -- add to help file search path
-open <file>     -- open file(s) on startup
-lib <file>      -- load object library(s)
-font <n>        -- specify default font size in points
-typeface <name> -- specify default font (default: courier)
-verbose         -- extra printout on startup and when searching for files
-version         -- don't run Pd; just print out which version it is
-d <n>           -- specify debug level
-noloadbang      -- suppress all loadbangs
-stderr          -- send printout to standard error instead of GUI
-nogui           -- suppress starting the GUI
-guiport <n>     -- connect to pre-existing GUI over port <n>
-guicmd "cmd..." -- start alternatve GUI program (e.g., remote via ssh)
-send "msg..."   -- send a message at startup, after patches are loaded
-rt or -realtime -- use real-time priority
-nrt             -- don't use real-time priority
```

To learn more about Pd's startup options, please see the **ConfiguringPD** chapter.

# Starting Pd from a Script

Once you have created a command line for your specific situation, you can save that command as a **script**, which is a short file containing a list of commands, which can be run by typing its name in the terminal or

shell. The exact format of your script depends on which operating system you use.

### Windows

Windows uses the DOS language for its commands, so we must create a .**bat** (DOS batch) file containing the location of the Pd program and the startup flags we want to use. Using a simple text editor, make a file named "*pdstart.bat*", and place the following in it, for example

```
"c:\pd\bin\pd.exe" -font 10 -path "c:\pd\doc\vasp" -lib cyclone -lib iem_t3_lib -lib iem_mp3 -lib
```

Though it may appear to be many lines, this command must in fact be one long line with no breaks. If the version of Windows you are running has a "Save as type" option, choose the type "All files" to prevent your .bat file from being saved as a text file. Once this is saved, you can double-click on the file to run it.

### Linux and OS X

Since both Linux and OS X use the same Unix-type system to interpret and run command lines, the process for creating a script is the same for both. In your favorite text editor, create a new file and start it with the line:

```
#! /bin/bash
```

which tells the operating system that what it is reading is a script, and that it will use the **bash** command line interpreter. On the line below that, copy this or a similar line:

```
/usr/local/lib/pd -font 10 -path /home/pdfreek/pd/my_abstractions -lib cyclone -lib iem_t3_lib -l
```

This should be all in one line, with no breaks. Please note that you should give it the correct path to the Pd program in the beginning (which could be different if you are running OS X for example), and you should replace the example flags with ones of your own.

Once you have written and saved this file with the **.sh** (shell script) file extension, such as "*start_pd.sh*", you must make it **executable** as a script with the following command:

```
chmod +x start_pd.sh
```

After you have done this, you can start this script, which will run Pd with all the flags you have added to it, by typing:

```
sh start_pd.sh
```

Some Linux window managers such as KDE or Gnome may support double-clicking to start shell scripts either by default or by selecting the default application. On OS X, you could configure the Finder to open .sh files with the Terminal.app by default (but then you would have to manually chose to open them with TextEdit.app for editing later on).

## Advanced Scripting for Starting Pd

One of the beautiful things about the Unix system, which both Linux and OS X are based on, is that it is designed to allow many applications to communicate with each other and work together. This means that shell scripts can be constructed to do an enormous amount of tasks.

For example, the following script for Linux starts the JACK audio server (with some flags of its own), opens the Qjackctl interface for JACK and then starts Pd with the *-jack* flag and the *-open* flag listing two specific files:

```
#! /bin/bash

jackd -d alsa -d hw -r 44100 -p 1024 -s &
/usr/bin/qjackctl & sleep 5 ; /usr/local/bin/pd -jack -open /home/derek/pd/delnet/delaynet.pd:/ho
```

The **ampersand** (&) between the commands means that the command preceeding it will be run in the background. In other words, the previosu command will keep running while we execute the next ones, instead of quitting. The section "**sleep 5**" tells the shell to wait 5 seconds before running the next command, in this case in order to give JACK time to start up. The **semicolon** (;) is used to seperate jobs, meaning that the next command won't be run until the previous one is finished (in the case of "sleep 5") or sent to the background (in the case of the ampersand symbol).

This script could be expanded to open other applications (in the following case, the looping application **SooperLooper**), use the **aconnect** application to make ALSA MIDI connections from Pd to SooperLooper, and use the **jack_connect** command to make audio connections between Pd, SooperLooper and 6 channels of a sound card via the JACK audio server:

```
#! /bin/bash

jackd -d alsa -d hw -r 44100 -p 1024 -s &
/usr/bin/qjackctl & sleep 5 ; /usr/local/bin/pd -jack -open /home/derek/pd/delnet/delaynet.pd:/ho
```

Detailed syntax for *aconnect* and *jack_connect* can be found by typing:

```
aconnect --help
```

or

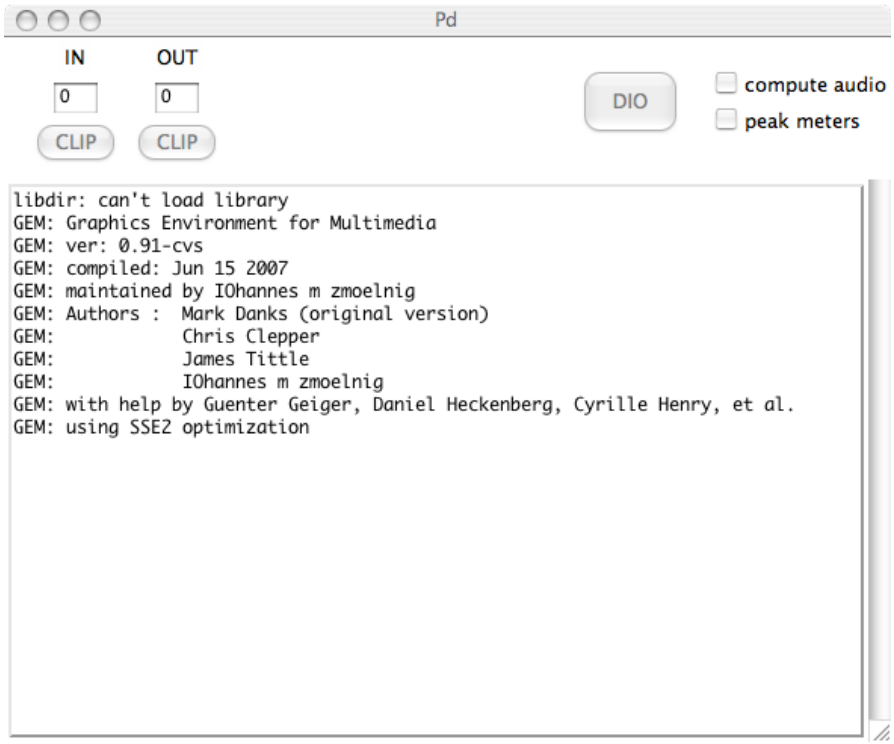
```
jack_connect --help
```

Bash shell scripting is a huge area to investigate, curious readers are encouraged to check out one of the many websites and books detailing the Bash environment.

# The Interface

## The main PD window

Now that we have PD configured and your audio and MIDI are working, let's have a look at the rest of the main PD window.




As of PD 0.39, all of the messages that PD produces are sent to the main PD window (before this, they were sent to the shell which was running PD). When you start PD, this main PD window should tell you important information, such as the externals you are loading and whether any errors occurred while loading them, as well as any errors connecting to the soundcard. Later, you will also use this main PD window to see information about the patch you are working on, as well as for debugging (correcting errors in your patch). So keep this window in a place where you can find it on your screen.

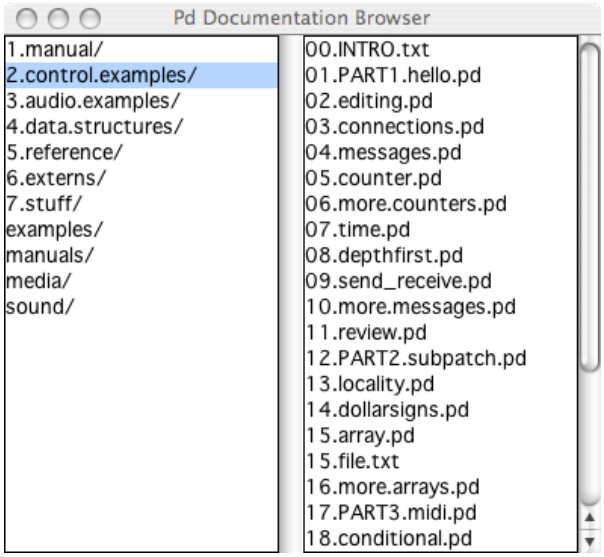
There are a few other important features about this main PD window. It has audio level indicators, so you can can a general idea of the loudness of the sound that you are sending to the soundcard. If this level goes to 100 or higher, you are sending to high a level and you will hear a distorted sound. The boxes marked "Clip" will also flash red. To use the audio level meters, check the box that says "peak meters" in the main PD window.

There is also a box marked "compute audio", which you can use to turn on and off audio processing. When you open the "Test Audio and MIDI" patch, PD will automatically turn audio processing on for you.

Last is a box marked "DIO". This stands for Digital In Out errors, and this box should flash red when PD has difficulties sending data to your sound card. If you click this box, PD will print a list of times when these DIO errors occurred in the main PD window.

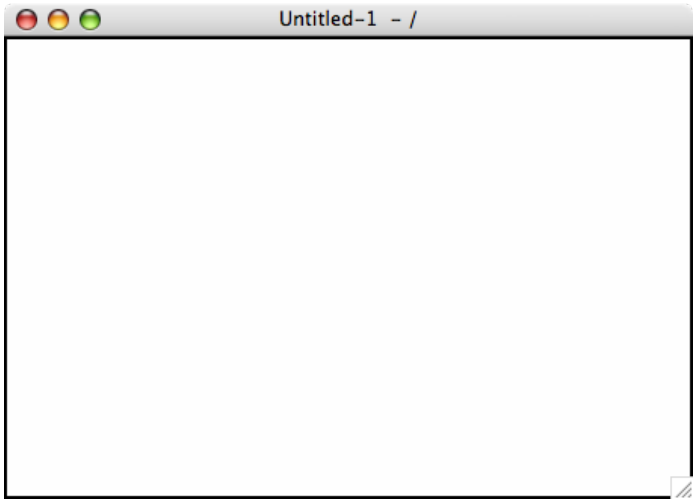
The last thing to pay attention to is the "Help" menu. Under this drop-down menu, you can open the official PD manual, written by Miller S. Puckette in "HTML" format, which can be viewed in your web browser. You can also open a file "Browser", which will list the built-in help patches which come with PD. All of these documents are valuable resources, however many newcomers to PD can find them confusing. We will cover

some of these basics in the "Dataflow", "Audio" and "Patching Strategies" tutorials in this manual, after which you can return to the built-in help files with a bit better understanding.




## Starting a New Patch

Under the "File" menu in the main PD window, create a "New" PD patch. It should look something like this:




Unlike other software for creating audio or video media, such as **Ableton Live**, **CuBase** or **Final Cut Pro**, where a new file shows you a variety of buttons, menus and timelines, PD gives you a blank, white space. Within that white space, you can make a synthesizer or video mixer, translate sensor input into the movements of a robot or stream movies to the internet, for example. The difference between PD and software like Live is that it doesn't start with any preconceived ideas about **how** to make your artwork. Where Live provides you with a set of tools suited primarily for the production of loop-driven dance music, PD acts more like a text editor where anything is possible, so long as you know how to write it. It is this kind of possibility and freedom that attracts many artists to using PD.

To explore these possibilities, you must understand PD as being a written language like German or Chinese. As in any language, PD has a vocabulary (the words used in the language) and a grammar (the way to put these words together so that they make sense). And like learning any language, you first have to learn how to say simple things like "What is your name?" before you can write poetry! So let's start simple.

You will notice that once we have opened a new PD patch, there are a few new menu items to choose from. The "Edit" menu has all the kinds of functions you would expect from a text editor like **Notepad**, **TextEdit**, **OpenOffice** or **Word**, such as "Cut", "Paste", "Duplicate", "Select All", etc etc.

There is also a "Put" menu, containing a list of the kinds of things you will be putting in your patch, such as "Object", "Message", "Number", "Symbol", "Comment" and a range of GUI (Graphical User Interface) elements such as "Bang", "Toggle", "Slider", etc.

# Interface Differences in Pure Data

While the main functionality of Pure Data doesn't change between operating systems, the locations and contents of some of the menus do. Depending on the system you are running, you will be able to do the following:

## Linux

From the "File" menu, you can:
1) Create a "New" PD patch
2) "Open" a PD patch which is saved on your computer
3) Send a "Message" to the running PD application
4) Set the search "Path" which PD uses
5) Change the "Startup" flags which PD uses
6) "Quit" PD

From the "Find" menu, you can:
1) "Find last error" which occurred in the program

From the "Windows" menu, you can:
Change between the different open PD patches

From the "Media" menu, you can:
1) Turn audio "ON" and "OFF"
2) Change between the different available audio drivers
3) Change between the different available MIDI drivers
4) Change the "Audio Settings"
5) Change the "MIDI Settings"
6) "Test Audio and MIDI"
7) View the CPU "Load Meter"

And from the "Help" menu, you can:
1) Read information "About PD"
3) Open a "Browser" to see some help patches which are included in PD

## Mac OS X

From the "Pd" menu (which should contain the version number as well), you can:
1) Read information "About PD"

2) Change the following "Preferences":
A) Set the search "Path" which PD uses
B) Change the "Startup" flags which PD uses
C) Change the "Audio Settings"
D) Change the "MIDI Settings"
3) "Quit" PD

From the "File" menu, you can:
1) Create a "New" PD patch
2) "Open" a PD patch which is saved on your computer
3) Send a "Message" to the running PD application
4) "Quit" PD

From the "Find" menu, you can:
1) "Find last error" which occurred in the program

From the "Media" menu, you can:
1) Turn audio "ON" and "OFF"
2) Change the "Audio Settings"
3) Change the "MIDI Settings"
4) "Test Audio and MIDI"
5) View the CPU "Load Meter

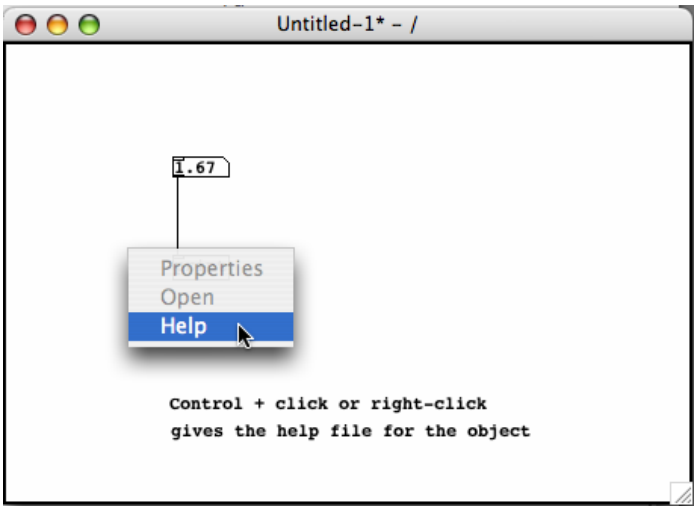
From the "Windows" menu, you can:
Change between the different open PD patches

And from the "Help" menu, you can:
1) View the author's documentation as an HTML file
2) Open a "Browser" to see some help patches which are included in PD

# Placing, Connecting and Moving Objects in the Patch


Use the "Put" menu to place an "Object" in your patch. Click on the patch to drop the object in its place. You will see a box made of a broken blue line, with a flashing cursor inside indicating that you should type something there.

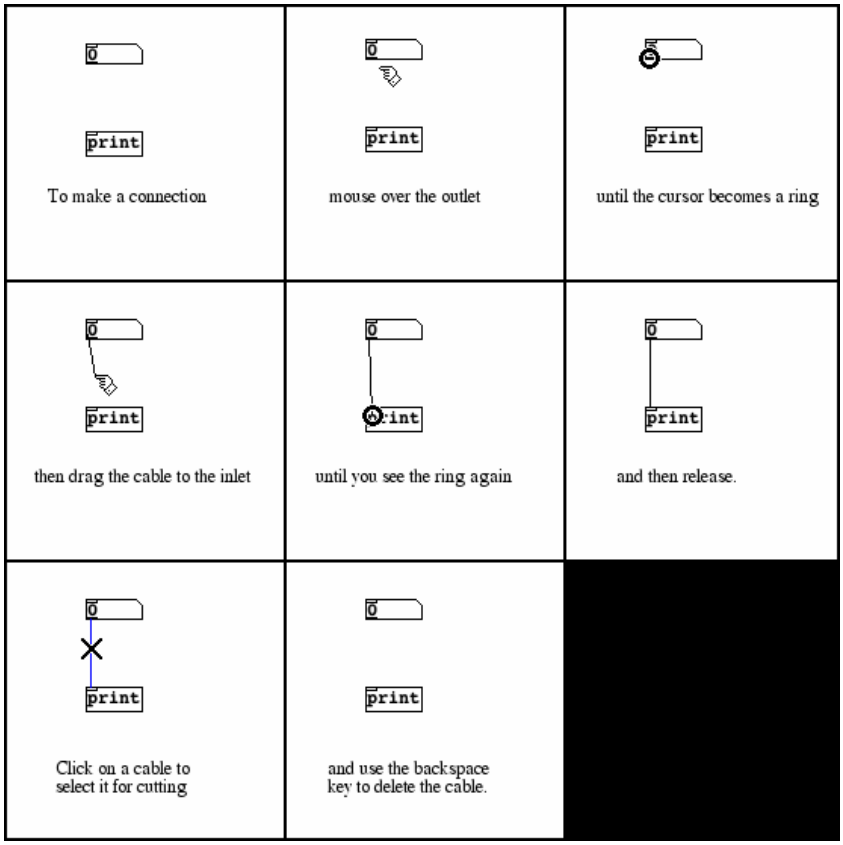
Objects are the "vocabulary" of PD. The more names of objects you know, the more complicated things you can do with PD. If you type the word "print" inside this object and click again outside the box, you will create the [print] object. If you right-click (or use the Control key and click on OS X), you will have the option to open the help file for that object. This is something like the "dictionary entry" for the object, and should define what it does and also show several examples of its use.



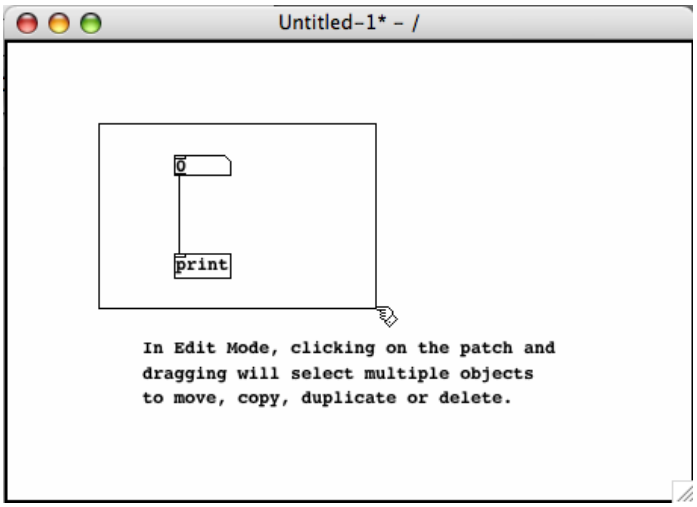

Return to the "Put" menu, and this time place a "Number" in your patch. Notice that the shape of the number box is different from the shape of the object box.

You should also notice that both the object and the number boxes have small rectangles at the corners. If these are at the top of the object, they are called "inlets", and at the bottom they are called "outlets". When you are working on your patch, your cursor is shaped like a pointing finger. If you put that finger over an outlet, it changes into a black circle which indicates that the outlet is selected.

Select the outlet of the the number box, click and drag that black circle until it reaches the inlet at the top of the [print] object. When you have done that, you will see the cursor change from the pointing finger to the black circle again. If let go of the mouse button now, you will make a connection from the outlet of the number box to the inlet of [print]. If you want to remove this connection, place your cursor over the connection until you see a black X and then click. The connection will turn blue and you can remove it with the Backspace or Delete key on your keyboard.




If you click on the patch away from the number box and [print] object and drag, you can draw a box which selects them. You will see they are selected because they will turn blue. Single objects can be selected by clicking once on them.




Once the objects on screen are selected, you can:

- Move them by dragging them with the mouse
- Move them in small increments with the Arrow keys
- Move them in larger increments with the Shift and Arrow keys
- Delete them with the Backspace or Delete keys
- Copy them by using the Control and C keys (Apple and C keys on OS X) or the Copy menu item under Edit
- Cut them by using the Control and X keys (Apple and X keys on OS X) or the Cut menu item under Edit

- Once Cut or Copied, you can Paste them with the Control and V keys (Apple and V keys on OS X) or the Paste menu item under Edit
- You can also Duplicate the selected items with the Control and D keys (Apple and D keys on OS X) or the Duplicate menu item under Edit

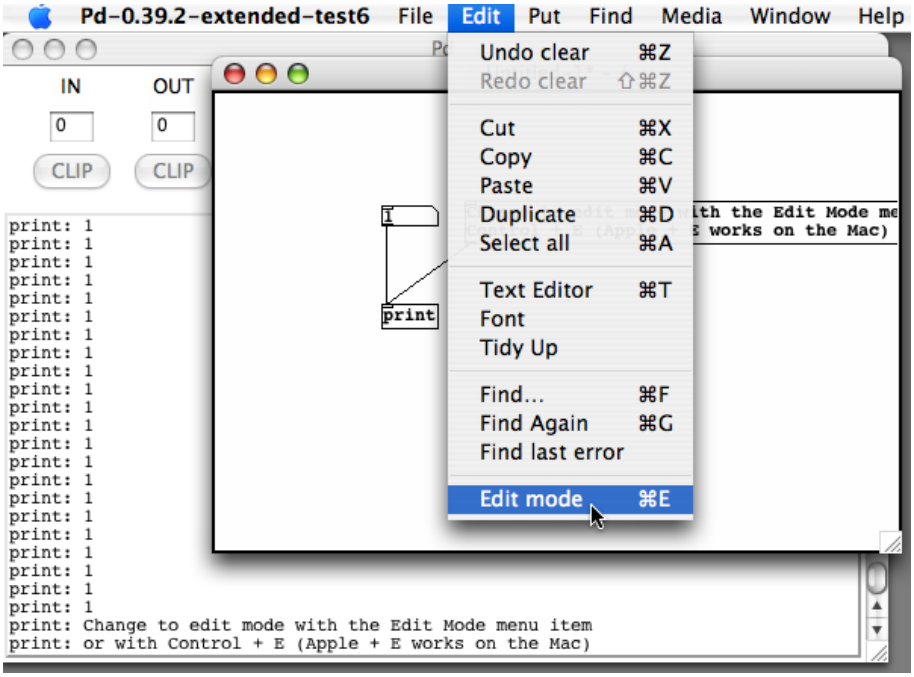
It is recommended to use the duplicate function rather than the paste function, because pasted objects are placed directly on top of the previous object, making it difficult to see them. Duplicated objects are placed to the lower right side of the original, making them easier to find and move.


Pasted or duplicated objects are automatically selected together, so you can grab ahold of them and move them immediately after placing them in the patch.

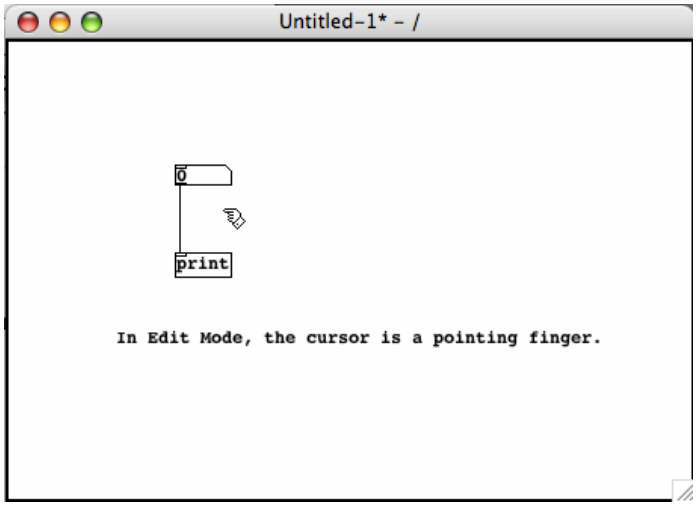
# Edit Mode and Play Mode

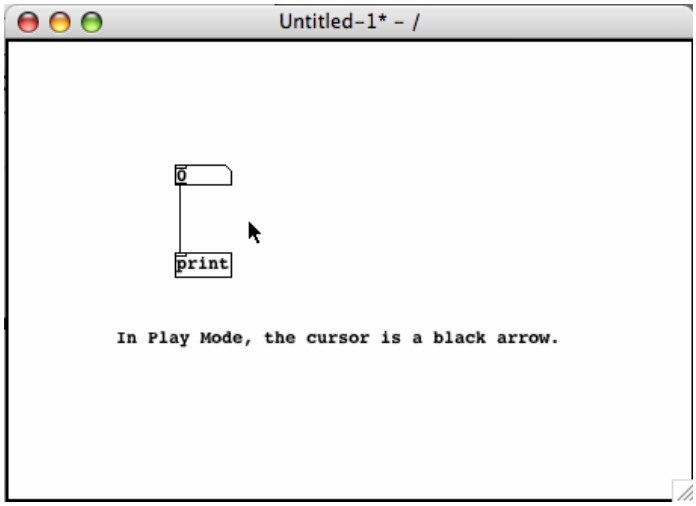
So far we've been able to put objects in the patch, connect them, move them around or delete them. But how does one get some results from this patch? In this case, we have connected a number box to a [print] object, which should print the numbers we send to it in the main PD window.

To make this happen, we need to change out of "Edit Mode" and into "Play Mode". You can do this by clicking on the "Edit Mode" item in the Edit menu, or by using the Control and E keys (Apple and E keys on OS X).

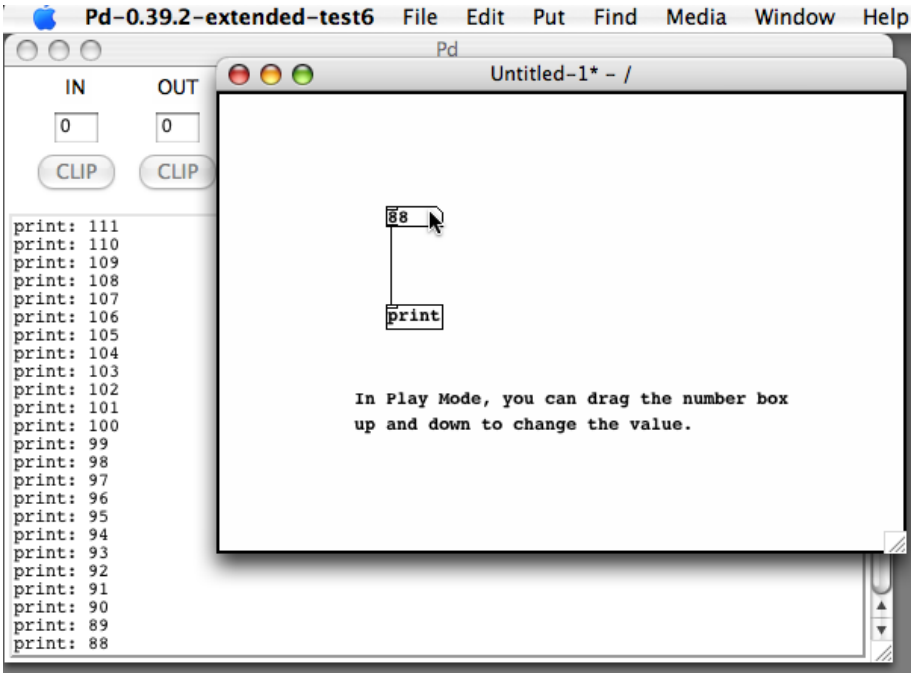



When you do this, you will see that the pointing finger cursor changes into an arrow cursor.

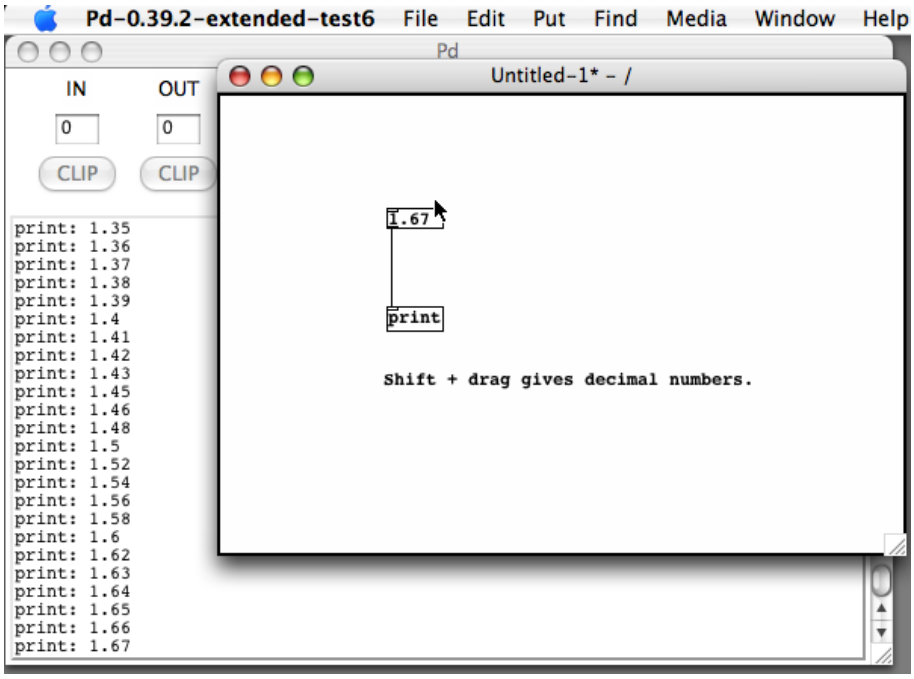



If you click and drag inside the Number object now, you can change the numbers inside of it. Any changed number is sent to the outlet, which then goes on to the inlet of the [print] object, and the number is printed to the main PD window.



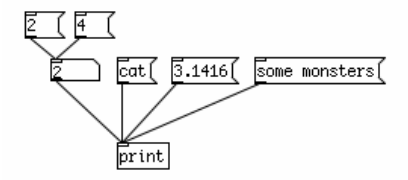
If you click once on the number box in Play Mode, you can also use your keyboard to change the value, and the Enter key to send the value to the outlet. If you hold the Shift key while using the mouse to change the number, you will have decimal numbers. Using the Alt key plus a mouseclick will toggle the Number box between 0 and 1.




If you would like to make any changes to this patch, you can use the "Edit Mode" menu item, or the key combination Control (or Apple) and E to change back and forth between Edit and Play modes. Note that you are automatically placed in Edit Mode whenever you add any new item from the "Put" menu to your patch.

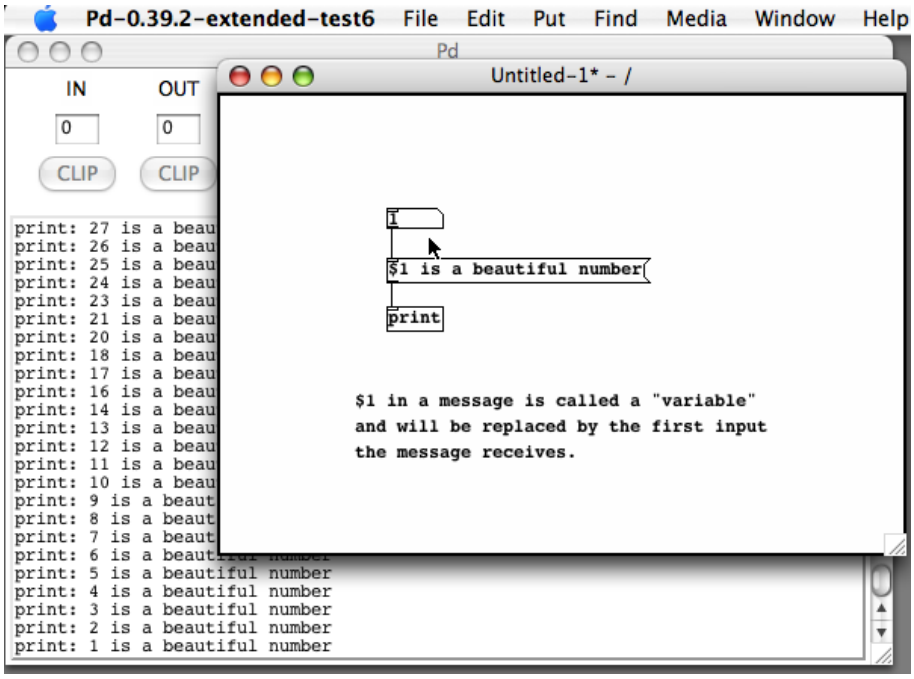
# Messages, Symbols and Comments

The "Message" box is used to store and send information to other objects, and can contain numbers or text. It also has a unique shape, which resembles an envelope like you would use to send a letter. Place two different messages above the number box in our exercise. Like the object, messages also give a flashing cursor indicating that you should enter some information when you create them. Enter "2" in one of the messages and "4" in the other, and connect both to your number box. Switch to Play Mode and click on each of the messages. When you do, you will see that the number box changes according to the message that you send it, and that the message is also sent onwards to the [print] object.



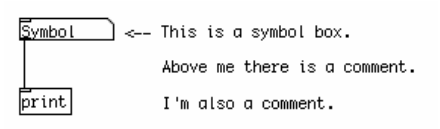

You can also send numbers and other information to the message box. Create a message with the text "$1 is a beautiful number", and connect it to the [print] object. Then connect a Number to the inlet of the message, and in Play Mode change the value of the number. You will see in the main PD window that whatever number you

send to this message replaces the $1. This is because $1 is a "variable", and will take the value of whatever you send to it. This is important because different objects need to be sent different messages in order to do things. We will look at more uses for messages and variables later in the Dataflow Tutorial.



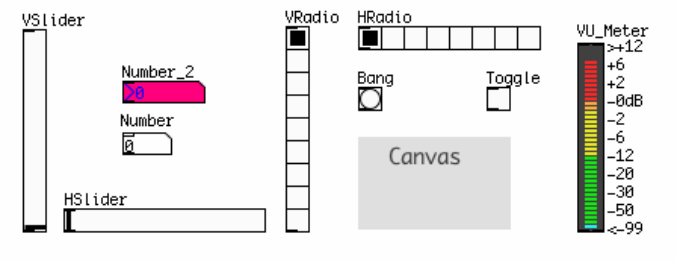

A "symbol" is another way of storing and sending information. Once created, you can use it to display the output of some objects, or you can type directly into it and hit Enter to send the text out. Please note that no spaces will appear in the symbol box when you type into it, since separate words would be considered separate symbols.

A "comment" is simply a way of making a note to yourself so that you (or someone else) can understand what you were trying to do later on. You can make as few or as many as you want, and they have no effect on the patch itself.




# GUI Objects

PD has a number of GUI objects you can use to graphically control your patch and to improve its visual appearance. These are:




1. **Bang:** this GUI object sends a Message named "Bang" every time it is clicked. "Bang" is a special message, which many Objects interpret as "do an action right now!". Using the Bang GUI object is the same as creating a Message box with the word Bang in it. The Bang GUI object can also be used to receive and display Bang messages. For more information on this, see the "Counter" chapter in the Dataflow Tutorial.

2. **Toggle:** when clicked, the Toggle sends out one of two values--a zero when it is unchecked and a non-zero number when it is checked. The non-zero number is 1 by default, however this can be changed in the "Properties". The Toggle also has an inlet, which can be used to display whether an incoming number is zero or not.

3. **Number2:** this is almost identical to the Number box, however it has further options in its "Properties", including the ability to save its current value when the patch is saved (by changing the "no init" box to "init"). The Number2 has in anlet which can be used to display incoming numbers as well.

4. **Vslider and Hslider:** these are Vertical and Horizontal sliders which send out their current value when moved with the mouse. The default range of a slider is 0-127, which can be changed in the "Properties". Both sliders have an inlet which can be used to display incoming numbers within the range of the slider.

5. **Vradio and Hradio:** these are Vertical and Horizonal "radio buttons", which send out their current value when one of the buttons in them is clicked with the mouse. The default size of a radio button is 8 buttons, which can be changed in the "Properties". Both radio buttons have an inlet, which can be used to display integer (whole) numbers within the range of the radio buttons.

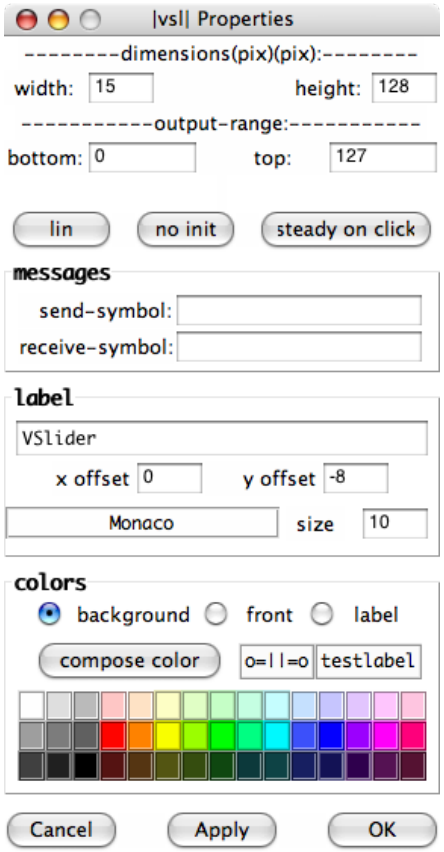
6. **VU:** a VU meter displays the average volume level of any audio signal which is connected to it in Decibels. You may switch the value scale on the right side on and off in the "Properties".

7. **Canvas:** a canvas is a rectangular area of pixels, whose size and color may be changed under its "Properties". Canvases are useful as backgrounds in your patch to improve its visual appearance and readability. Canvas also can be used as movable GUI objects that gather information about their position (x,y) inside a patcher. Keep in mind that PD remembers the order in which anything is placed in the patch, so if you want your canvas to be behind certain objects, you must either create it first, or you must Select, Cut and Paste the objects you want in the foreground so that they appear in front of the canvas.

## GUI Object Properties

If you right-click (or Control and click on OS X) on any GUI object, you will see the "Properties" menu. Here, you can change many aspects of each GUI object, such as its default values, size in pixels or its color.

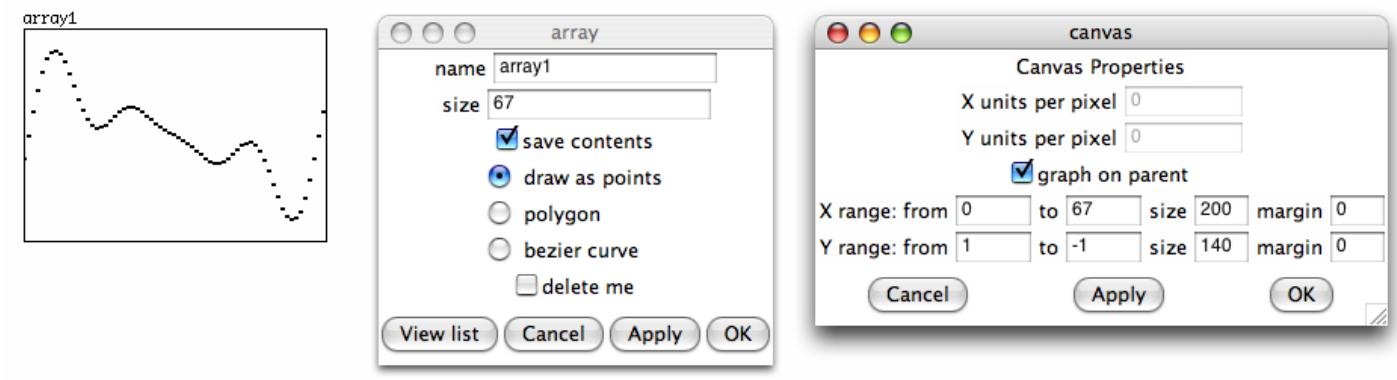
To change colors on Linux and Windows you should see a selection of available colors. On OS X these boxes are empty, so you must click on the "Compose Color" button. You can also add a label to your GUI object as well as set the Send and Receive symbols. For more information on Send and Receive, please see the Send/Receive chapter of the Patching Strategies tutorial.




# Arrays and graphs

An "array" is a way of graphically saving and manipulating numbers. It works in an X/Y format, meaning you can ask the table for a value by sending it a value representing a location on the X (horizontal) axis, and it will return the value of that position value on the Y axis.

To create an Array, use the "Put" menu. When the new array is created, you will see two menus where you can change the properties of the array.



In the "canvas" properties menu, you can set the "X range" and "Y range", which represent the length in units of each axis, as well as the visual size of the array in pixels. In the "array" properties menu, you can set the "size" of the Array, which represents its length on the X axis, as well as it's name. Each Array you create must have a unique name, otherwise you won't be able to read from them.

Once an array is created and you are in Play Mode, you can click on the line inside and draw curves into the array. Arrays can also be filled with information from datafiles or soundfiles on your computer, as well as with mathematical functions. We'll discuss arrays in more detail in the arrays chapter of the Dataflow Tutorial.

### Graph

A "graph" is simply a container a graphical container that can hold several arrays. An array needs a graph to be displayed, so whenever you create an array from the menu, you will be asked whether you want to put it into a newly created graph or into an existing graph.

## A Note on using GUI Objects

PD uses a "vector-based" system for drawing the user interface. That means that every element on the screen is defined by a set of numbers rather than an image, and every change to these elements means that your computer must recalculate that part of the screen. For this reason, having a lot of GUI elements which are constantly changing is not recommended, as it can cause interruptions in the audio or slow down the response time of the interface.

In particular, be careful not to use too many of the following:

1. VU meters
2. Graphical bangs, number boxes, sliders or radio buttons with rapidly changing inputs
3. Arrays which are visible on the screen and which are redrawn

For a way of "hiding" GUI elements when they are not in use, please see the Subpatches and Abstractions chapters of the Patching Strategies Tutorial. And for a way of "hiding" the connections between GUI elements, please see the Send/Receive chapter of the Patching Strategies Tutorial.

# Troubleshooting

## I don't hear any sound!

First make sure that the box marked "compute audio" is checked in the main PD window. Then check to see that you have selected the right soundcard and drivers for your system, and that the soundcard is connected and operating. On OS X, make sure the check-boxes next to your selected soundcard have been checked in "Audio Settings". On Linux or OS X with Jack, make sure the Jack application is running. On all platforms, check the audio control panel which comes with your Operating System and make sure the proper output is enabled there, and that it's playback volume is turned up. Also make sure you are using the correct sampling rate in PD to match that of your soundcard.

## There are clicks, glitches or crackles in the test tone!

More than likely you have chosen a latency that is too fast for your computer and soundcard to handle. Return to the "Audio Settings" menu and increase the "delay" time there. On Linux, it is also possible that other processes running on your computer, or even a badly configured or slow graphics card, can affect the performance of PD. Consider running PD with the "-rt" flag enabled (Linux only!). This can be done from the command line, or by adding "-rt" to the "startup flags" under the "Startup" menu. On Linux or OS X with Jack, it is possible to set the latency of the Jack application to a greater amount and reduce glitches (called "xruns" in Jack) there as well.

## The test tone sounds distorted!

It is possible that you are playing the sound too loud for your soundcard. Using the controls of your soundcard to reduce the playback volume. Also make sure you are using the correct sampling rate in PD to match that of your soundcard.

## I'm not seeing any audio input!

Perhaps you did not enable sound input. On OS X, make sure the check-boxes next to your selected soundcard have been checked in "Audio Settings". Also, some cards with an uneven number of in and out channels can have problems in PD. Try setting the number of channels the same for the input and output. On all platforms, check the audio control panel which comes with your Operating System and make sure the proper input is enabled there, and that it's recording volume is turned up.

## I don't see any MIDI input!

Check to see that your MIDI devices or programs are actually sending data, and that your Operating System is correctly sending this data to PD. On OS X, check to see that you have selected the proper MIDI devices, and that the "Audio MIDI Setup.app" was running before you started PD. On Linux using the default MIDI drivers, check to see that you selected the proper MIDI device at startup. On Linux with the ALSA-MIDI drivers, make sure you have properly connected your MIDI devices or MIDI programs to PD. Using Jack with the "QJackctl" application is recommended for this purpose. On Windows, consider using an application like MIDI Ox/MIDI Yoke Junction to see, analyze and manage your MIDI connections.

## I get the message "... couldn't create" when I type an object's name and there's a dashed line around my object!

The reason for this error is that you have asked PD to create an object which does not exist. There can be several reasons for this error, and the most common one is spelling. Object names in PD must be spelled correctly, and they are case sensitive. [*Osc~*] or [*OSC~*] will not create in place of [*osc~]*, for example, nor will [*osc*] without the tilde. Sometimes users accidentally combine the creation argument and the object name, such as [*+1*] instead of [*+ 1*]. New PD users also often get confused between Objects and Messages, which are very different types of elements which can be placed in the patch from the "Put" Menu. You can use the "Find last error" function under the "Find" menu to track down which objects did not create. Please see the chapter called "The Interface" for more details.


# I get the message "... couldn't create" when I open a patch and there's a dashed line around my object!

If you get this error when opening a patch which you're pretty sure works otherwise (i.e. you've downloaded it from the internet or you created it in a previous PD session), then it's likely that there is an External Object which was available when the patch was created, but is not available now. You can use the "Find last error" function under the "Find" menu to track down which objects did not create. PD will preserve the location and connections of an object which fails to create, but it will not function. While most of the PD Externals are available in the PD Extended distribution, some are not, or require additional configuration of the "Path" and "Startup" settings. Please see the relevant sections in the "Configuring PD" chapter. If the External is not available in PD Extended, you may need to install it yourself.

# I get the message "error: signal outlet connect to nonsignal inlet (ignored)" when I open a patch.

This error tends to go with the previous error "I get the message '... couldn't create' when I open a patch...". Often this error means that an object has failed to create, usually because it uses an External Object which is not available in the current installation or configuration of PD. PD will preserve the location and connections of an object which fails to create, but it will not function. You can use the "Find last error" function under the "Find" menu to track down which objects caused errors. PD will treat uncreated objects as Dataflow Objects even if they were originally Audio Objects, so this error will follow the previous one. Please see the relevant sections in the "Configuring PD" chapter for information about setting the "Path" and "Startup" options. If the External is not available in PD Extended, you may need to install it yourself.

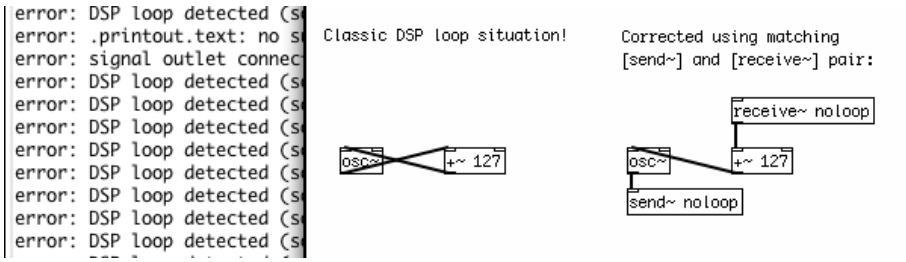
# I get the message "error: can't connect signal outlet to control inlet" and I cannot connect two objects together!

The output of Audio Objects (those with a tilde ~ in their name) normally cannot be connected to Dataflow Objects (those without a tilde ~ in their name). So PD will not allow these connections to be made. You might want to look at your patch and make sure that you are using the proper combination of objects.

# I get the message"error: DSP loop detected (some tilde objects not scheduled)" when I click "Audio ON", and the sound is not working!

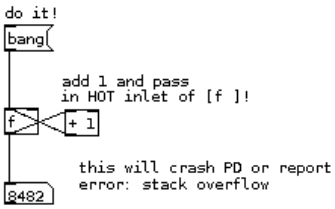
In an analog electronic system, you can easily connect the output of a mixer back to one of the inputs, turn up the channel and get feedback. This is because everything in an analog system happens pretty much

simultaneously. Computers do not work like this, however, and therefore you cannot ask a PD patch to compute results based on it's own simultaneous output. PD works in what are called Blocks (i.e. a group of samples, such as the default number of 64 samples), and all the Samples in each Block must be computed before they are output. So a DSP loop occurs when a patch needs information which is calculated inside the same Block in order to create output. You can use the "Find last error" function under the "Find" menu to track down which objects are causing the DSP loop. The easiest way around this problem is to create at least a one Block delay between the objects which are connected together. The objects [*send~*] and [*receive~*] are useful for this, because they have a built-in delay of one Block. To change the number of Samples computer in each Block, you can use the [*block~*] object.




# I get the message "error: stack overflow" when I connect two Dataflow Objects together!

A "stack overflow" happens when you have asked PD to compute a recursive operation, and this operation causes PD to run out of memory. Often this is the first step before crashing PD! A common example of a recursive operation which could cause this error is the classic counter, using [*float*] and [*+ 1*]. If the output of [*float*] is connected to the input of [*+ 1*], and the output of [*+ 1*] is connected to the right-most ("cold") inlet of [*float*], then a "bang" message sent to the left-most ("hot") [*float*] will output a number which increases by one every time that message is sent. If, however, the output of [*+ 1*] is connected to the left-most ("hot") inlet of [float], then sending the message "bang" to the left inlet of [*float]* will have a different effect. It will ask [*float*] and [*+ 1*] to add numbers together as fast as the computer will let them do it. Because PD will not stop and ask you "are you sure you want to do this?", this operation will quickly use up all the memory resources which PD has, and cause a stack overflow. Please see the sections on "Hot and Cold" as well as on "Trigger" in the "Dataflow Tutorials" section for more information on how to avoid stack overflows.




# I get the error message "connecting stream socket: Network is unreachable" when I start Pd!

If you are using the Linux operating system, and see this message when you start Pd, it means your machine cannot make a network connection to itself. You must configure your loopback network device. In many

Linux distributions, you can do this by answering "yes" when the system configuration tools ask if the machine will be a "network" (even if it won't).

# What is digital audio?

Since we'll be using **Pure Data** to create sound, and since PD treats sound as just another set of numbers, it might be useful to review how digital audio works. We will return to these concepts in the audio tutorial later on.

## Frequency and Gain

First, imagine a loudspeaker. The move the air in front of it and make sound, the membrane of the speaker must vibrate from it's center position (at rest) backwards and forwards. The number of times per second it vibrates makes the **frequency** (the note, tone or pitch) of the sound you hear, and the distance it travels from it's resting point determines the **gain** (the volume or loudness) of the sound. Normally, we measure frequency in **Hertz** (Hz) and loudness or gain in **Decibels** (dB).

A microphone works in reverse - vibrations in the air cause its membrane to vibrate. The microphone turns these acoustic vibrations into an electrical current. If you plug this microphone into your computer's soundcard and start recording the soundcard makes thousands of measurements of this electric current per second and records them as numbers.

## Sampling Rate and Bit Depth

To make audio playable on a Compact Disc, the computer must make 44,100 measurements (called **samples**) per second, and record each one as a **16-bit number**. One **bit** is a piece of information which is either 0 or 1, and if there are 16 bits together to make one sample then there are $2^{16}$ (or 2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2 = 65,536) possible values that each sample could have. Thus, we can say that CD-quality audio has a **sampling rate** of 44,100 Hz and a **bit-depth** or **word length** of 16 bits. In contrast, professional music recordings are usually made at 24-bit first to preserve the highest amount of detail before being mixed down to 16-bit for CD, and older computer games were famous for having a distinctively rough 8-bit sound. By increasing the sampling rate, we are able to record higher sonic frequencies, and by increasing the bit-depth or word length we are able to use a greater **dynamic range** (the difference between the quietest and the loudest sounds it is possible to record and play).

The number we use to record each sample has a value between - 1 and 1, which would represent the greatest range of movement of our theoretical loudspeaker, with 0 representing the speaker at rest in the middle position. When we ask PD to play back this sound, it will read the samples back and send them to the soundcard. The soundcard then converts these numbers to an electrical current which causes the loudspeaker to vibrate the air in front of it and make a sound we can hear.

## Speed and Pitch Control

If we want to change the speed at which the sound is played, we can read the samples back faster or slower than the original sampling rate. This is the same effect as changing the speed of record or a tape player. The sound information is played back at a different speed, and so the pitch of the sound changes in relation to the change in speed. A faster playback rate increases the pitch of the sound, while a slower playback rate lowers the pitch.

## Volume Control, Mixing and Clipping

If we want to change the volume of the sound, we have to multiply the numbers which represent the sound by another number. Multiplying them by a number greater than 1 will make the sound louder, and multiplying them by a number between 1 and zero will make the sound quieter. Multiplying them by zero will **mute** them

- resulting in no sound at all. We can also mix two or more sounds by adding the stream of numbers which represent them together to get a new stream of sound. All of these operations can take place in real-time as the sound is playing.

However, if the range of numbers which represents the sound becomes greater than -1 to 1, any numbers outside of that range will be truncated (reduced to either -1 or 1) by the soundcard. The resulting sound will be **clipped** (distorted). Some details of the sound will be lost and frequencies that were not present before will be heard.

## The Nyquist Number and Foldover/Aliasing

A similar problem occurs if one tries to play back a frequency which is greater then half the sampling rate which the computer is using. Thus, if one is using a sampling rate of 44,100 Hz, the highest frequency one could theoretically play back without errors is 22,050 Hz. This number which represents half the sampling rate is called the **Nyquist number**.

If you were to tell PD to play a frequency of 23,050 Hz, what you would hear is one tone at 23,050 Hz, and a second tone at 21,050 Hz. The difference between the Nyquist number (22,050 Hz) and the synthesized sound (23,050 Hz) is 1,000 Hz, which you would both add to and subtract from the Nyquist number to find the actual frequencies heard. So as one increased the frequency of the sound over the Nyquist number, you would hear one tone going up, and another coming down. This problem is referred to as **foldover** or **aliasing**.

## Block Size

Computers tend to process information in batches or chunks. In PD, these are known as **Blocks**. One block represents the number of audio samples which PD will compute before giving output. The default block size in PD is 64, which means that every 64 samples, PD makes every calculation needed on the sound and when all these calculations are finished, then the patch will output sound. Because of this, a PD patch cannot contain any **DSP loops**, which are situations where the output of a patch is sent directly back to the input. In such a situation, PD would be waiting for the output of the patch to be calculated before it could give output! In other words, an impossible situation. PD can detect DSP loops, and will not compute audio when they are present. For more information, see the "Troubleshooting" section.

## It's All Just Numbers

The main thing to keep in mind when starting to learn Pure Data is that audio and everything else is just numbers inside the computer, and that often the computer doesn't care whether the numbers you are playing with represent text, image, sound or other data. This makes it possible to make incredible transformations in sound and image, but it also allows for the possibility to make many mistakes, since there is no 'sanity checks' in Pure Data to make sure you are asking the program to do something that is possible. So sometimes the connections you make in PD may cause your computer to freeze or the application to crash. To protect against this save your work often and try not to let this bother you, because as you learn more and more about this language you will make fewer and fewer mistakes and eventually you will be able to program patches which are as stable and predictable as you want them to be.

# Building a Simple Synthesizer

## Introduction

These tutorial uses the concept of very simple electronic music instruments to introduce some of the core concepts of synthesizing and processing audio in Pure Data. Those who are already familiar with audio synthesis should quickly grasp how it is done in PD, while those with no previous knowledge will be introduced to its theory alongside its practical application in PD.

A synthesizer is one of the most fundamental instruments in electronic music. Its essential function is to generate a musical tone when it receives a note from either a keyboard or a sequencer. In analog electronic music, a synthesizer is built from several **modules**, or parts:

1) The **Oscillators**, which generates the tones.
2) The **LFO** (**Low Frequency Oscillator**), which usually modulates either the frequency or gain of the Oscillator(s), or the frequency of the Filter.
3) The **Filter**, which emphasizes and/or removes certain frequencies.
4) The **Envelope Generator**, which controls changes in frequency or gain over the duration of the note.
5) The **Amplifier**, which controls the gain of the synthesizer.

Synthesizers can be capable of playing one note at a time (**monophonic***)*, or several notes at a time, allowing for chords (**polyphonic**). The number of simultaneous notes that a synthesizer can play are called its **voices**. Originally, the word "Voltage" was used (i.e. Voltage Controlled Oscillator, Voltage Controlled Filter or Voltage Controlled Amplifier) because in an analog synthesizer each of these modules was controlled by electrical voltage from the keyboard, sequencer or another module. Because we're working in the digital domain, this voltage is replaced by data in the form of numbers, messages and streams of digital audio.

For this tutorial, we will construct a monophonic synthesizer in PD based roughly on the design of the famous **MiniMoog** analog synthesizer (but **much** simpler!), and with a sound which is useful for generating basslines. It will take input from the computer keyboard, a MIDI keyboard or the sequencer we will build in the the next tutorial. This synthesizer will be based on two **Oscillators** to produce the note, another oscillator (the **Low Frequency Oscillator**) which will change the gain of the sound, a **Filter** which will only allow only certain frequencies of the sound to pass, an **Envelope Generator** which will control the "shape" of the gain of the note, and a final **Amplifier** which will be controlled by the Envelope Generator and a volume setting on the screen.

## Downloads

The patches used in this tutorial can be downloaded from:

http://en.flossmanuals.net/floss/pub/PureData/SimpleSynthesizer/simple_synth.zip

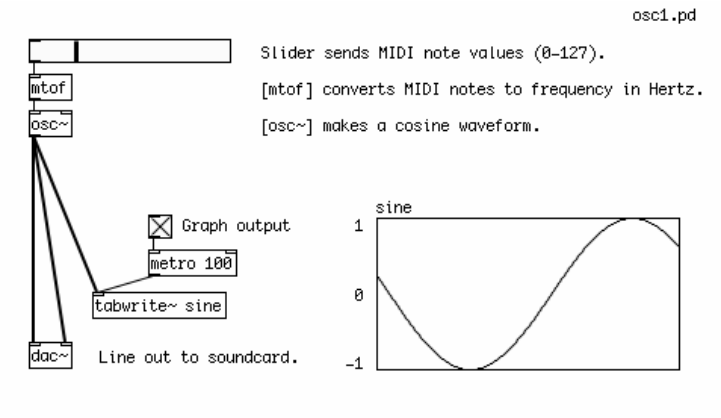
## Oscillators

Oscillators are the basic signal generators in electronic music. By combining, filtering or modulating them, almost any imaginable sound can be created. In Pure Data, audio signals are represented by a stream of numbers which are between the values of -1 and 1. So the waveform of each oscillator has been programmed to send out values within this range.

The name of each oscillator refers to their waveform, which is the shape of one period (or one Herz) of that oscillator. Different waveforms make different sounds.

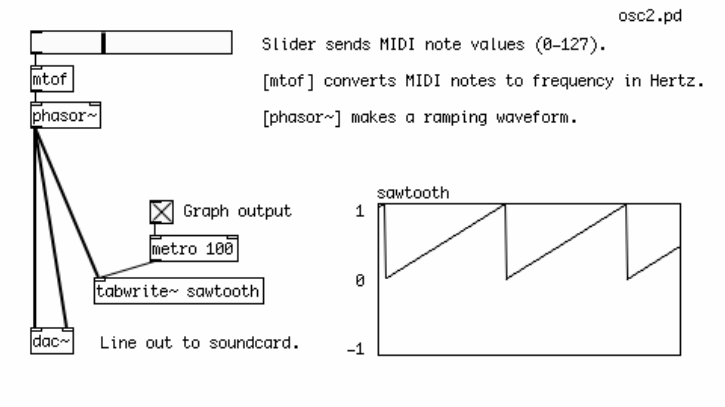
## Sine Wave Oscillator

The Sine Wave Oscillator makes a pure tone with no harmonics. The shape of the wave smoothly moves from 0 up to 1, back down through 0 to -1 and back up to 0.



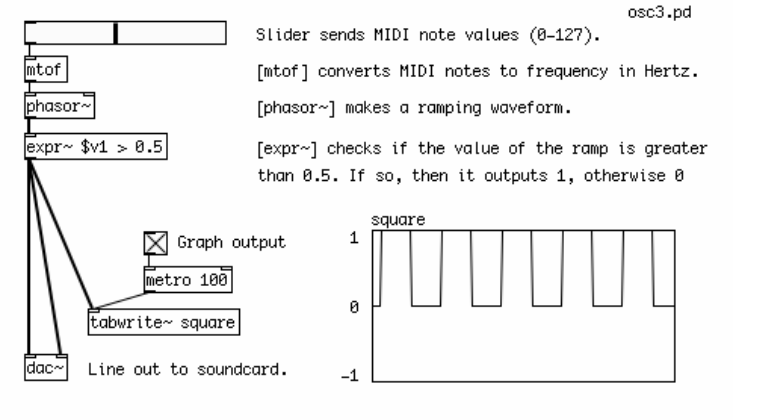

## Sawtooth Wave Oscillator

The Sawtooth Wave Oscillator sounds harsher in comparison to the sinewave, and it contains both odd and even harmonics of the fundamental frequency. This makes it ideal for filtering and for synthesizing string sounds. The shape of this wave ramps up sharply from 0 to 1, then immediately drops back to 0.



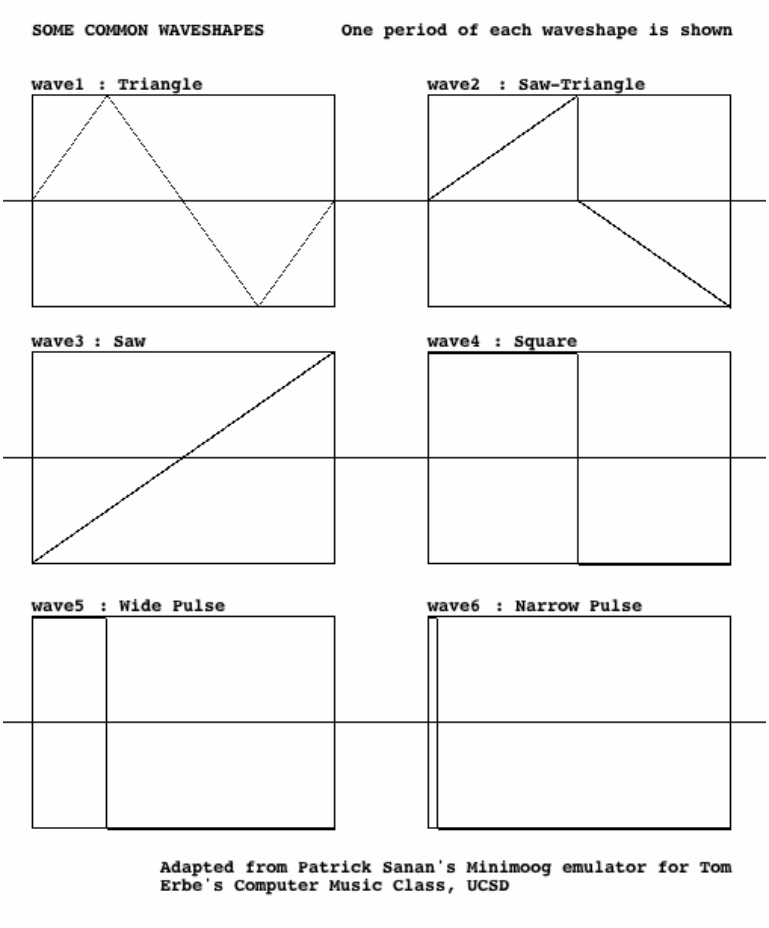

## Square Wave Oscillator

And the Square Wave Oscillator has a "hollow" sound, and contains only odd harmonics and is useful for synthesizing wind instrument as well as "heavy" bass sounds. It's shape alternates instantly between 0 and 1. Since there is no square wave object in PD, we create a square wave by checking to see if the output of the Sawtooth Wave object [phasor~] is greater than 0.5. If it is, the Expression object [expr~] outputs a 1, otherwise it outputs a zero. This creates the "high" (1) and "low" (0) states of the square wave, as you can see in the graph.




## Other Waveforms

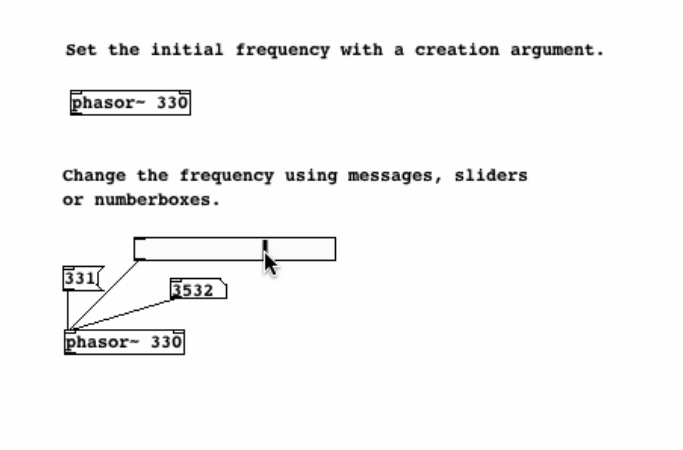
Other possible waveforms include a triangle wave as well as many other mathematical shapes.



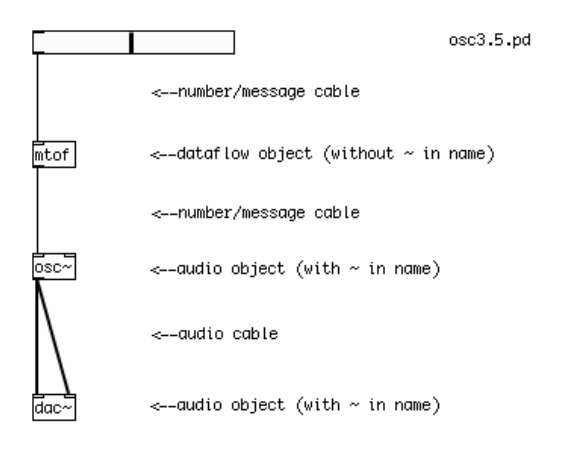

## Frequency

In order to to create sound, each oscillator object, takes an input of a number which represents a frequency in Hertz. This number determines the number of times the oscillator will make its waveform during one second.

By using an creation argument (a default setting typed into the object box when the object is first created), we can set the initial frequency of an oscillator. And by using an [hslider] (Horizontal Slider), a Number or a Message, we can send numerical messages to change the frequency of the oscillator.




In all the examples so far, notice the difference between the cable for numbers and messages, which is thin, and the cable for audio, which is thicker. Numbers and messages can be sent to audio objects (those with a ~ in their name), but usually audio cannot be sent to dataflow objects (those without a ~ in their name). Attempting to do so will cause PD to print "error: can't connect signal outlet to control inlet", and it will not allow the connection to be made.




## MIDI and Frequency

For many musical applications, the MIDI scale is a useful way of controlling the frequency of an oscillator. One can imagine the MIDI scale as a piano keyboard with 128 keys on it, and each key has been marked with a frequency in Hertz which represents that musical note. Below is a part of the table which makes up the MIDI scale. Three octaves are shown. The most important thing to notice is that a note which is one octave higher than another note (for example, the three A notes of 110 Hz, 220 Hz and 440 Hz) has a frequency which is twice that of the lower note.

| | MIDI Note | Frequency | MIDI Note | Frequency | MIDI Note | Frequency |
|---|---|---|---|---|---|---|
| C | 36 | 65.4063913251 | 48 | 130.8127826503 | 60 | 261.6255653006 |
| Db | 37 | 69.2956577442 | 49 | 138.5913154884 | 61 | 277.1826309769 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| D | 38 | 73.4161919794 | 50 | 146.8323839587 | 62 | 293.6647679174 |
| Eb | 39 | 77.7817459305 | 51 | 155.5634918610 | 63 | 311.1269837221 |
| E | 40 | 82.4068892282 | 52 | 164.8137784564 | 64 | 329.6275569129 |
| F | 41 | 87.3070578583 | 53 | 174.6141157165 | 65 | 349.2282314330 |
| Gb | 42 | 92.4986056779 | 54 | 184.9972113558 | 66 | 369.9944227116 |
| G | 43 | 97.9988589954 | 55 | 195.9977179909 | 67 | 391.9954359817 |
| Ab | 44 | 103.8261743950 | 56 | 207.6523487900 | 68 | 415.3046975799 |
| A | 45 | 110.0000000000 | 57 | 220.0000000000 | 69 | 440.0000000000 |
| Bb | 46 | 116.5409403795 | 58 | 233.0818807590 | 70 | 466.1637615181 |
| B | 47 | 123.4708253140 | 59 | 246.9416506281 | 71 | 493.8833012561 |

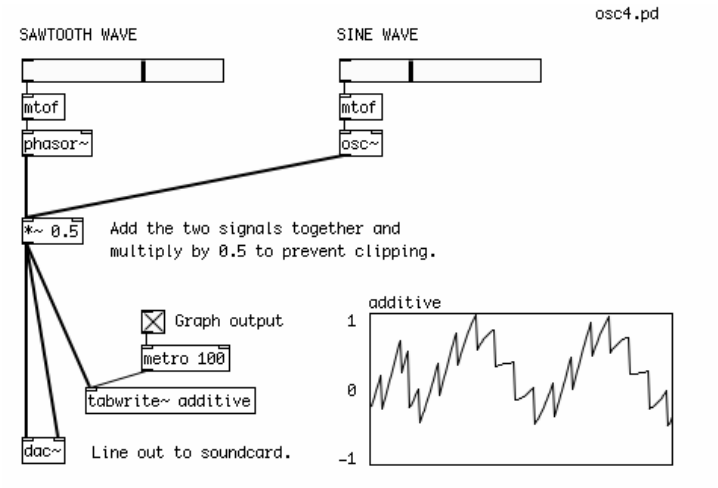
For the complete table, see http://www.borg.com/~jglatt/tutr/notefreq.htm

The object in PD which turns a MIDI note into a frequency in Hertz is called [mtof], or MIDI to Frequency. When the MIDI note "69" is sent to it, for example, it will output the number "440". Looking at our examples, you can see that each slider has a range of 0-127, and this is converted by an [mtof] object to a frequency which tells the oscillator what to do.

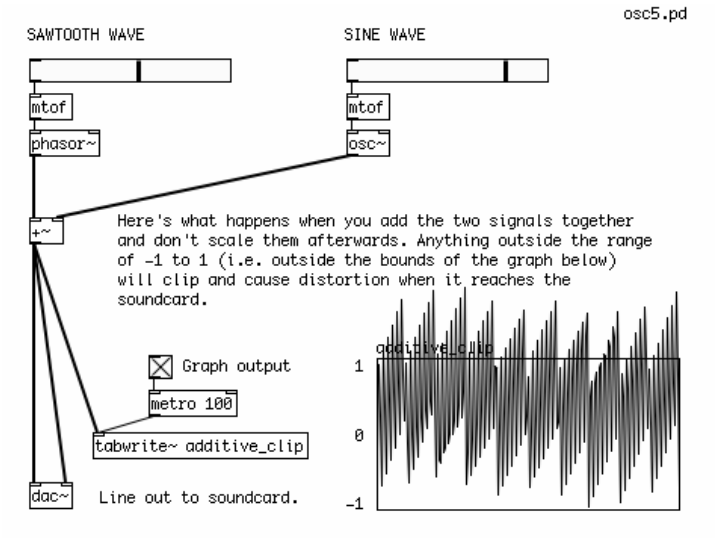
Of course, you aren't limited to the notes that Western music schools teach you are correct. So-called "microtonal" notes are possible as well. If you hold down the Shift key while using the mouse to change a Number, decimal numbers are possible, so that you can tell an [osc~] to play MIDI note number 76.89, for example.

## Additive Synthesis

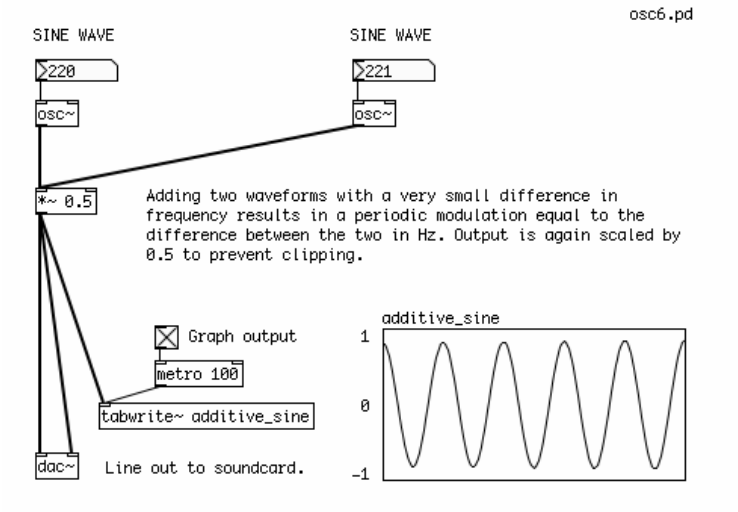
Because PD adds together the audio signals which come to the inlet of any audio object, it's simple to add two or more signals together into a single waveform. Below, we can see what happens when a Sawtooth Wave and a Sine Wave are added together. The resulting waveform is a combination of the shapes of both, added together. Note that the two waveforms are sent to an Audio Multiplication [*~] object, which multiplies the combined signal by half to reduce the total range of values sent to the soundcard.




Remember that, at full volume, each oscillator is going from either 0 or -1 to 1 many times a second. Because most everything in PD is simply numbers, any number of signals can be added together. However, if the combined values of those signals go outside the -1 to 1 range when they reach the Digital to Analog Converter [dac~] object (i.e. the line out to the sound card), then clipping and distortion will occur. Any value outside of the accepted range will simply be treated as a -1 or a 1. You can see how two combined signals can go outside this range on the graph in the patch below.




An interesting thing happens when we combine two waveforms whose frequencies are very close to each other. The combined values of the two waves interfere with each other, causing a periodic modulation of the sound. The frequency of this modulation is equal to the difference of the two original frequencies in Hz. This is known as a "beating frequency", or "phase interference". The sound of two oscillators slightly "de-tuned" from each other is often used for different kinds of electronic music sounds, such as a "fat" bass effect.



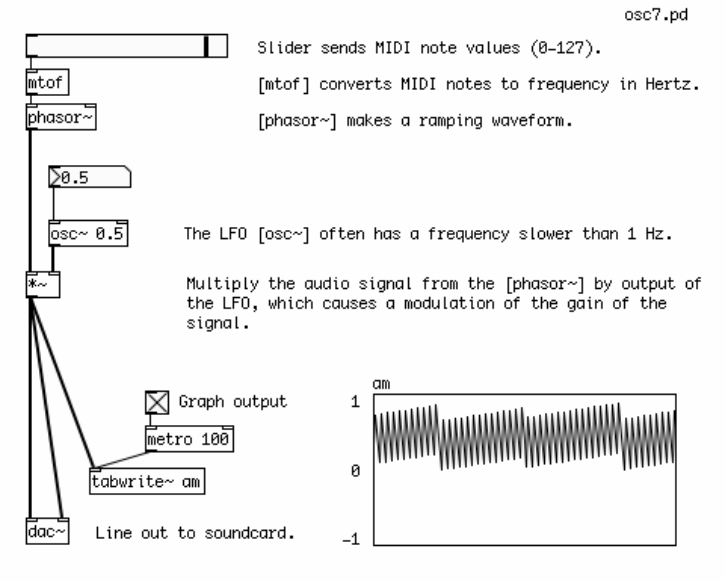

# Low Frequency Oscillators & Modulation

Low Frequency Oscillators (or LFOs) are used to control many aspects of a synthesizer, including the frequency of the Oscillators, the gain of the synthesizer or the cutoff frequency of the Filters (see below), and a complex synthesizer can contain many LFOs.

## Amplitude Modulation

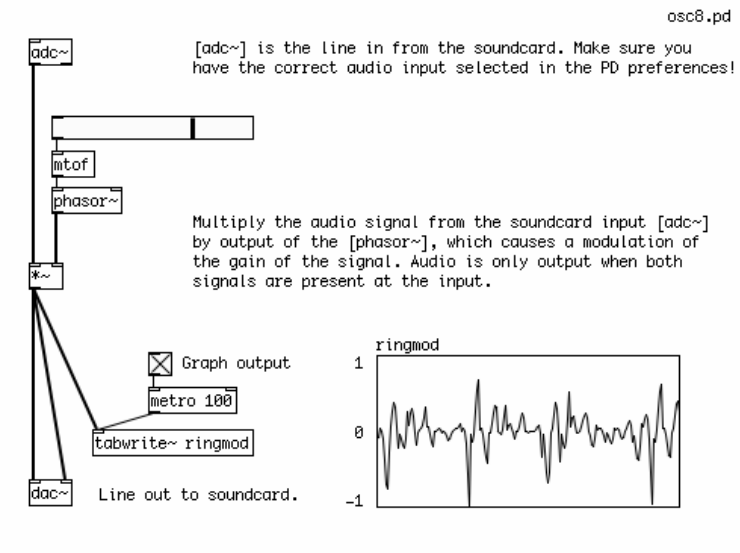
For a typical LFO, we can use the [osc~] object. By connecting the output of the LFO [osc~] to an Audio Multiplier [*~], we can modulate (i.e. change over time) the gain of any signal which passes through it. This effect is commonly called Amplitude Modulation, or AM Synthesis, and it gives additional character to the sound of a synthesizer.

Unlike the sound-generating Oscillators, we will not use MIDI note numbers to control this oscillator. This is because the speed of this oscillator must be much slower than that of musical notes. A typical LFO oscillates at speeds close to or even slower than once a second. So to control this [osc~], we will use a Number connected directly to its left-most inlet. By changing the number is this box, we send a frequency in Herz directly to the [osc~]. To send numbers smaller than one (where 0.5 would equal a speed of two seconds, for example), or numbers with any decimal place, use the Shift key while changing the Number with the mouse.




## Ring Modulation

You can also modulate one audio signal with another audio signal. This effect is called Ring Modulation. If you have a microphone connected to your computer, try the following patch. The sound of your voice will enter PD through the Analog to Digital Converter [adc~] object (the line in from the soundcard), and be modulated by a Sawtooth Wave [phasor~] object. Notice that there is no sound when only one audio signal is present (i.e. when you are not speaking). This is because one audio signal multiplied by zero (no audio signal) will always be zero. And the louder the input signal is, the louder the output will be.




The Ring Modulation effect was often used in Science Fiction movies to create alien voices. You may want to use headphones when running a microphone into PD to prevent feedback (the output of the speakers going

back into the microphone and making a howling sound).

## Frequency Modulation

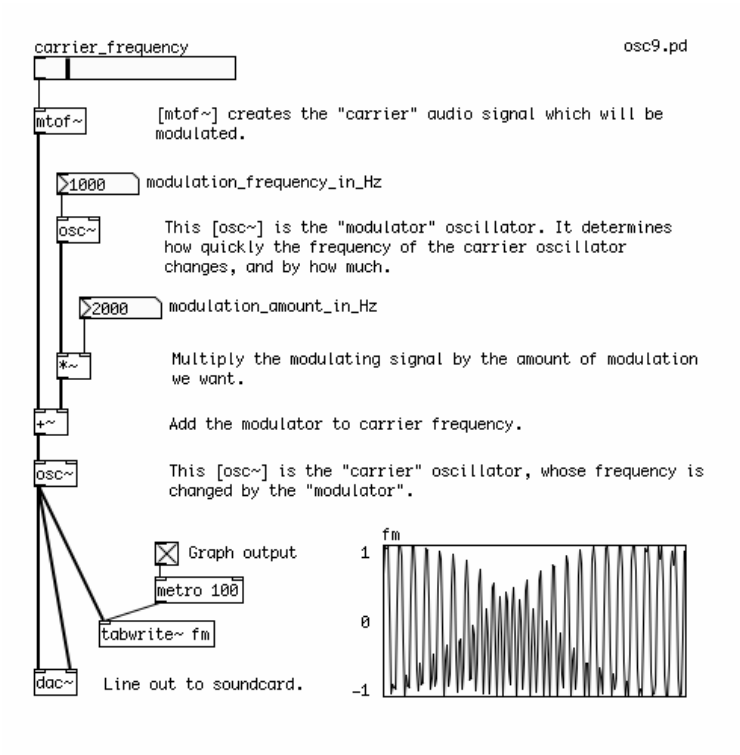
While Amplitude Modulation Synthesis changes the gain or volume of an audio signal, Frequency Modulation Synthesis, or FM Synthesis, is used to make periodic changes to the frequency of an oscillator. In it's simplest form, Frequency Modulation uses two oscillators. The first is the "carrier" oscillator, which is the one whose frequency will be changed over time. The second is the "modulator" oscillator, which will change the frequency of the "carrier".

For the "carrier", we only set the base "carrier frequency" using a Number box and a MIDI to Frequency [mtof~] object. Because all the adjustments afterwards will be done by audio signals, it's best to use the audio version of [mtof], hence the tilde is added to its name.

The "modulator" is where we do most of the adjustments. The first thing we want to do is set the frequency of the "modulator", i.e. how fast it will change the frequency of the "carrier". We do this with a Number box. The second thing we want to set is how much change we will make in the base frequency of the "carrier". So the output of the "modulator" [osc~] is multiplied by another Number box using an Audio Multiplier [*~] object to get the "modulation amount".

When this stream of numbers, which is changing with the speed the "modulator" and in the range set by the "modulation amount", is added to the "carrier frequency", then the "carrier frequency" will change as well. This stream of numbers is sent to the second [osc~], where it produces a complex sound which you can see in the graph.

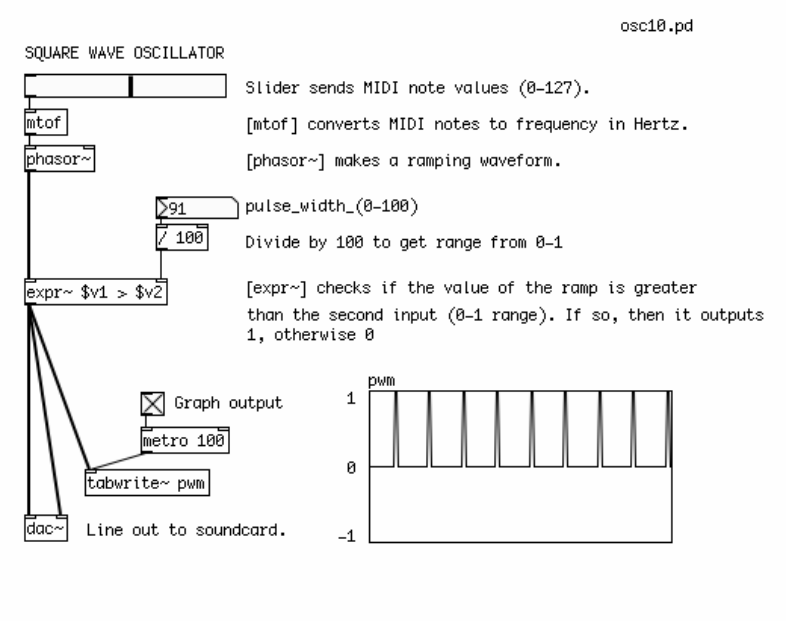
Even more complex sounds can be created by using further "modulators" to make changes in the frequency of the main "modulator" oscillator.



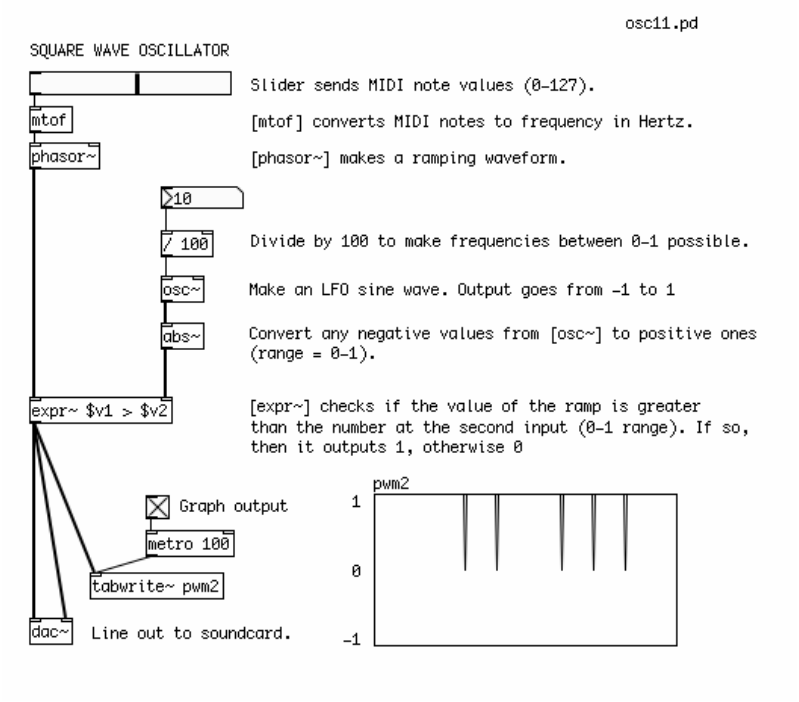
## Pulse Width Modulation

We've already seen how a simple mathematical check ("is the value of this audio ramp greater than 0.5?") can be used to turn a Sawtooth wave into a Square wave. This produces a Square Wave which is 1 half the time, and 0 the other half of the time. This is called the Pulse Width of the Square Wave. Different Pulse Widths make a different sound. And when we use a Square Wave as an LFO, different Pulse Widths will have different effects on the sound it is modulating.

When the Square Wave is 1 half the time and 0 the other half, it is said that it has a Pulse Width of 50%. To change the Pulse Width, it is necessary to send a new number to replace the "0.5" in the [expr~] object. The [expr~] object current has one Variable, which is written as $v1, and one constant, "0.5". If the constant is replaced with a second variable, $v2, then we can use a Number box to change the Pulse Width. Sending the number 0.25 will result in a Pulse Width of 25%, i.e. the Square Wave will be 1 a quarter of the time, and 0 three quarters of the time.




It is also possible to modulate the Pulse Width of the Square Wave with an LFO, which creates a unique sound. Instead of using a Number box, the output of a Sine Wave Oscillator is sent to an Absolute audio [abs~] object, which converts any negative values from the [osc~] into positive ones, and this stream of numbers is sent to the second inlet of the [expr~] object.




## Math & Logic Operations

Once we are working with Square waves, whose value is either a 0 or a 1, then we can also use Logic operations to create patterns. Logic operations take as their inputs either a 0 or a 1 (and nothing in between!), and compare the two numbers, giving either a 0 or a 1 as an output.

The AND operation works like this:

```
0 AND 0  =  0
0 AND 1  =  0
1 AND 0  =  0
1 AND 1  =  1
```

In short, this means that the output is 1 only when both inputs are also 1, otherwise the output is 0. In PD, this is represented by the [&&] object for numbers, and the [&&~] object for audio.

The OR operation works like this:

```
0 or 0 = 0
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

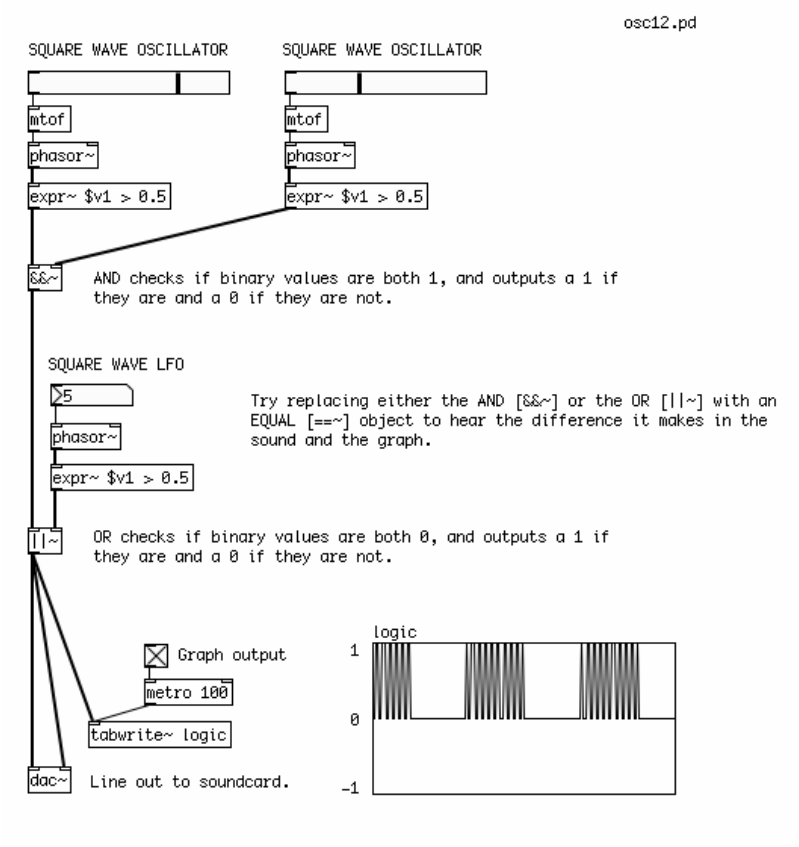
In short, this means that the output is 1 only when both inputs are also 0, otherwise the output is 0. In PD, this is represented by the [||] object for numbers, and the [||~] object for audio.

And the EQUAL operation works like this:

```
0 or 0 = 1
0 or 1 = 0
1 or 0 = 0
1 or 1 = 1
```

In short, this means that the output is 1 only when both inputs are the same, otherwise the output is 0. In PD, this is represented by the [==] object for numbers, and the [==~] object for audio.

In the following patch, different logic operations are used to make patterns from two Square Wave Oscillators, which are then compared with a final Square Wave Low Frequency Oscillator. What you will hear is a pattern of Square Waves which are switched on and off by each other. The final LFO makes a recognizable rhythm in the sound.



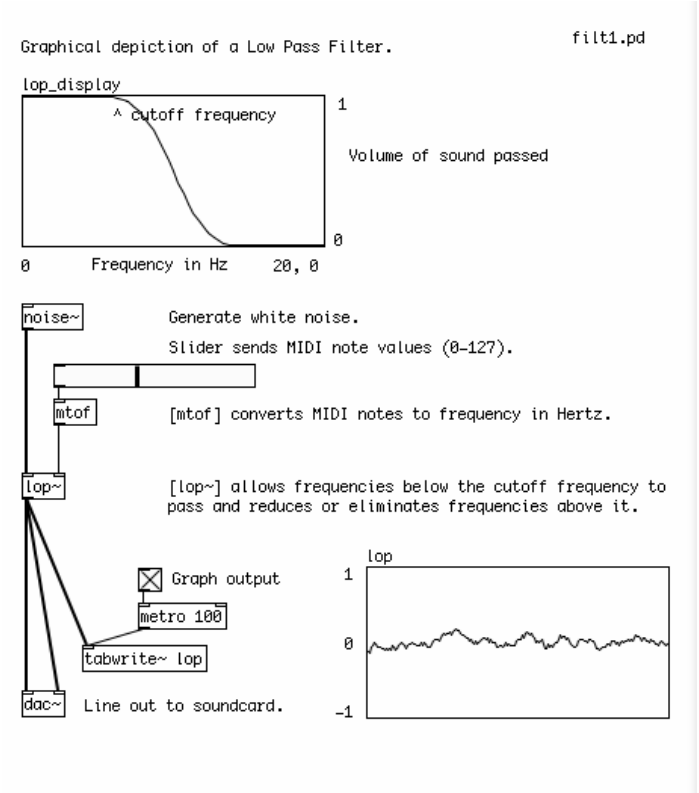

Try replacing any of the AND [&&~] or OR [||~] objects with an EQUAL [==~] object to hear the difference it makes in the sound. Or add further Logic operations to the output of the LFO to make more complex rhythmic patterns. You can also experiment with changing the Pulse Width as described in the previous patches.

# Filters

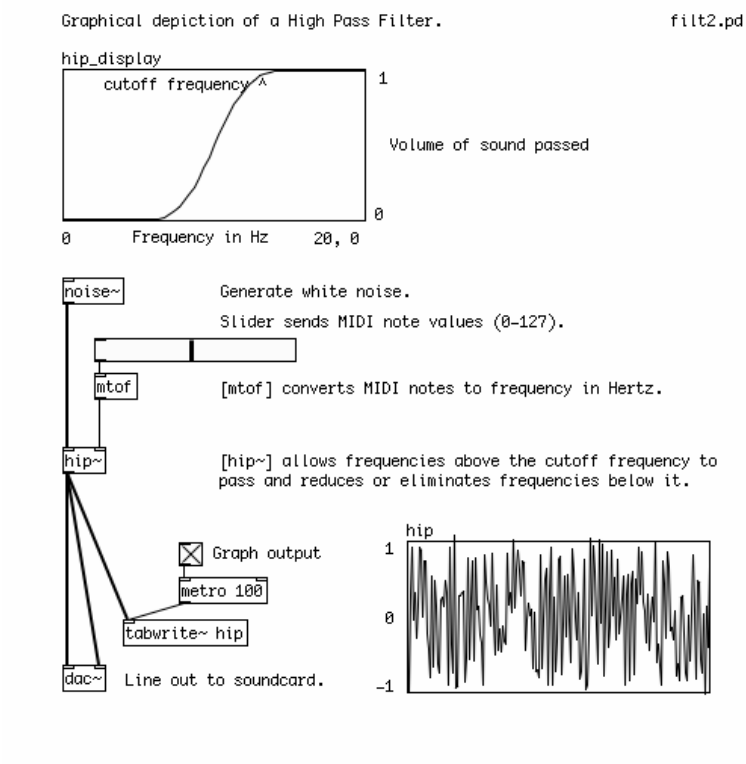
A filter works by allowing some frequencies through, while reducing or eliminating others.

A filter which allows only low frequencies to pass is called a Low Pass Filter. The object for this kind of filter in PD is [lop~]. It has one inlet for audio and one inlet for a number which determines the frequency in Hertz where the filter starts to reduce the audio (the Cutoff Point). Frequencies above the Cutoff Point are reduced

or eliminated.

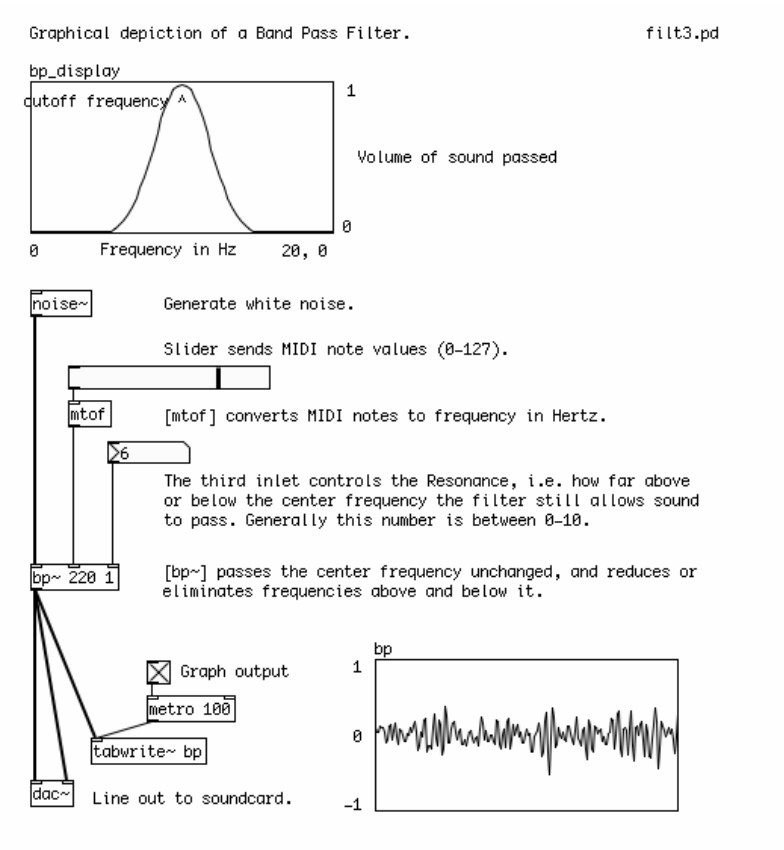



While one which allows only high frequencies is called a High Pass Filter. The object for this kind of filter in PD is [hip~]. It has one inlet for audio and one inlet for the the Cutoff Point. Frequencies below the Cutoff Point are reduced or eliminated.



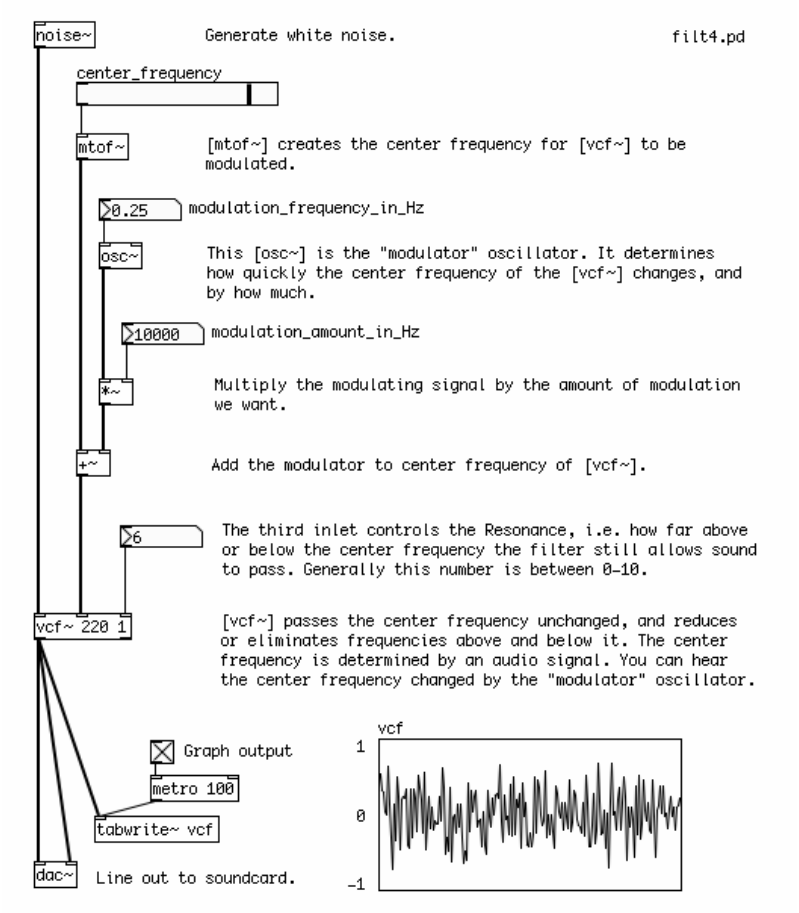

A filter which allows some range of frequencies between highest and lowest is called a Band Pass Filter. The

object for this kind of filter in PD is [bp~]. It has one inlet for audio, a second inlet for the center frequency that it will allow to pass and a third inlet for the Resonance, which determines the width of the range of frequencies it allows to pass (the Pass Band). The Center Frequency will pass unchanged, and frequencies higher or lower than that will be reduced or eliminated. How much they will be eliminated depends on the Resonance. Useful numbers for the Resonance tend to be between 0 and 10.




The three filters we've seen so far all take numbers to control their Cutoff or Center Frequencies as well as their Resonance (in the case of [bp~]. However, there are times when you might want to control the frequency of a filter with an audio signal. A typical situation is when a filter is "swept" by an LFO.

[vcf~] (Voltage Controlled Filter) is a filter whose Center Frequency and Resonance can be controlled by audio signals. The way this is done is quite similar to the tutorial on Frequency Modulation. A Slider sends a MIDI note to a MIDI to Frequency audio [mtof~] object to provide the Center Frequency to be swept, or modulated. Then we have an LFO [osc~] object, whose output is multiplied by the amount in Hertz which we want to sweep the filter frequency. This stream of numbers is added to the Center Frequency coming from the [mtof~] object and sent to the Frequency inlet of the [vcf~]

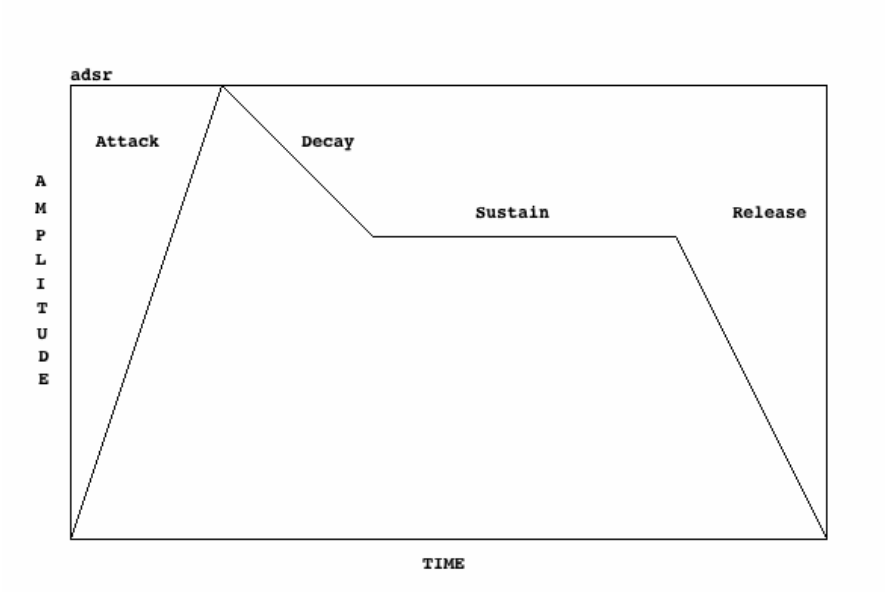



# The Envelope Generator

The **Envelope** of a sound refers to changes in either its pitch or gain over the duration of a note. A gain envelope is the most common, because it is used to synthesize the dynamics of acoustic instruments. For example, a piano has a very sharp or percussive attack, with the note becoming loud quite quickly before gradually fading out. A violin, on the other hand, takes a longer time for the sound to build up as the strings begin to vibrate, and then fades away relatively quickly. A gain envelope has five main characteristics:

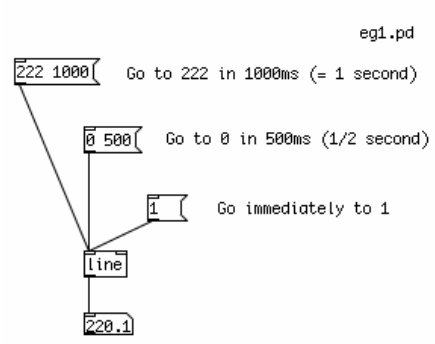
1) **Attack**: the length of time it takes the note to reach it's loudest point.
2) **Decay**: the length of time after the Attack it takes the note to reach it's Sustain volume.
3) **Sustain**: the volume of the note which is held until the note is Released.
4) **Release**: the length of time it takes the note to fade to zero after the key on the keyboard has been released.

This is commonly abbreviated as ADSR, and can be drawn graphically like this, where the horizontal axis represents time and the vertical axis represents volume:

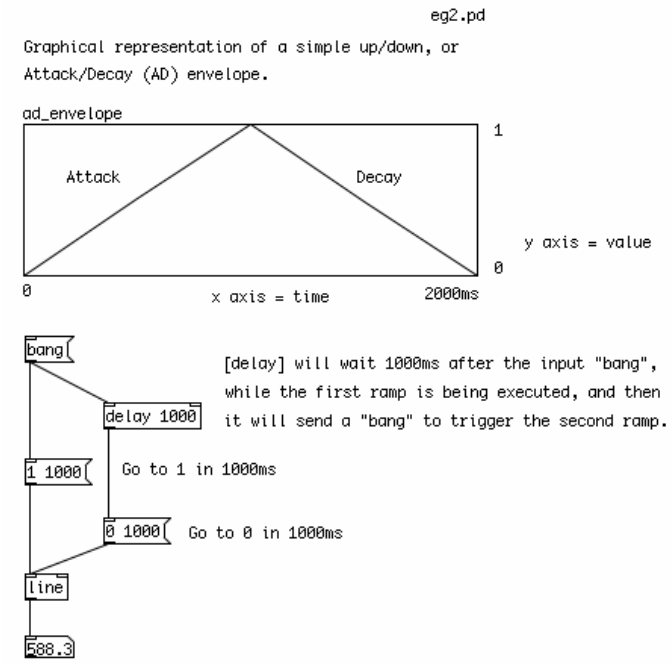



An additional parameter which comes from the MIDI world is called "Velocity", and it refers to how hard the key of the keyboard has been pressed. In our synthesizer, Velocity will refer to the volume of the note at its loudest point, i.e the peak of the Attack.

The simplest Envelope Generator can be made using the object [line]. This object takes two numbers, a target and a time (in milliseconds), and interpolates numbers to that target in the time given. If it is sent a single number, the time of the ramp is assumed to be zero, and [line] "jumps" to that value. It remembers that last value that it reached, so the next pair of numbers will start a new ramp from the current value. If a new pair of numbers is sent to [line] while it is still making a ramp, it will immediately stop that ramp and start the new one.

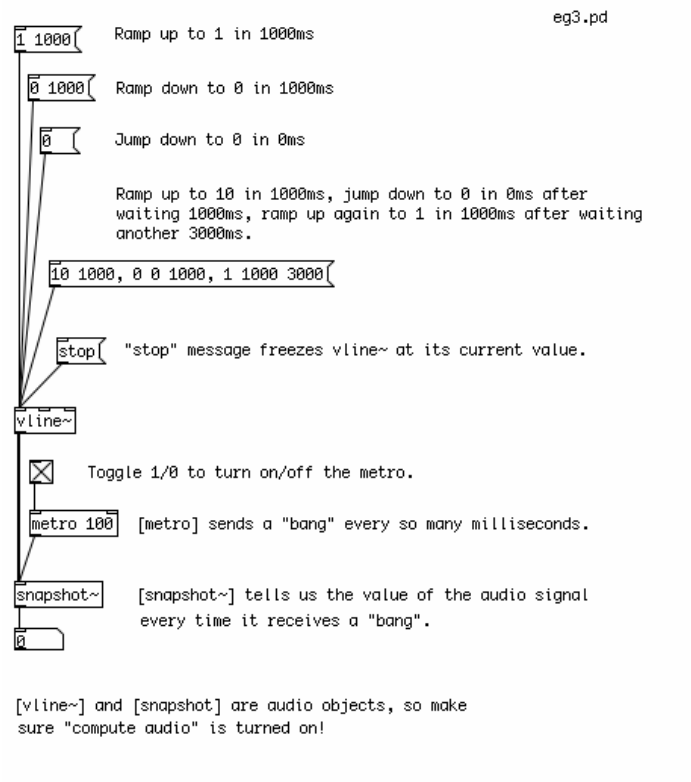



To make a simple up/down, or Attack/Decay envelope, we need to send two different messages to [line]. The first will tell it to go to "1" in a certain amount of time, the second will tell it to go back to "0" in a certain amount of time. These two messages can be triggered with a single "bang", as long as we delay the triggering of the second message long enough for the first ramp to finish, using the [delay] object.

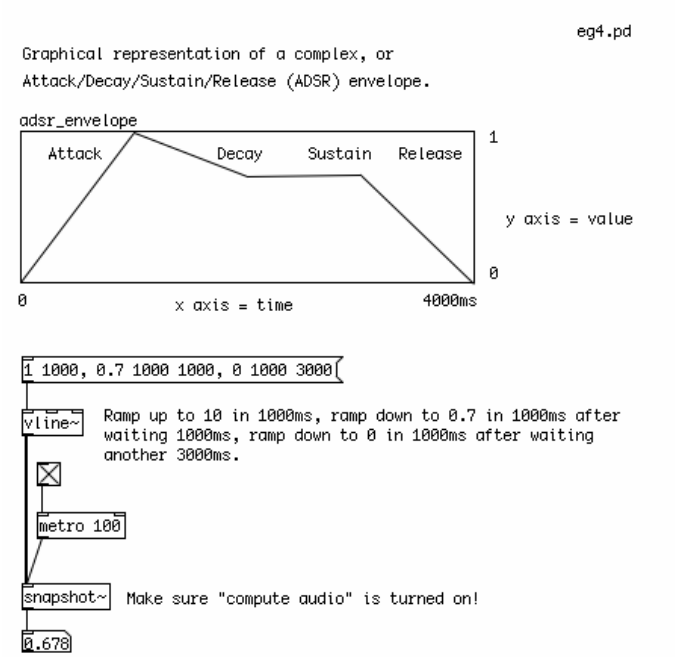



A more complex envelope can be created with the [vline~] object. This object can be programmed to make sequences of ramps in order, and with a delay in between them. For example, the message "10 1000, 0 0 1000, 1 1000 3000" would tell [vline~] to do the following:

Ramp up to 10 in 1000ms, then jump to 0 in 0ms after waiting 1000ms (from the start of the ramp), and finally ramp back up to 1 in 1000ms after waiting 3000ms (from start of the ramp).

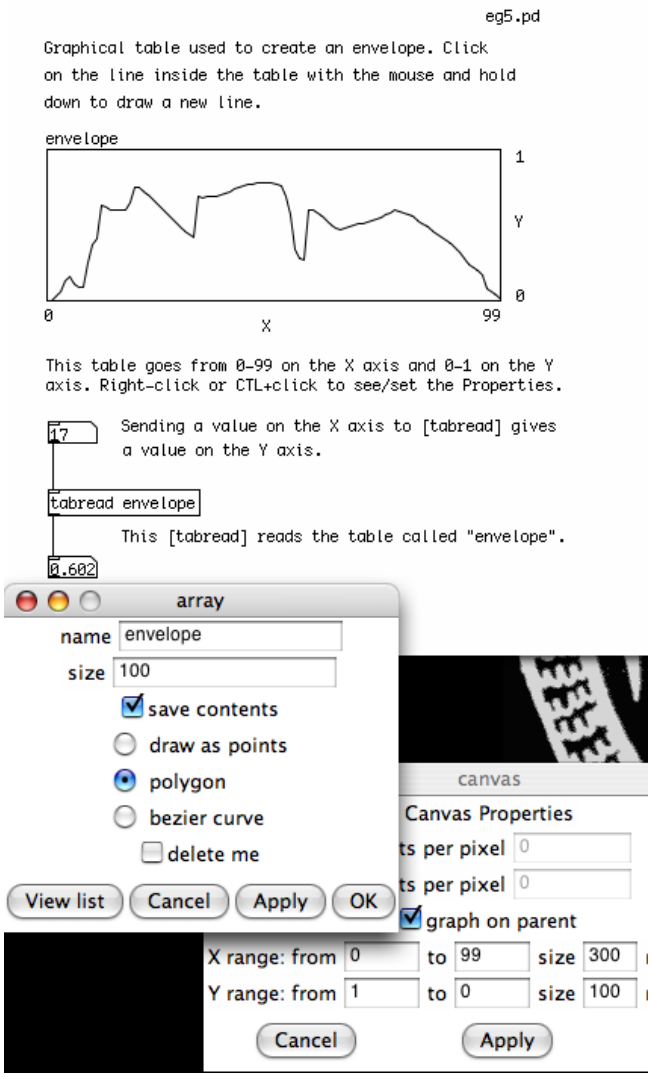



Because it accepts more complex messages, [vline~] is useful for the traditional Attack/Decay/Sustain/Release envelope. Also, [vline~] is an audio object rather than a dataflow object, which means it is more suitable for audio multiplication, as we will see in the next section.



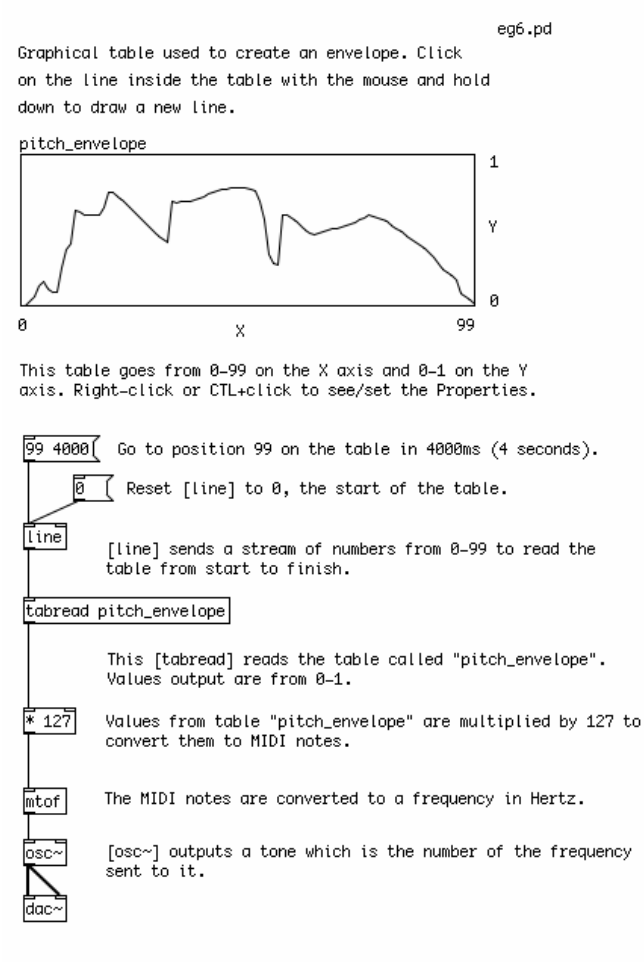

For an envelope with an arbitrary curve, a table is the most useful way to go. First we must create a table, by using the Put menu to place and Array in the patch. When we do that, we will see two Properties dialogs appear. In one, we name the Array "envelope" and set a length of 100 units. In the second we can change the graphical appearance and the range of the X and Y axes. In this case, set the X range to "0 to 99", and the Y range to "1 to 0". The size can be anything that is convenient, and is measured in pixels. You can get these Properties dialogs back by Right-clicking or CTL+clicking on the Array. These dialogs appear under the screenshot below.

To read a table, we can use the object [tabread]. The [tabread] object takes a creation argument of the name of the table it is supposed to read.  In order to draw inside the table, you should click on the line and drag with the mouse. A value sent to the inlet of [tabread] represents a point on the X axis, and the output is the corresponding value on the Y axis.




If we want to read a table continuously from start to finish, we can use [line] to send a continuous stream of numbers to [tabread], starting with the first position on the X axis of the table ("0"), and ending with the last position ("99"). Remember that [line] takes a pair of numbers, the target ("99", the end of the table) and the time it takes to get there (4000 milliseconds, or 4 seconds). When we want to start over from the beginning of the table, we send a single number, "0", and the [line] object jumps back to it.

In the example below, [tabread] gets values between 0-1 from the table "pitch_envelope". We multiply these numbers by 127 with a [*] (Multiplication) object, to get a MIDI note value between 0-127. After that, we use a [mtof] (MIDI to Frequency) object to convert the MIDI notes into a frequency in Hertz. The frequency is sent to a sine wave oscillator [osc~] object, which sends audio to the [dac~] (Digital to Analog Converter), PD's connection to the soundcard.

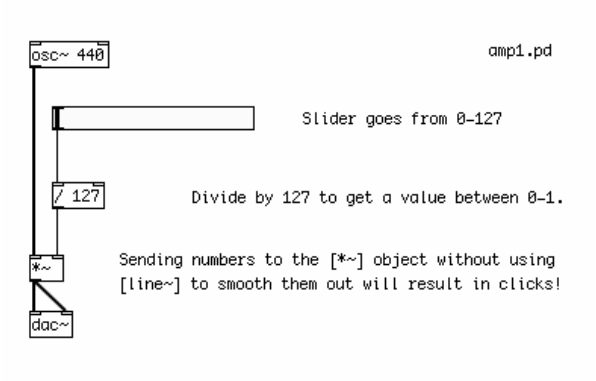



# The Amplifier

The next step in our synthesizer is to create the audio amplifier, which will change the gain of the signal. Whatever method you use to create your envelope, if you are using it to control the amplitude of a signal you will want to make sure the output is an audio signal as well. This is done to avoid clicks in the audio.

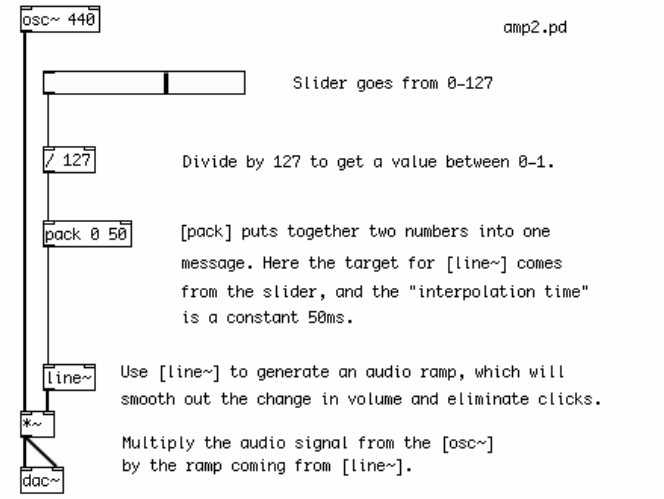
## Using a Slider

In the two examples below, an audio signal from the Sine Wave Oscillator [osc~] is being changed manually, via a slider, in the same way as the Volume knob on your home stereo might work. In the first example, the numbers from the slider, which go from 0-127, are divided by 127 with a Division [/] object, to get them within the range of 0-1. These numbers are sent directly to the right inlet of the Audio Multiplication [*~] object, so that every audio sample coming from the [osc~] is multiplied by a number between 0-1. This will reduce the volume of each sample. "0" means no sound, "1" means full volume. However, these changes in volume will have clicks in them, as each number from the slider is sent to the [*~].

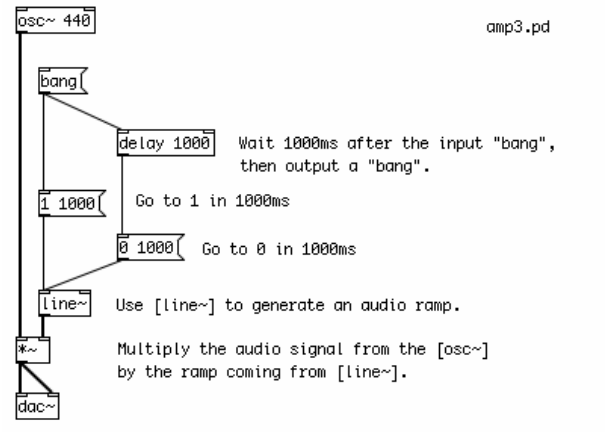



## Using [line~], [vline~] and [tabread4~]

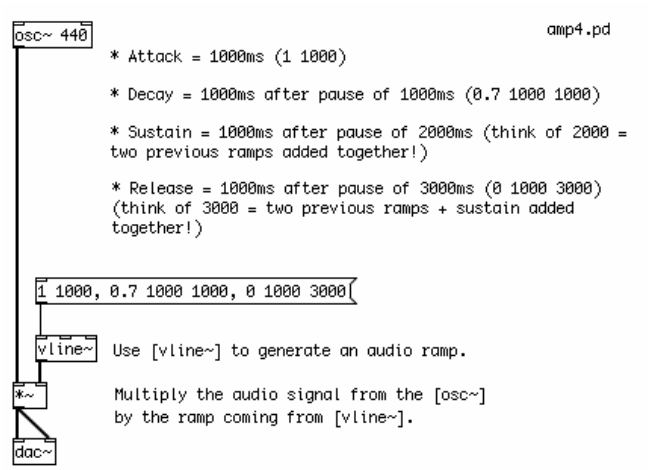
In the second example, the numbers from the slider are sent to an Audio Ramp object [line~], after being packed together into a message by [pack] with the number 50. What this message, which might appear as "0.76 50" for example, tells line is that it should ramp to the next number in 50 milliseconds. This is known as Interpolation, which is to smoothly transition from one value to another by providing (or guessing) all the values in between. Since the [line~] object is an audio object, the signal it sends out should cleanly control the volume of the audio signal.



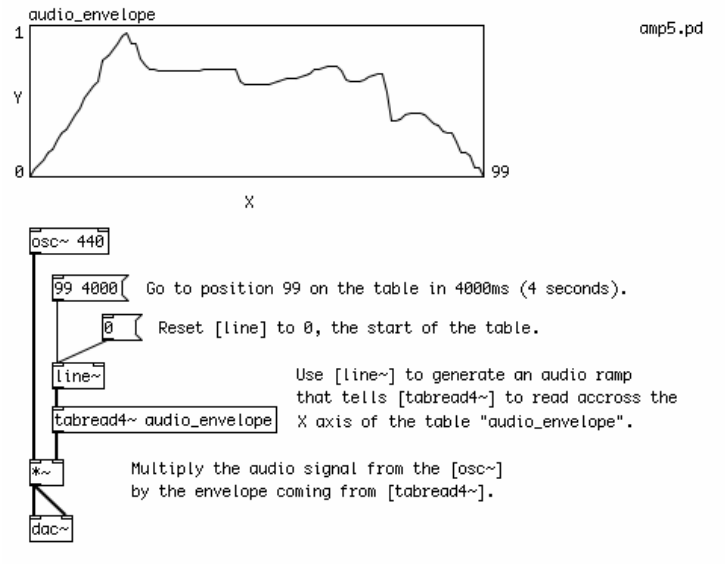

If you use [line] to make your envelope, you can make an audio signal by using the audio object [line~] instead.




[vline~] outputs an audio signal already.




And to read a table and get an audio signal out, the [tabread4~] object is useful. Note that [tabread4~] responds better when controlled with an audio signal as well, so [line~] is used instead of [line].




# Controlling the Synthesizer

Reviewing what we've covered in this tutorial, we can see that all the building blocks of a simple synthesizer are present.

We have various Oscillators to generate the tones. Then there are Low Frequency Oscillators, which provide the possibility to modulate either the frequency or gain of the Oscillator(s), or the frequency of a Filter. There are also different types of Filters, which emphasizes and/or removes certain frequencies. Envelope Generators control changes in frequency or gain over time, and Amplifiers control the final gain of the synthesizer.

The way each of these elements are put together gives the final definition to the sound and functionality of your synthesizer. And there are an almost infinite number of was to do this! In the following examples, we'll look at some simple ways to combine the different elements of a basic synthesizer with ways of controlling it, either from the computer keyboard, a MIDI keyboard or a 16 step sequencer which we will build.

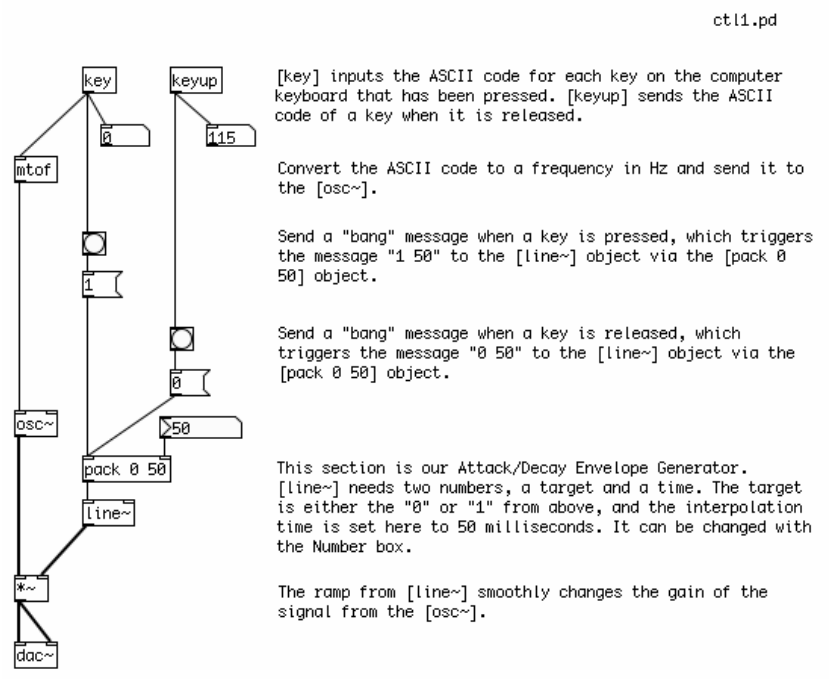
## Input from the Computer Keyboard

To get a very crude input from the computer keyboard, we can use the objects [key] and [keyup]. Each key on the computer keyboard has what is called an ASCII value, which is a number used to represent that key. [key] outputs this number when a key is pressed, and [keyup] sends this number when a key is released. Like MIDI Notes, these numbers are within the range of 0 to 127. However, the layout of these ASCII values on the computer keyboard is far from musical! But they are a good way to get some immediate input into a patch, and later on [key] and [keyup] can used to trigger different elements of a PD patch besides musical notes.

In the following patch, the ASCII values of the computer keyboard are treated as MIDI notes and control the frequency and volume of a Sine Wave Oscillator. We will use [line~] as a simple Attack/Decay Envelope Generator here, to make the envelope of the note smooth and to avoid clicks.

When a key is pressed, [key] sends the ASCII value, which becomes a frequency through [mtof] and controls the [osc~]. At the same time, when the key is pressed, the output of [key] is converted to a "bang", which triggers the message "1" to be sent to [pack]. In [pack], this "1" is packed together with "50" to make a message which says "1 50". [line~] interprets the message "0 50" to mean "ramp to 1 in 50 milliseconds". This will smoothly ramp the audio signal from the [osc~] up to full volume.

When a key is released, then [keyup] will send a number out as well. We will convert this to a "bang", which sends the message "0" to [pack]. [pack] then makes the message "0 50" and sends it to [line~], and [line~] will ramp back down to 0 in 50 milliseconds.

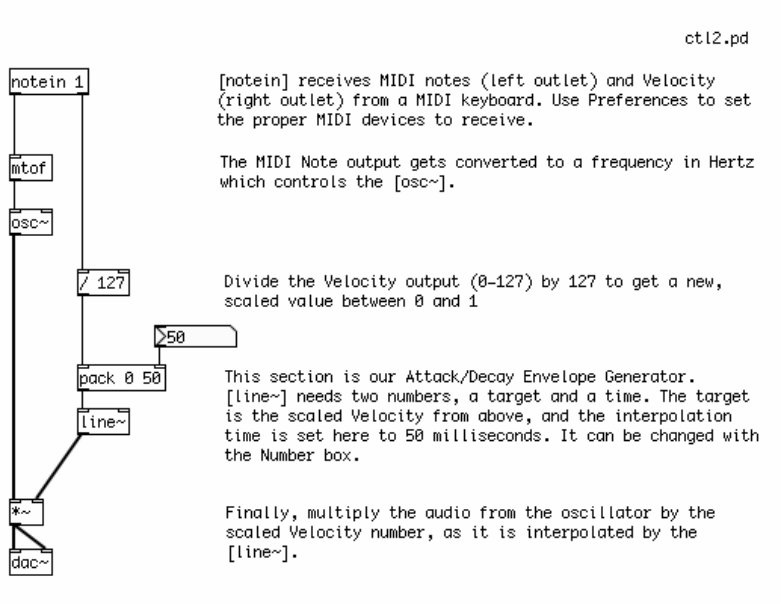



## Input from a MIDI Keyboard

This task is made simpler (and more musical!) with the addition of a MIDI keyboard. Make sure you have selected the proper MIDI input device in your Preferences (see Configuring PD for more details). The [notein] object receives the MIDI Note and Velocity information from the MIDI keyboard. Because usually you will want to listen to the first MIDI keyboard you plug in, we give [notein] a creation argument of "1", thus [notein 1] will receive MIDI Notes on MIDI Channel 1. The MIDI Note played will come out the left outlet, and the Velocity (how hard the key is pressed) will come out the right outlet.

The MIDI Note we send to an [mtof], which converts it to a frequency and sends it to the [osc~]. The Velocity

we divide by 127 to get a value between 0 and 1. This value gets [pack]ed together with 50, and sent to the [line~] object, which we will use again as an Attack Decay Envelope Generator. [line~] makes a 50 millisecond audio ramp, either to "0" when the MIDI key is released and the Velocity is "0", or to a value between 0 and 1 when the MIDI key is pressed, depending on how hard it has been pressed. [line~] sends an audio ramp to the Audio Multiplier [*~], which smoothly changes the volume of the audio signal form the [osc~].

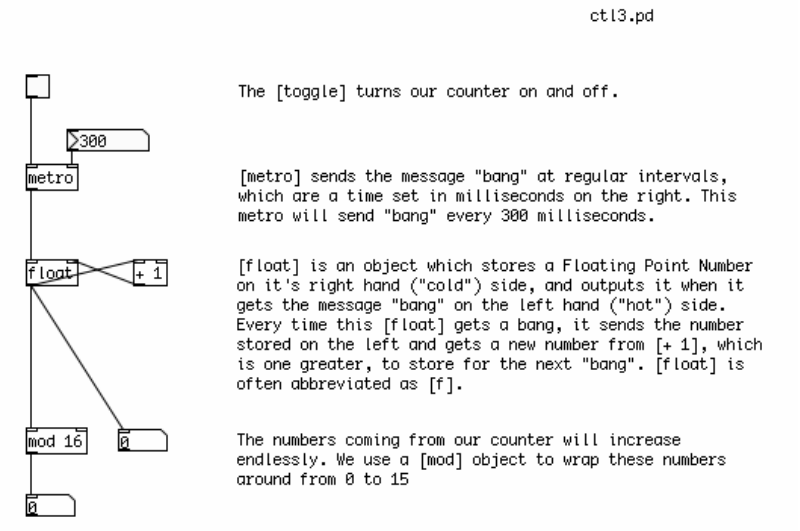



## Building a 16-Step Sequencer

Besides using a keyboard, another way to control a synthesizer is with a Sequencer, which stores MIDI Notes and plays them back in sequence, and at a speed which can be changed from faster to slower.

Before we can build the note-storing section of the Sequencer, however, we have to learn a little bit about dataflow in PD in order to make a counter. This counter will count from 0 to 15, and each number it sends out will trigger one of the steps in a 16-Step Sequencer.

The patch below shows a counter, built with basic PD objects.

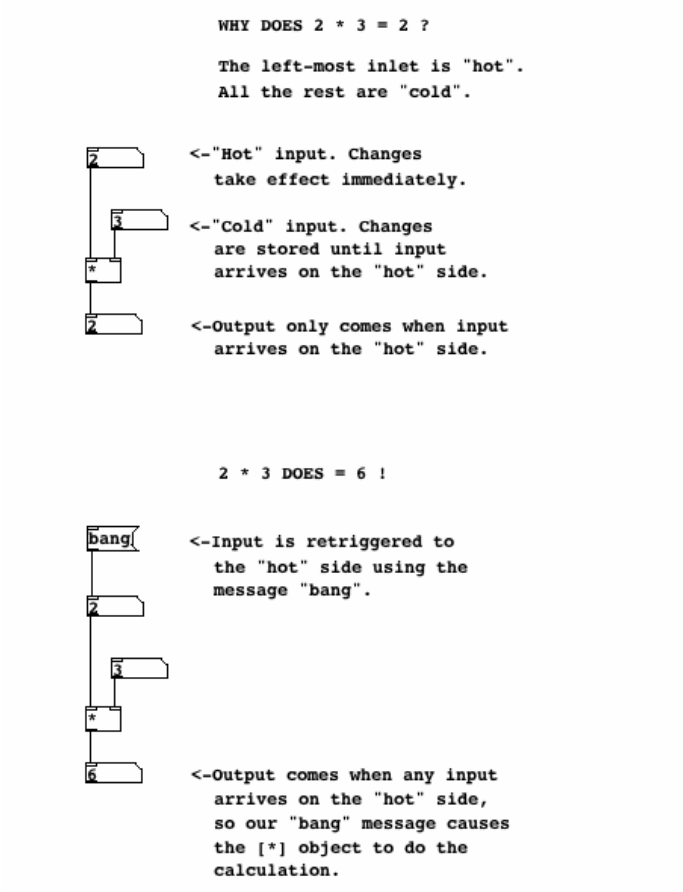



[metro] is used to send the message "bang" every so many milliseconds. This interval is set by a Number sent

to the right inlet. The [metro] is turned on and off by sending either a "1" or a "0" to the left inlet. We use the [toggle] object to send these messages.

## Hot and Cold

Below the [metro] is a construction which takes advantage of one of the most fundamental lessons in learning about dataflow in PD: "hot" and "cold" inlets. The left-most inlet of almost any non-audio PD object is called the "hot" inlet. Any input to the hot inlet of an object gives immediate output. Any other inlet to the right of the hot inlet is called a "cold" inlet. Input to a cold inlet is stored in the object, waiting for input on the hot side. In this case, when a new number comes to the hot side of the [*], it is multiplied by the number stored in the cold side. The information in the cold inlets is kept there until new information received at that inlet changes it, or until the object is re-created (by retyping its name, cutting/pasting the object or by reopening the patch).




So in our counter, there is an object called [float], which stores and outputs a Floating Point Number. Floating Point Number is another name for a number with a decimal place, usually called simply a "float". The opposite of a "float" is an Integer, or "int", which has no decimal place. All numbers in PD are assumed to be floats. When [float] receives a "bang" to its left ("hot") inlet, it outputs the float which is stored on it's right ("cold") inlet. When this [float] outputs a number, it is also sent to the inlet of a [+ 1] object, where 1 is added to that number and sent back to the "cold" inlet of [float] to wait for the next "bang". So, every time this construction receives a "bang", the number it will output will be 1 more than before.

For more information on "hot" and "cold", as well as other descriptions of how to get used to how dataflow works in PD, please see the Dataflow Tutorials in this FLOSS Manual.

The numbers sent from our counter will increase endlessly. In order to keep them within the bounds of our 16-Step Sequencer, we need a way to "wrap" these numbers around so that they start over when the counter reaches 16, and every other division of 16 that comes later on. [mod] is the object which does this.

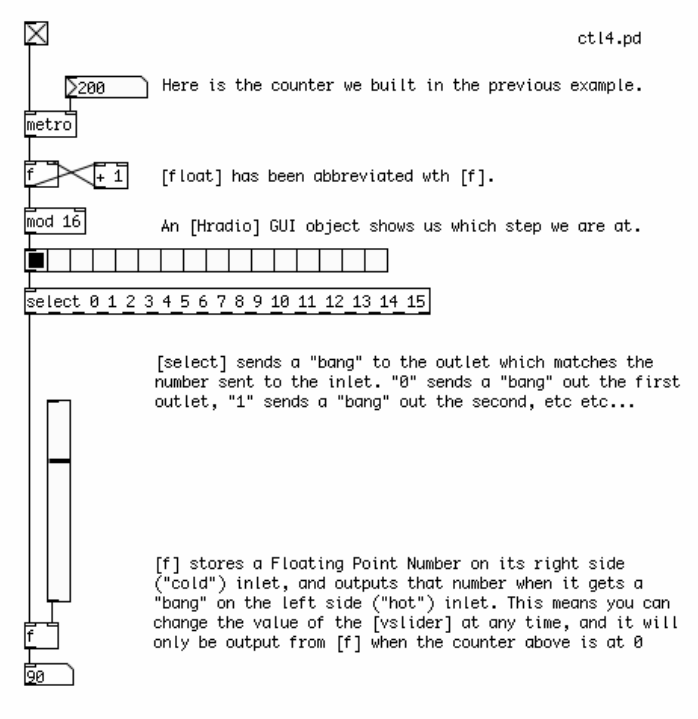
Technically, [mod] means "modulo", and it outputs the remainder of a division operation using the number in the creation argument. Thus "16" becomes "0", "17" becomes "1", "18" becomes "2" and so on.

## Storing and Retrieving MIDI Note Values

In the next patch, we see how to store and recall a number from an [hslider] using the [float] object as well. Here, [float] has been abbreviated to the commonly used [f]. At the bottom of our counter construction from the previous example, we have replace the Number with an [hradio] array of buttons, so that we can see which step of our Sequencer we are at. (Right or Control+Click on the [hradio] to see its properties, and type "16" in the "number" field to set the size.)

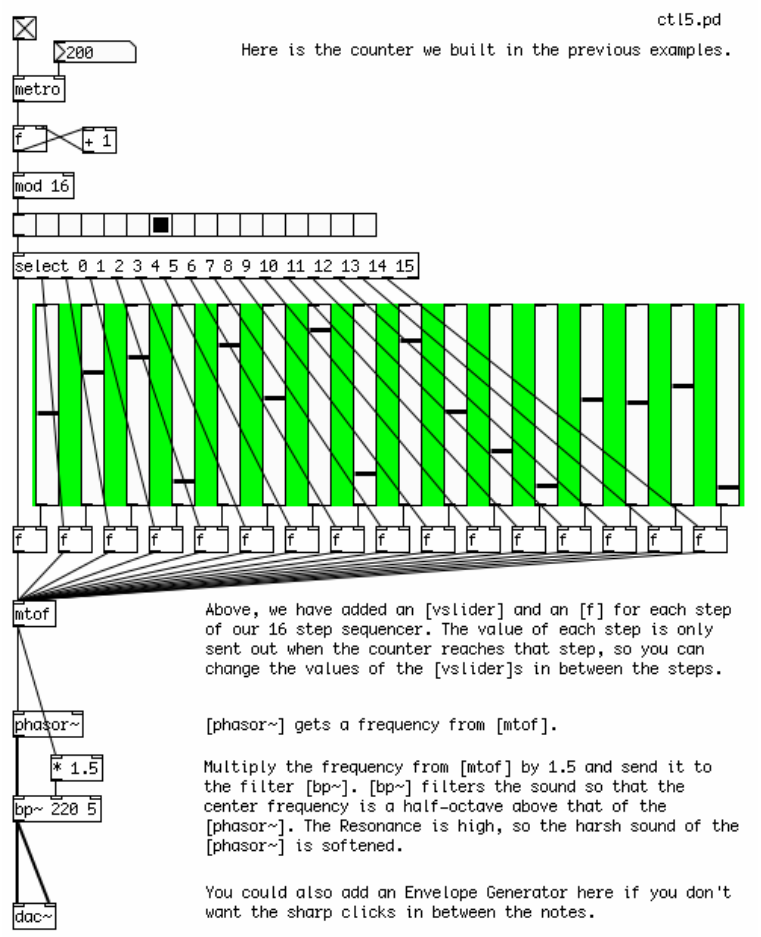
Below the counter we have the object [select]. This object checks the input on its left inlet against either the input on the right inlet, or in this case against a series of creation arguments. When the input on the left matches one of the creation arguments, then the message "bang" comes out of the corresponding outlet. Thus, an input of "0" will send a "bang" out the first outlet, an input of "1" sends a"bang" out the second outlet, etc etc. In this way, we have a seperate "bang" for each step in the Sequencer.

For each step in the Sequencer, we will use a [f] object to store a MIDI Note send from a [vslider]. The range of the [vslider] is 0-127, and the number it outputs is sent to the "cold" inlet of [f], to wait for a "bang" to come to the "hot" inlet. When that "bang" comes, the MIDI Note is sent out. You can change the value of the [vslider] with the mouse at any time, and the MIDI note will only be sent at step 0 of the sequence.




## The Finished 16-Step Sequencer Patch

And here we have the finished patch, with all 16 steps included, connected to a simple synthesizer. Each step of the sequence sends a MIDI Note to [mtof], which controls the frequency of a [phasor~] as well as the frequency of a Band Pass Filter [bp~]. The creation arguments of the [bp~] set it's starting frequency as 220 Hz, but this is changed every time it receives a new number from the [mtof], which has been multiplied by 1.5 to make the center frequency of the filter a half octave above that of the Sawtooth Oscillator [phasor~]. The resonance is set very high, at "5", so the harsh sound of the [phasor~] is softened.

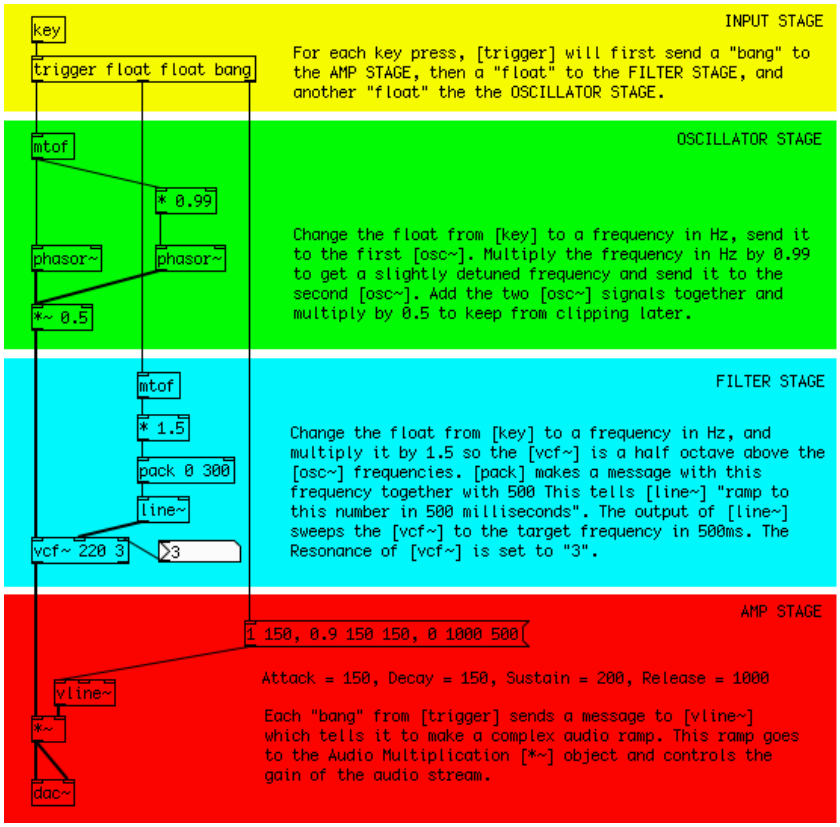



In this version, no Envelope Generator is used, so the volume of the audio going to the soundcard remains constant. This leads to noticeable clicks when the frequencies of the MIDI Notes change. An Envelope Generator based on [line~], [vline~] or [tabread4~] could be inserted between the output of [bp~] and the [dac~] if desired.

# A Four Stage Filtered Additive Synthesizer

Our final example shows all the different elements of a simple synthesizer combined together into an instrument which can be played by the computer keyboard using [key]. It has four distinct sections:

* The INPUT STAGE: where note information is received and sent to the other stages.
* The OSCILLATOR STAGE: where the notes received from the INPUT STAGE are converted to frequencies which control two detuned Sawtooth Oscillators.
* The FILTER STAGE: where notes received from the INPUT STAGE are turned into an audio signal which sweeps a Voltage Controlled Filter, and where the audio signal from the OSCILLATOR STAGE is filtered.
* And the AMP STAGE: where the "bang" at the start of every note from the INPUT STAGE is used to trigger a message to the [vline~] Envelope Generator, which smoothly changes the volume of the audio from the FILTER STAGE.




## The Input Stage

At the INPUT STAGE, we use the [key] object to get the ASCII values of the computer keys being pressed. This information is passed to the [trigger] object. [trigger] is a very important PD object used to specify the order in which events happen.

What [trigger] does depends entirely on its creation arguments. When it receives any input, [trigger] sends messages to its output in a right to left order, based on these creation arguments. In this case, our [trigger] has the creation arguments "float", "float" and "bang". So on any input from [key], which sends a Floating Point Number (a "float"), [trigger] will first send the message "bang" out its right-most outlet, which will go the AMP STAGE. The it will send that float which came in to the center outlet, which will go to the FILTER STAGE. And finally it will send that float to the left-most outlet, which will go the OSCILLATOR STAGE. [trigger] is often abbreviated as [t], so the [trigger] in this example could also be typed as [t f f b].

For more information on [trigger], please see the Dataflow Tutorials in this FLOSS Manual.

## The Oscillator Stage

This stage is concerned only with the Additive Synthesis of two detuned Sawtooth Oscillators. This means that the output of two [phasor~] objects, whose frequencies are slightly different from each other, will be added together. Since the difference in frequency is quite small (one [phasor~]'s frequency is 99% of the other's), instead of hearing two tones we will hear a periodic modulation of one tone.

The float from the [trigger] in the INPUT STAGE arrives at an [mtof] object, which converts it to a frequency in Hertz. This frequency is sent immediately to one [phasor~], and also to a Multiplication [*] object, which makes a new frequency number which is 99% of the other, and this new scaled frequency is sent to a second [phasor~].

The audio output of the two [phasor~] objects is added together in an Audio Multiplier [*~] object, which reduces the overall volume by 50% to prevent clipping when it reaches the soundcard. The resulting audio

signal is sent to the FILTER STAGE.

## The Filter Stage

The FILTER STAGE is responsible for taking the audio from the OSCILLATOR STAGE and applying a swept Voltage Controlled Filter [vcf~] object to that signal. The center frequency of the [vcf~] is also determined by the key which has been pressed on the keyboard.

When the float sent by [trigger] from the INPUT STAGE reaches this stage, it is converted into a frequency number by [mtof]. This number is multiplied by 1.5 so that the center frequency of [vcf~] is a half octave above that of the Sawtooth Oscillators. The number from [mtof] is [pack]ed together with 300 and sent to a [line~] object. This message tells [line~] to ramp to any new number it receives in 300 milliseconds.

The audio ramp from [line~] is used to control the center frequency of the [vcf~] object. The result is that the [vcf~] will not jump to any new frequency it receives, but it will smoothly ramp there over 300 milliseconds, resulting in the distinctive "filter sweep" sound.

The audio leaving the Voltage Controlled Filter is now sent to the AMP STAGE.

## The Amp Stage

This final stage controls the overall volume of each note played by this synthesizer. It uses a [vline~] object as a complex Envelope Generator.

Every time a key on the keyboard is pressed, the [trigger] object in the INPUT STAGE sends the message "bang" to the AMP STAGE. Here it triggers the message "1 150, 0.9 150 150, 0 1000 500", which is sent to the [vline~] and tells [vline~] to make this audio ramp.

The exact instructions the message tells [vline~] are as follows:

* First ramp to 1 in 150ms
* Then ramp down to 0.9 in 150ms after a delay of 150ms from the start of the complex ramp
* After that, ramp down to 0 in 1000ms after a delay of 500ms from the start of the complex ramp

This translates to:

* Attack: 150ms
* Decay: 150ms to a value of 0.9
* Sustain: 200ms (the 500ms of the last ramp minus the 300ms of the first two ramps equals a "rest period" of 200ms)
* Release: 1000ms

With these instructions, [vline~] creates an audio ramp which smoothly controls the overall volume of the audio coming from the FILTER SECTION via an Audio Multiplication [*~] object.

## Subpatches

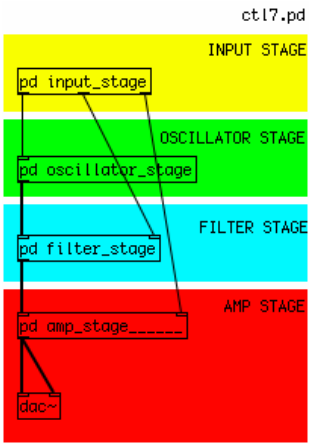
Now that we have an instrument that is separated into four distinct stages, we may want to make the screen a bit easier to look at by putting each stage inside its own Subpatch.

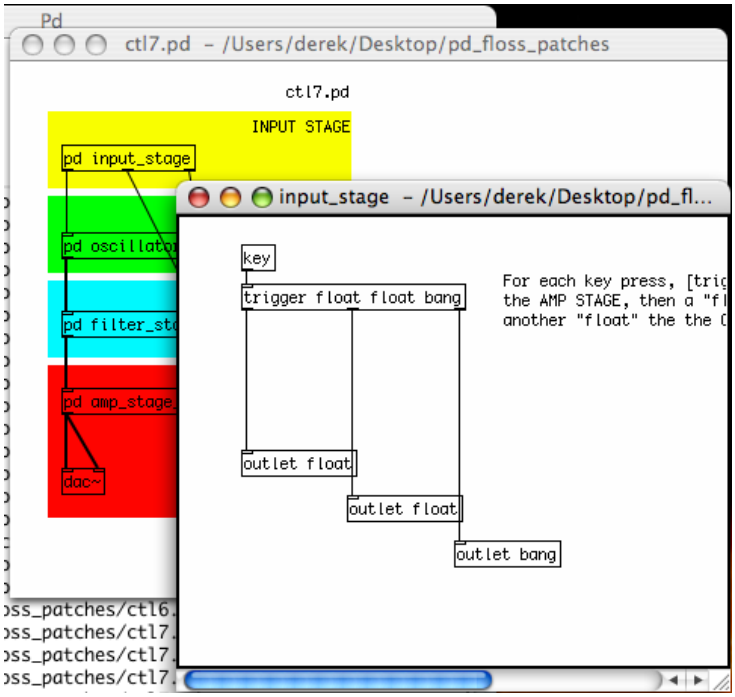
A Subpatch is simply a visual container which objects can be placed in to get them out of the way. To create a Subpatch in a PD patch, simply create an object named [pd mysubpatch], where "mysubpatch" can be any name you choose. A new empty patch window opens up and you can cut or copy and paste the objects you want to place in the Subpatch inside this new window. When you close the window, the objects will be inside

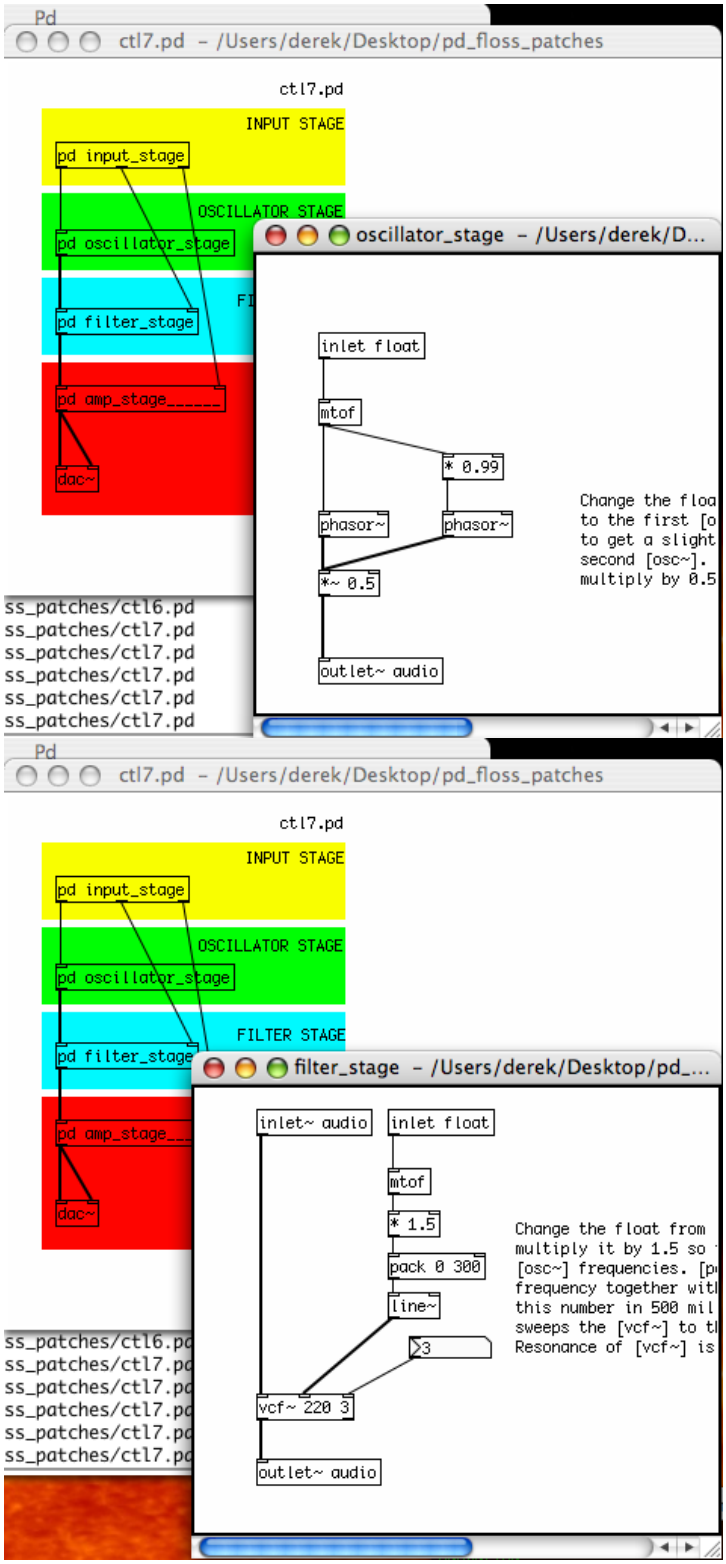
this Subpatch, still functioning as normal.

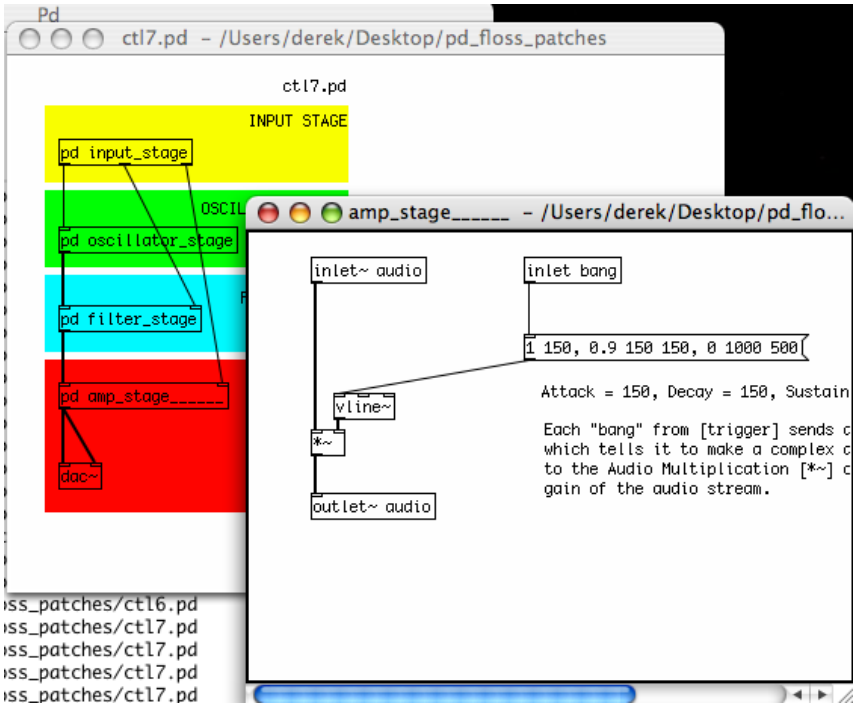
To get information in and out of a Subpatch, you can use the objects [inlet] and [outlet] for numbers and other messages, and the objects [inlet~] and [outlet~] for audio. This will create inlets and outlets in the Subpatch in the main patch you are working in, that you can connect as normal. You can give a creation argument to each inlet or outlet, which could be a reminder of what is supposed to come in our out of that inlet or outlet ("midi_note", "start_trigger", "audio_from_filter", etc etc).

Here is our Four Stage Subtractive Filtered Synthesizer, with each stage inside it's own Subpatch.




Pd
ctl7.pd - /Users/derek/Desktop/pd_floss_patches
ctl7.pd
INPUT STAGE
pd input_stage
OSCILLATOR STAGE
pd oscillator_stage
pd filter_stage
pd amp_stage
dac~
oscillator_stage - /Users/derek/D...
inlet float
mtof
* 0.99
phasor~
phasor~
*~ 0.5
outlet~ audio
Change the floa
to the first [o
to get a slight
second [osc~].
multiply by 0.5
ss_patches/ctl6.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd
Pd
ctl7.pd - /Users/derek/Desktop/pd_floss_patches
ctl7.pd
INPUT STAGE
pd input_stage
OSCILLATOR STAGE
pd oscillator_stage
FILTER STAGE
pd filter_stage
pd amp_stage
dac~
filter_stage - /Users/derek/Desktop/pd_...
inlet~ audio
inlet float
mtof
* 1.5
pack 0 300
line~
3
vcf~ 220 3
outlet~ audio
Change the float from
multiply it by 1.5 so
[osc~] frequencies. [p
frequency together wit
this number in 500 mil
sweeps the [vcf~] to t
Resonance of [vcf~] is


Pd
ctl7.pd - /Users/derek/Desktop/pd_floss_patches
ctl7.pd
INPUT STAGE
pd input_stage
pd oscillator_stage
pd filter_stage
pd amp_stage_____
dac~
amp_stage_____ - /Users/derek/Desktop/pd_flo...
inlet~ audio
inlet bang
1 150, 0.9 150 150, 0 1000 500
Attack = 150, Decay = 150, Sustain
Each "bang" from [trigger] sends c
which tells it to make a complex c
to the Audio Multiplication [*~] c
gain of the audio stream.
vline~
*~
outlet~ audio
ss_patches/ctl6.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd
ss_patches/ctl7.pd

# Streaming Audio with PureData

We shall look at streaming **mp3** to a streaming server using **Pure Data** ( PD ). You should have a running version of PD installed.

Additionally, you should have access to a **streaming server**.

If you have somebody that can lend you a server for this trial, then you will need to know to from them the following:

- what **mountpoint** do you use?
- the **hostname** or **IP Address** of the server
- the **password** for sending streams
- the **port number** of the server
- the **type** of server (Icecast2? Icecast1? Darwin? Shoutcast?)

## Creating the mp3cast object

Now create a new object and type **mp3cast~** :


If all is installed well the object will look like the above. If there is a problem the object will be surrounded by dotted lines, this means that the object couldn't be created.

## Add an osc~

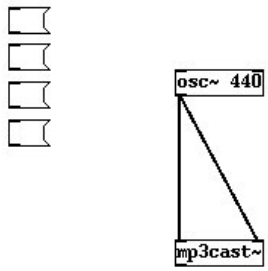
If all is ok, you can now add an audio object to the page so that we can send some audio to the **signal inlet** of the **patch**. We will use the **osc~** object.

The **osc~** object is created in the same way and it we will also give it a parameter. This parameter sets the frequency of the **osc~** sound wave, and we will use **440** (Hz). Then attach the **signal outlet** of **osc~** to the **signal inlet** of **mp3cast~**:


Now we have a mono input to **mp3cast~** but we want a stereo connection, so we will connect the same **signal outlet** to **right signal inlet** of **mp3cast~** :


## mp3cast~ Settings

We wish to create 4 empty **messages boxes**. Put them on your document like so:




Enter the following into these newly created **message boxes**. One should contain the following:

```
passwd
```

another should have:

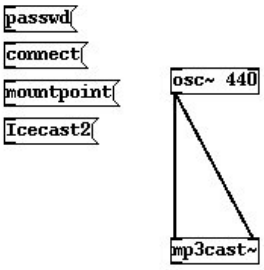
```
connect
```

the third should have:

```
mountpoint
```

and the last:

```
Icecast2
```

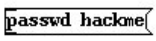



OK, so now we are ready to enter the details of our streaming server.

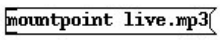
In the **passwd message box** type a **space** after 'passwd' and enter your **password**. In this example the I will use the **password** 'hackme', and I would type this:

```
passwd hackme
```

So I get this:

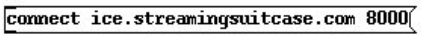



Then we enter the **mountpoint** in a similar fashion into the **mountpoint message box** . I will use the **mountpoint live.mp3**.




**note** : you do not need to enter the suffix **".**mp3" in the **mountpoint**.

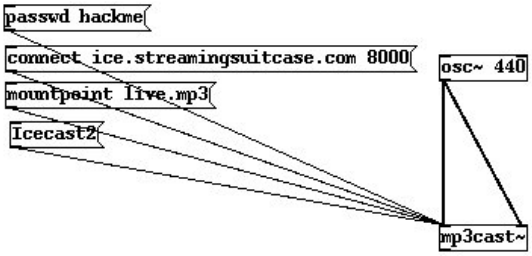
We also wish to enter the **hostname** and **port** of the streaming server. I will use the non-existant **ice.streamingsuitcase.com** as the **hostname** and **port 8000**:




**note** : do **not** put in the leading http:// in the **hostname**.

Lastly, we have the **Icecast2 message box**. This defines what kind of server you are logging into. If you are using an **icecast1** server you would instead have **Icecast1** in this box. Similar for **Shoutcast**. If you are streaming to a **Darwin** server use **Icecast1**.

Connect all the **control outlets** from these **message boxes** to the **left control inlet** of the **mp3cast~ object box**. You may have to move the boxes around a bit to make space :




# Start the Stream

Now, to start to stream you must goto **run mode** and press the boxes in the following order:

1. press **passwd** to set the **password**
2. press **Icecast2** (or whatever server type you are using) to set the **server type**
3. press **mountpoint** to set the **mointpoint**

Now...this process has loaded **mp3cast~** with the server settings. Click the **connect message box** and you should be streaming!

To test connect with your favourite player using the the following syntax :

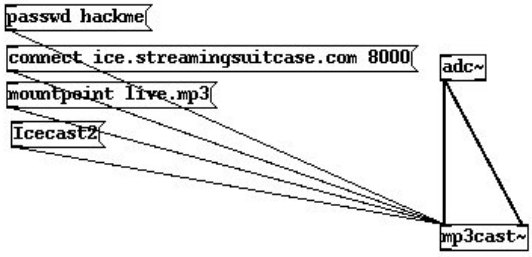
```
http://hostname:port/mountpoint
```

In my case this would be:

```
http://ice.streamingsuitcase.com:8000/live.mp3
```

# Streaming from The Mic

Lets replace the **osc~** with **adc~** like so:



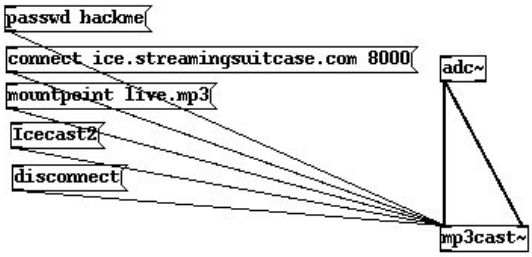

The **adc~** object takes the input from your computers sound input. **adc** is short for **Analog Digital Converter**. If you now stream the sound will be coming from your soundcard input!

Incidentally, if you need to disconnect the stream make a new **message box** , type:

```
disconnect
```

then connect this to the left **control inlet** of **mp3cast~** , return to **run mode** and press it.

# Disconnect



# Dataflow tutorials

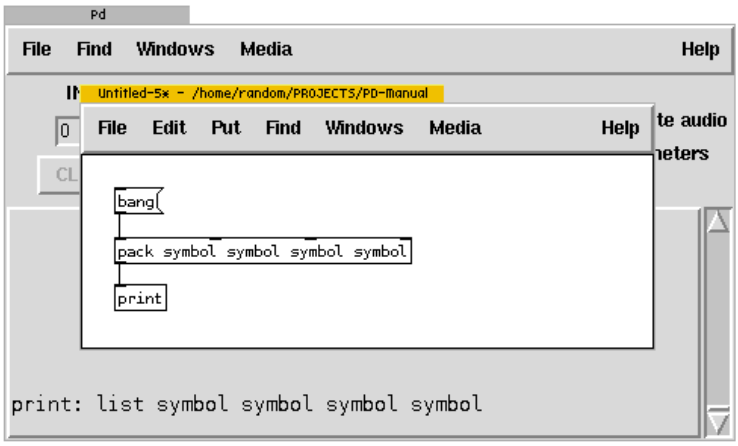
While a PD user (which is, lovingly enough, a PD programmer at the same time) can learn how to use it just by playing around and trying new things, there are important functionalities that are not immediately apparent through play, trial and error. It may take some time to understand underlying functionalities of objects and messages. Following tutorial(s) try to explain and practically demonstrate in quick simple way some of the more important 'grammatical' aspects of this patcher language graphical programming environment.

All examples in this tutorial(s) are available as PD patches too. As some of them demonstrate certain functionality much more vividly when opened inside the Pure Data it is recommended to download them and try them out while reading the tutorial. Get the zip here:
http://en.flossmanuals.net/floss/pub/PureData/DataFlow/DataFlowTut_patches.zip

These tutorials can be used in two ways: they can be followed from start to finish (there is slight gradation of difficulty of material and examples), but they can be accessed also as some kind of simple reference. So if something is too obvious to the reader, she can skip a section or two (or just check the screenshots).

## Messages

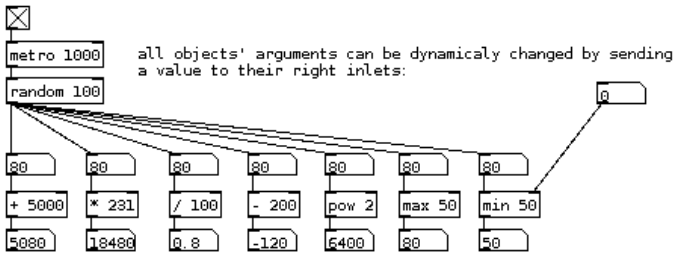
In the same way as basic atoms of a spoken language are words, PD's objects intercommunicate using messages. When data is "sent" from outlets, "travels" along the connections and is "received" at the inlets of objects in the patch it is "understood" or "decoded" by objects in a specific way. Apart from audio signals most (all?) other data is known to be "messages". As with nouns, verbs and pronouns, PD messages can be of different type: numbers (also known as floats - floating point numbers), words (known as symbols) and lists (of numbers and/or symbols), to name the most frequent ones. To help an object to understand what kind of data it is receiving a word called "selector" is frequently present before each data.




A [print] object is PD user's best friend when it comes to determining data types, or rather, their selectors. In above simple demonstration [print] prints a selector "list" before four symbols that are packed by [pack] object which receives special message *bang*, which is an instruction that is understood by all objects as a "do it!" command. Apart from bang message, if a message has no selector before it, it is assumed that message is a numerical value (PD doesn't distinguish between floating point and integer numbers - all are floats). In the above example, therefore, [pack] expects four symbols in its inlets and will report error if incoming data will not have correct selectors. It is possible to convert data types with objects like [trigger] and [symbol].

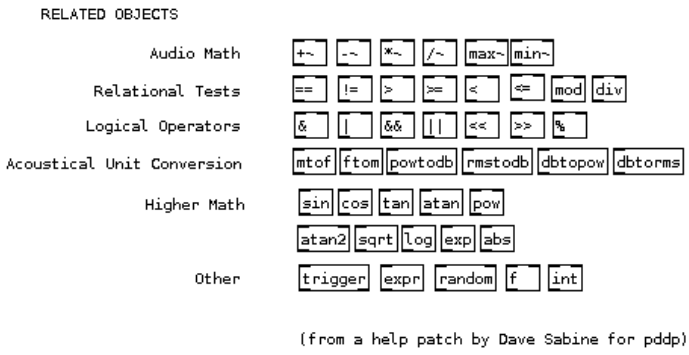
## Math

Numerical values (or streams of them) can be mathematical manipulated with numerous arithmetic objects in PD:




While it is possible and quite normal to build equations using multiple basic math objects in PD, there is very useful object called [expr] which offers many possibilities with more complex computations, where connecting many boxes is less convenient then writing a formula into [expr].

<examples of [expr]???>



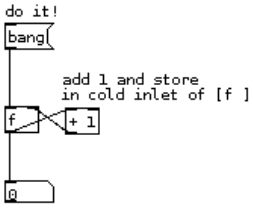

But first some truly important dataflow basics.

# Three little bits - temperature, order and depth

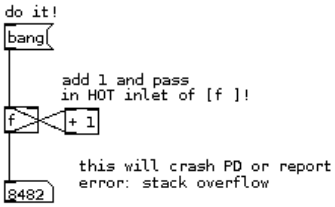
Once upon a time there was a PD object. Despite being a somewhat virtual construct by a certain Puckette, it had, curiously enough, a hot and cold side. Not only that, it demanded to be triggered from the right side and was extremely pedant in finishing what was started. Or was it? Not sure what this all means? Read on!

## Inlets: hot and cold

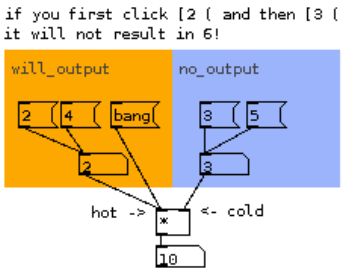
Because PD is not a traditional text-based programming language, scheduling actions at the inlets and outlets of an object is done with some special rules which need to be kept in mind before they become 'natural'. How and when processing and output of the object is triggered by input is defined by a 'hot and cold inlets' concept: the first leftmost inlet of any object is always a hot inlet. Whatever an object receives on hot inlet it triggers processing and output from the object. All other (right) inlets are cold inlets. Whatever the object receives in them, it stores that value, but *does not output*. This can be seen at work with a simple counter example:




A message "bang" to a hot inlet is a special message that causes an object to process and produce output using previously initialized, set or default values ([*float*] for example outputs 0 if nothing has set its value before). At its cold (right) inlet [*float*] object stores the result from addition object and does not output before it receives anything at hot inlet. In other words, when sent a "bang" message, float] sends a value (0 or anything stored from before), to this a 1 is added in [+ 1] object and result is sent to the cold inlet of [float], where - because it's a cold inlet - this value is stored until next bang or other action at its hot inlet. This is why above construction does not produce an endless loop (and crashes your PD session or report stack overflow) unlike an example below:

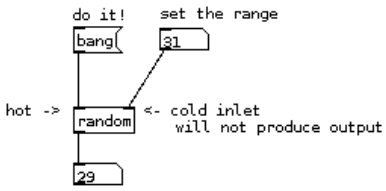



A simple example in which it is easy to see hot and cold inlets at work is also any basic math object [+ ], [* ], [/ ] or [- ].

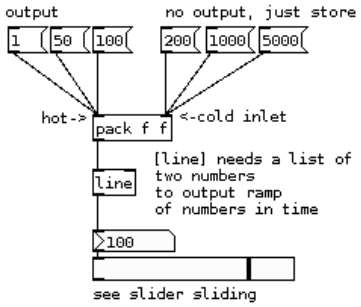



In the above example it is necessary to send a value or a bang to hot inlet *last* in order to receive correct result.

Similarly the [random] will not output anything unless banged on its hot inlet:

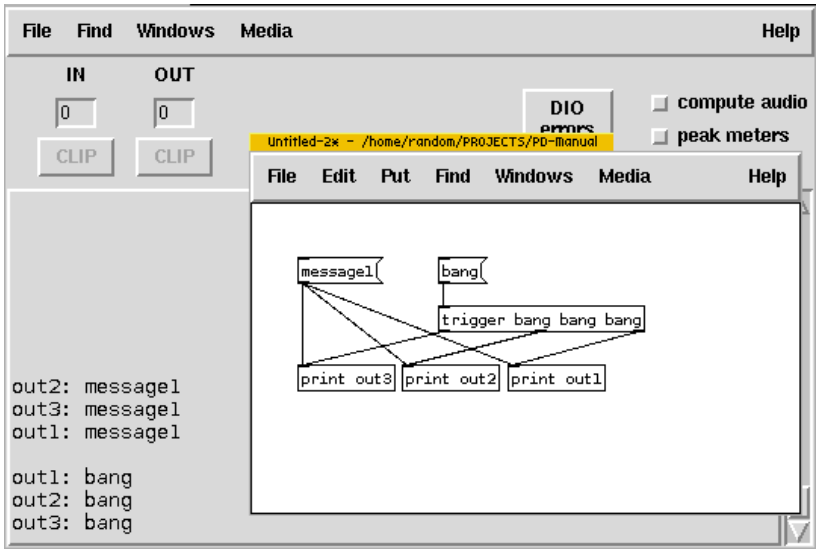



Another example of hot/cold inlets is [pack] which packs individual messages into a list. In the following example a [line] object produces ramps of numbers in time. The list of two object received by [line] determine the target value and time interval within which target value has to be reached.



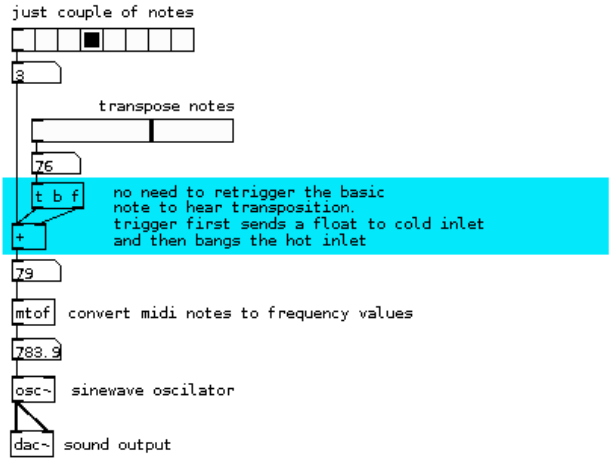

## Order of connecting and [trigger]

While multiple incoming connections to the same inlet are rarely problematic, care has to be taken when order of operations is important and when making multiple outgoing connections from a single outlet to various inlets in a patch. The order what goes out sooner or later is determined which connection was made before or ofter some other connection. This is later invisible in the patch itself (unless one inspects saved patch as a text) and is therefore discouraged in order to produce readable code. When copies of data are needed [trigger] is programmer's friend. Trigger takes a single incoming data, copies and converts it according to its arguments and outputs the copies through its outlets in order from right to left. Here's a brief demonstration (from which one can even figure out the (non visible) order of connecting with which message1 was connected to print objects:



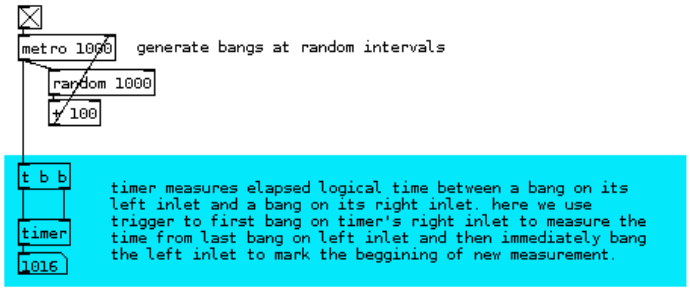

A simple practical application of hot+cold and trigger would be the following simple control of frequency of oscillator:

<explain in-line, rather than only with comments?>



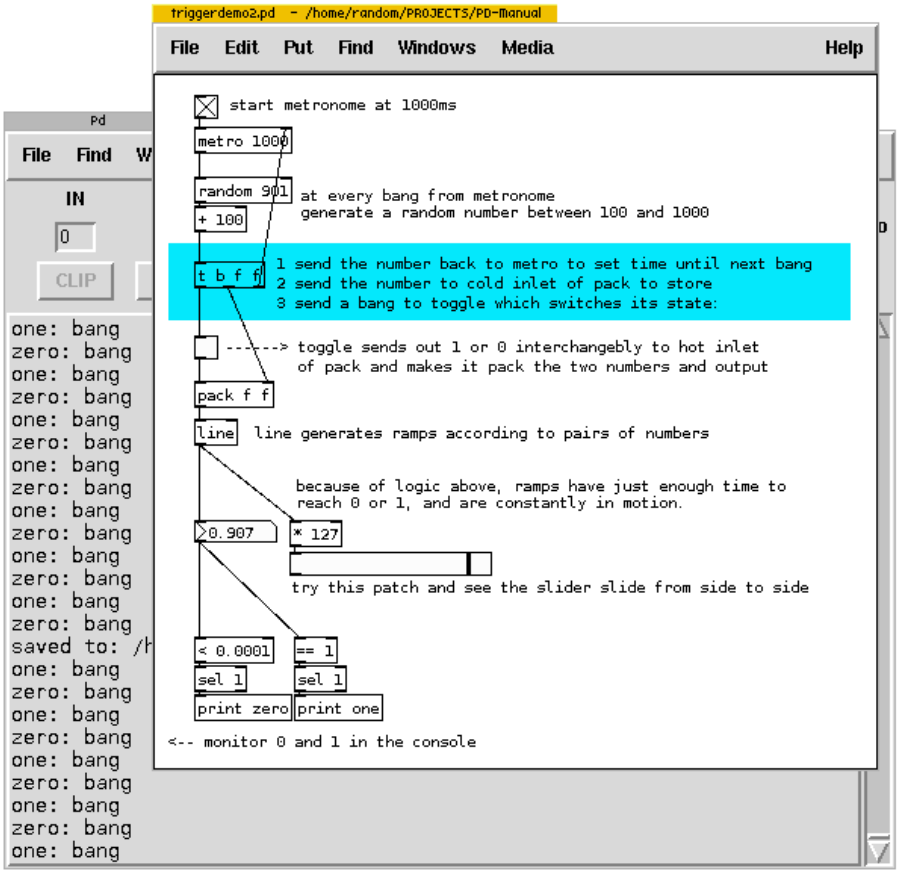

Consider this simple example of scheduling two bangs at inlets of [timer] object to count intervals between single bangs:

<explain in-line, rather than only with comments?>




Another practical application is slightly more complex generation of ramps from 0 to 1 and back whose speed varies in time, it never stops and it always reaches 0 or 1:

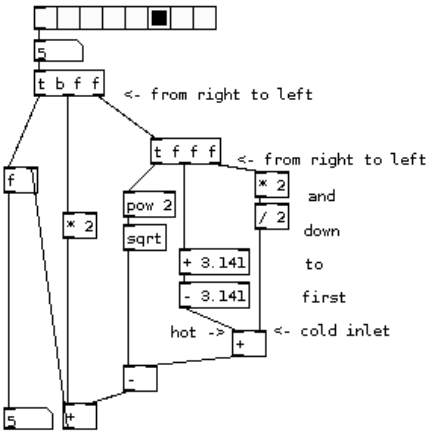
<explain in-line, rather than only with comments?>




## Depth first message passing

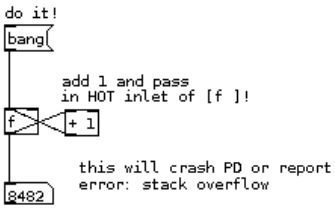
There is one more rule of Pure Data programming that is important when scheduling of events and order of execution becomes crucial to your code. Also called 'depth first message passing' the rule states that at a forking point (as in [trigger] or multiple connections from a single outlet) a single scheduled action is not "finished" until its whole underlying tree is "done". In other words, an action will send down all messages it

needs till it arrives at either cold inlet, end of chain of active messages or hot inlets or similar. To put it in even simpler terms, the message will first go all the way down the well until the next scheduled message down an other well. To imagine better what this means consider this simple demonstration example, which at first sight might look intimidating, but it's nothing more difficult than following with the finger the path through the labyrinth. Try it and remember the trigger's right to left order and depth first rule:




The resulting number will be always the same as the input number as the scheduling logic is taken care of according to rules we defined so far.

Consider again the wrongly connected counter example that can crash your PD session (or report stack overflow) because of infinite loop:




From the point of view of depth, the above example represents infinite depth - the message passing is never finished.

In conclusion of these three important dataflow rules it must be said that

1. hot and cold inlets
2. order of connecting + use of trigger object, and
3. depth first message passing

are in most cases intertwined and dependent on each other. In other words, these concepts and its practical applications appear constantly in the PD code. In this sense, they form some of the most basic rules of this language.

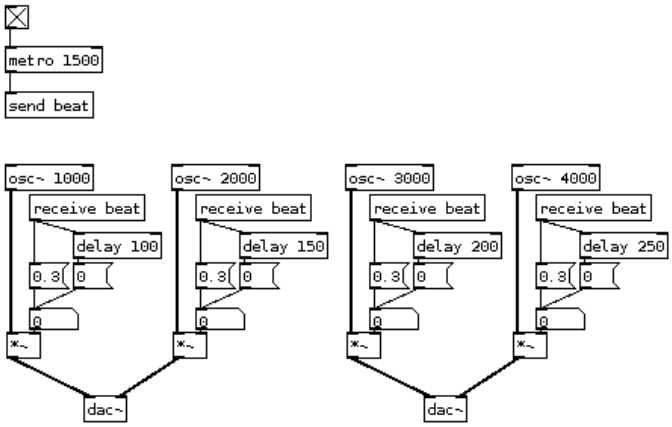
# Invisible connections, crossing borders, reusing code

While the main strength of PD seems to be its graphical/diagramatical nature where connections between objects represent actual path of data, it is only when data can be be exchanged between different windows, patches and applications and also when code can be abstracted and reused in multiple instances it is possible to create truly complex programs suited to one's needs.

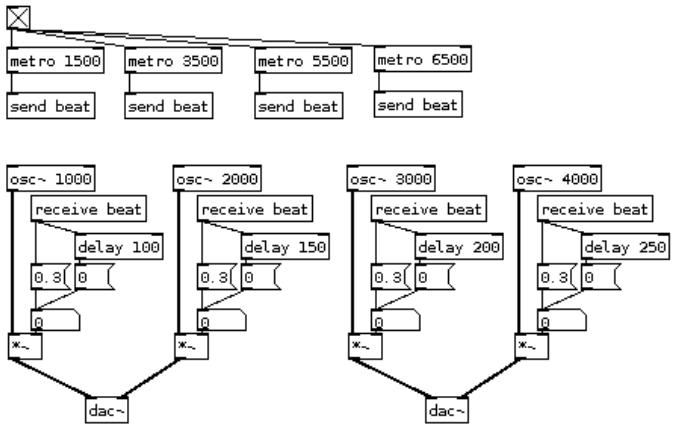
## Send, receive, throw and catch

Soon after some introductory patching, a PD user will soon discover the slight inconvenience of connection lines running over objects to reach other objects. For a while, with some imagination this tendency towards mess can be avoided, however when serious and more complex patching starts to take shape inconvenience turns into a drawback. Luckily, there's a solution.

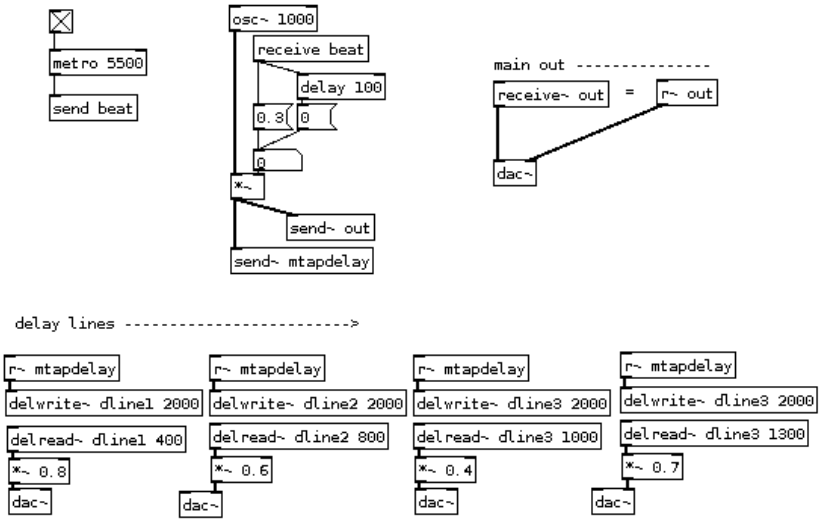
Using [send] and [receive] data can be sent from one part of the patch to another (or even into another window) without connecting lines. Both objects need an argument that tells from which [send] does a [receive] receive data. In this way we can use more [send]s and [receive]s. In the following patch




[metro 1500] generates bangs at the interval of 1.5 second (1500ms) and is sending them to [send beat] object. These bangs are picked up by [receive beat] objects because [send] and all [receive]s have the same argument - "beat". An argument required by [send] and [receive] can be arbitrary (but not numeric only). The rest of the code (osc, delay, dac) produce short beeps. Try it to hear it. There is no limit in number of [send]s and [receive]s with same argument. It is possible to have many sends. Just add to the example above more [metro] objects:

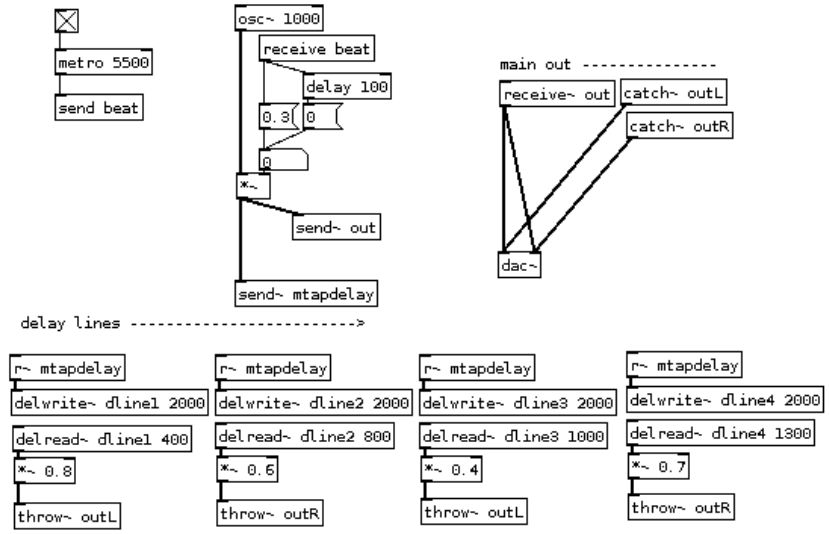


[send] and [receive] are for control data - messages, symbols, lists. For audio signals a 'tilde' version of these objects are needed. [send~] and [receive~] can be used to receive a single audio signal at many places. The slightly more complex example would be to create a multitap delay:



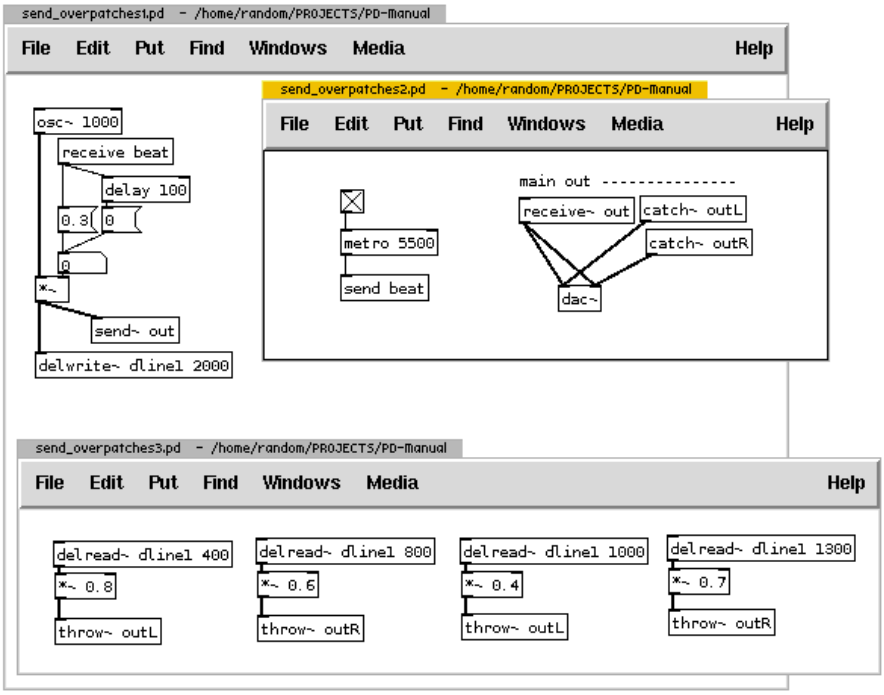

To consider above example: taking a [send beat] from previous examples and a single generator of a beep an audio signal is sent with a [send~] to 'mtapdelay' and 'out'. The latter is received and send to dac~ for immediate output (dry signal). Beeps are received also by [r~ mtapdelay] ([r~] being a working abreviation for [receive~] - there is no difference in functionality) and fed into four separate delay lines (notice dline1, dline2, dline3... and differing delay times in delread~ objects) and sound is also attenuated with [*~ 0.4] and alike before send to one of the two channels representing a stereo output.

Notice however that outputs from delay lines are not sent with [send~] back to [r~ out] for example. A reason behind that is that with audio signals there can only be one [send~] for many [receive~]s. While there are technical reasons for this difference, a handy pair of audio objects that help to achieve many-to-one sending are [throw~] and [catch~] which work in the sense of a summing audio bus: many [throw~]s can send audio signals to one [catch~] that simply sums all the signals:



Using [catch~] it is then possible to further control and process audio (i.e.: volume control, VU metering, limiting, reverbs, etc...).

Coincidentaly, all objects we described above ([send], [receive], [send~], [receive~], [throw~], [catch~], as well as [delwrite~] and [delread~]) all work across different patches, subpatches and abstractions (the latter two are explained in next sub-chapter). To build on the example above, the simple matrix of delay-lines could be in a separate patch. Furthermore, it is actually not necessary to have four delay-lines to achieve multitap delay. There can be only one [delwrite~] and more [delread~]s that read from that delay-line at different delays:

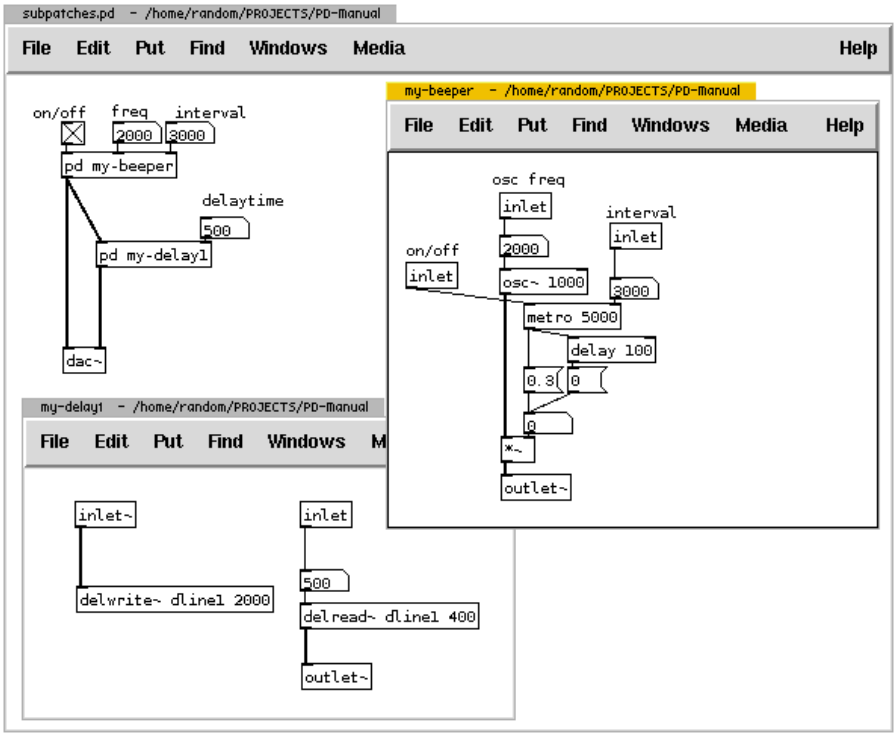



In conclusion, the objects described above are powerful tools  to not only send and copy data and audio around a single patch without messy connections, but to create connections between individual patches, subpatches (patches within patches) and abstractions (reusable "classes" of which multiple instances can be created) as well. A word of warning though: the arguments passed to these objects are always global - they are accessible from all patches and abstractions opened in a single PD session. This simply means that a situation can arise with unwanted 'crosstalk' of data or multiplies defined. Care has to be taken on names of arguments, while at the same time a technique exists to localize arguments using dollarsigns (see chapters on abstractions and dollarsigns).

## Subpatches

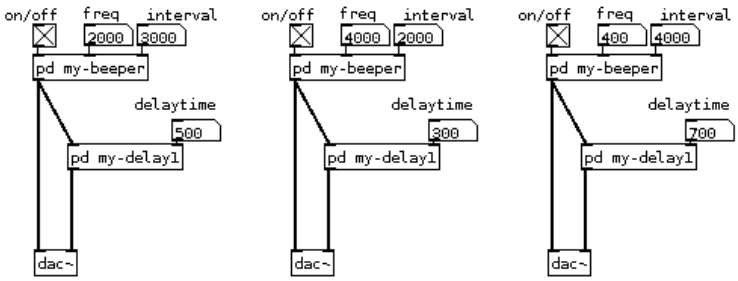
After a PD user discovers and learns how to use [send] and [receive] she is spared of messy criss-crossed patches, but not for long. With even more complex coding, user would either store code in different individualy saved patches - but would then have to open them separately, or expand its canvas to unscrollable size. However, there is a solution to 'hide' or store parts of a patch in a so-called subpatch.

It is useful to think of subpatches as container or drawers, where code is organized and put away. A subpatch is created by typing [pd any-name] into an object box, where "any-name" can be arbitrary word. When creating subpatch like this, a new empty subpatch window will appear. Consider the following practical example:




Subpatches can also have inlets and outlets (for data and audio respectively) which are created by using an object [inlet] or [outlet] (and [inlet~] or [outlet~] for audio signals) inside a subpatch. Example above uses all these. In a single patch saved as subpatches.pd a subpatch called "my-beeper" is created that contains a bit of code from previous examples - a simple switchable beeper, whose frequency of a beep and time interval can be set - from without the subpatch itself using [inlet]s. The horizontal order with which they appear in a subpatch (a little window on the right side) determines the corresponding order of inlet at the object box. "my-beeper" subpatch produces audio signal which is fed into corresponding [outlet~]. A copy of this signal is sent to one of the output channels (presumably your left speaker) and into "my-delay1" subpatch which contains simple code of and audio [inlet~], message/control [inlet] and a delay-line. This delayed audio signal is sent to the other output channel - presumably your right speaker.

By closing the windows of subpatches the code is not lost but still exists inside the subpatch objects ([pd my-beeper] and [pd my-delay1]). They are saved within the patch subpatches.pd. Subpatch windows can be reopened by left-clicking on subpatch objects or by rightclicking and choosing "Open" from menu. Subpatch objects can be freely copied, by which unique copies are created, that can be individually edited - changes are not reflected in any other subpatches, even if they have the same name.

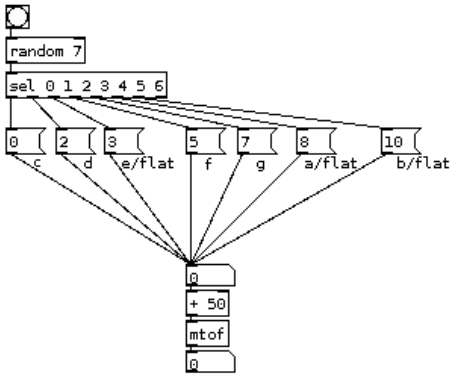



Care has to be taken, however, as in above example, subpatch my-delay1 uses delay-line called "dline1" and copies of that subpatch should not all use the same delay-line, but rather each should use it's own one (i.e.: "dline2" and "dline3"). If there is two or more [delwrite~]s that write to the same delay-line PD reports "multiply defined" and delays do not work. It also important to remember that arguments in an object name cannot be passed to subpatches - unlike in abstractions.

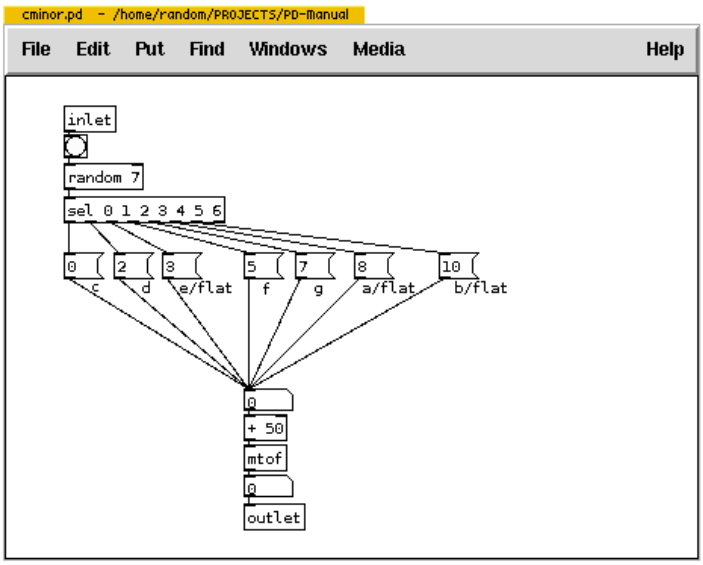
## Abstractions

Subpatches are useful to put away unique piece of code from the main canvas, to store functionality specific to the opened patch or to have similar but slightly different pieces of code that can be edited separately. However, sometimes precisely the same code is used again and again at different times in different patches, or same exact construct needs to be used multiple times, in which case it would be less convenient to create copies of subpatches, especially when this construct is improved and then improvement should be reflected in all subpatches. In all these cases it is much more useful to abstract this code, to make it available from the outside of the patch, so it is reusable by calling it from within as an instance. A possible analogy to this would be a radio broadcast - it is produced live in a radio studio, but many instances of it are heard across the country. Changes to it at the original location are reflected at all instances.

Consider a situation where a random note on minor C scale converted to frequency is needed multiple times in one patch. A basic construct for this would be:

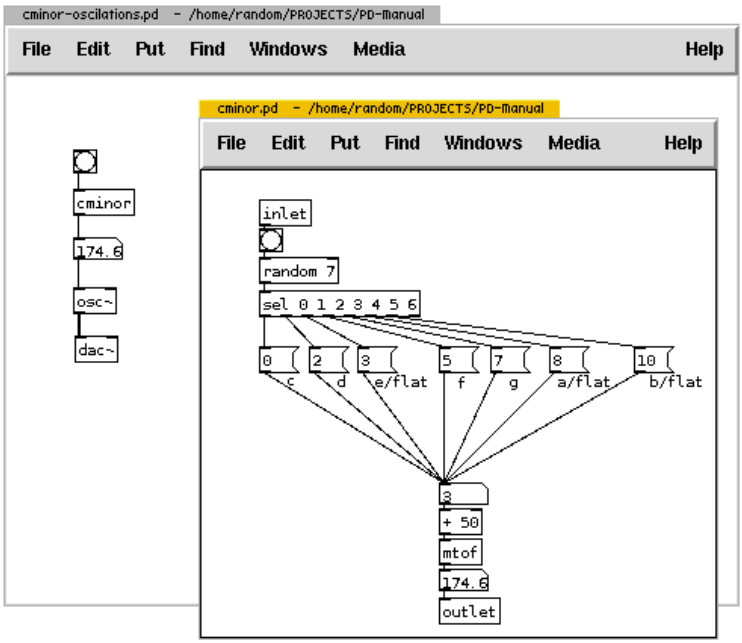



Every time [random] is banged, one of the displayed numbers will be transposed + 50 and through [mtof] converted to frequency. It's a construct that's inconvenient to reproduce many times in a patch. To abstract the code, it should be handled like it's a subpatch and inlet's and outlets are added but should be saved as a separate patch:

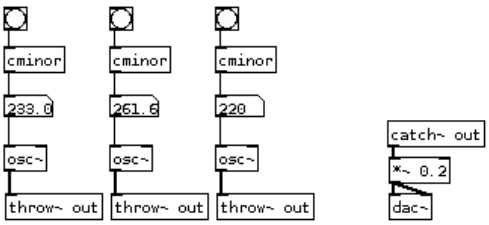



This patch needs to be saved on a path (folder) that PD looks into each time an object is created. That path (folder) can also be defined in PD preferences however the simple usage is to have this patch in the same folder where the calling patch is saved - a patch from within which this abstraction is called. Consider a main patch "cminor-oscilations.pd" saved in /home/user/puredata/ (or c:/pd-work/) and "cminor.pd" in the same

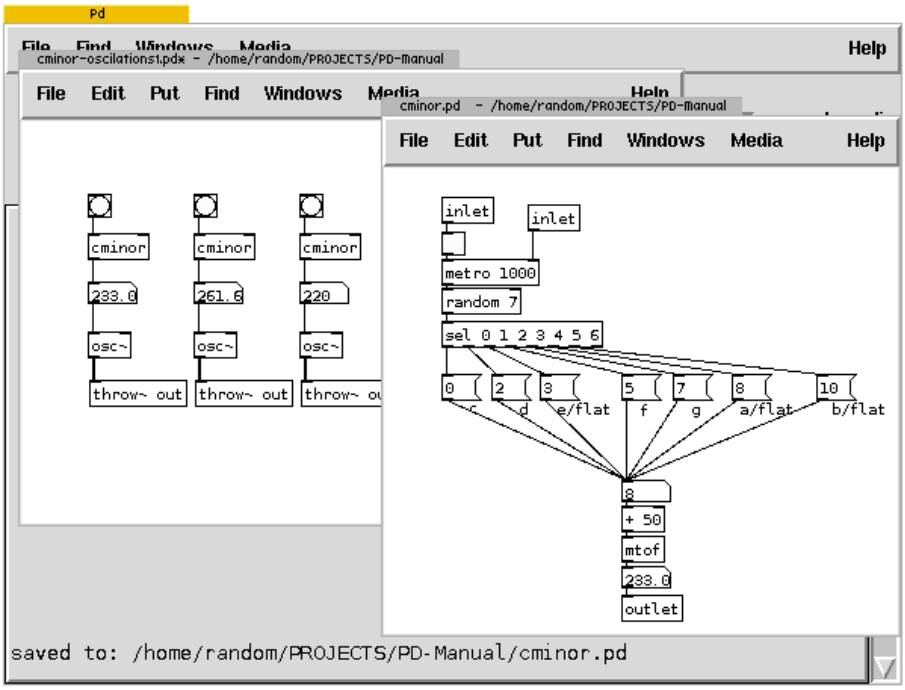
folder. The abstraction (or an instance of it) is called simply by typing the name of the patch (without extension .pd) into an object box, like this:




By clicking on the [cminor] (or rightclicking and choosing "open") the abstraction is opened in new window, just like subpatch. Alike, its inlets and outlets are defined by [inlet] and [outlet]. However now a separate patch (cminor.pd) is being edited. This means when changes are saved all instances in the calling patch are updated. In an example with more instances:

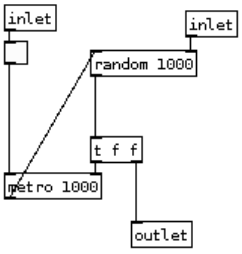



an inlet is added to [cminor] abstraction by opening it, adding code for "automatic" change of frequency and saved. This is immediately reflected in all instances by two inlets appearing on them:

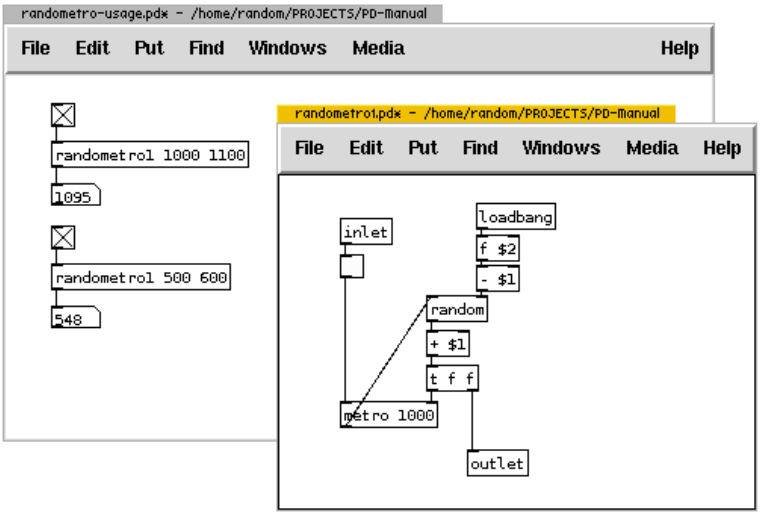



## Dollarsigns

In the same way as objects like [metro], [random] or [osc~] can (and need) to accept arguments (as in [metro 1000]) an abstraction can accept arguments that can be used inside of it. Consider an abstraction that combines [metro] and [random] objects to produce random numbers that also denote time intervals at which the are produced. In its basic form it could look like this:

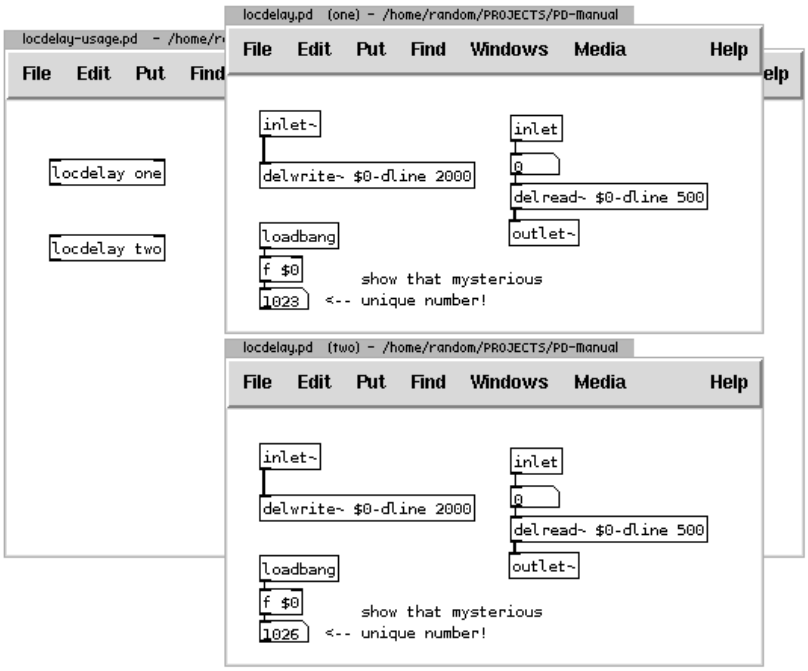



The abstraction above has two inlets, at left it would receive on/off (1/0 float) input and at right the range for the [random] which will then pass random numbers to [metro] and at abstraction's outlet. As it can be seen, the abstraction will initialize with 1000ms to [metro] object and range from 0 to 1000 to [random] object. To change the value of random object dynamically that value will have to be send at abstraction right inlet. However, this can be done differently by passing arguments to the abstraction at the creation time using dollarsigns inside the abstraction. Consider this change including demonstration of usage:




At the creation time two arguments are passed to an abstraction [randometro1]. Inside the abstraction, $1 is substituted with the first argument, and $2 with the second. The effect (which was goal in the first place) is to be able to define the min-max range (as opposed to only 0-max) at which abstraction works. Because [random] inside the object needs a 0-max range, first argument (presumably smaller) is subtracted from the second. The result is passed to random to produce random numbers which are then added to the first argument. In demonstration of usage in the window behind the abstraction this construct produces random numbers between 1000 and 1100 in the first case, and 500 and 600 in the second.

While $1, $2, ... and so on represent first, second, etc .. argument to the abstraction, there is one special dollarsign that is in Pure Data extremely useful. $0 is a variable that is internally substituted by unique four-digit number per patch or instance of abstraction. In other words, PD takes care that each instance of an abstraction or patch will be assigned this unique number and stored in $0 variable. The usefulness of this is immediately apparent in the following example of simple delay abstraction where delay-lines with the same name in multiple instances of same abstraction must be avoided:

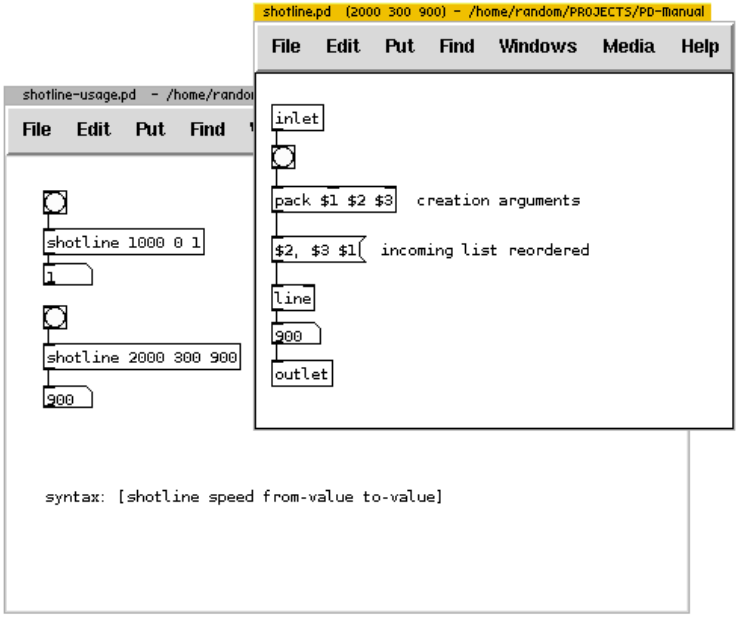



It is important to understand that, despite $0 isn't actually substituted with the unique number inside the delwrite~ object, the latter actually writes audio signal to delay-line named "1026-dline". $0 variable is

assigned in every opened or called patch, which also solves the problem of two or more instances of same patch (i.e.: simple synth). $0 also saves from situations from unwanted crosstalk of frequently used variables in different patches. An attentive reader/user could also point out a possibility to use $1, to use an argument passed to an abstraction (like "one" and "two" in above example), in which case care must be still taken to assign unique arguments to abstractions used in the same PD session.

$0 is at times called localized variable, however, in my view, that is not entirely true. A variable constructed with $0-something can still be accessed from the global namespace by simply finding that unique number and than calling that appropriate variable (like for example to read the delay-line named 1026-dline from above example from within another independent patch). In fact this can sometimes be even useful. It is however true that using dollar variables is a localization technique.

A frequent confusion arrises from the use of dollarsigns in message boxes. It is important to understand that dollar variables in message boxes are actually totally local to that message box itself regardless where they appear. They will be substituted only by what a message box receives on its inlet. In an example of abstraction within which both types of dollar variables are used:



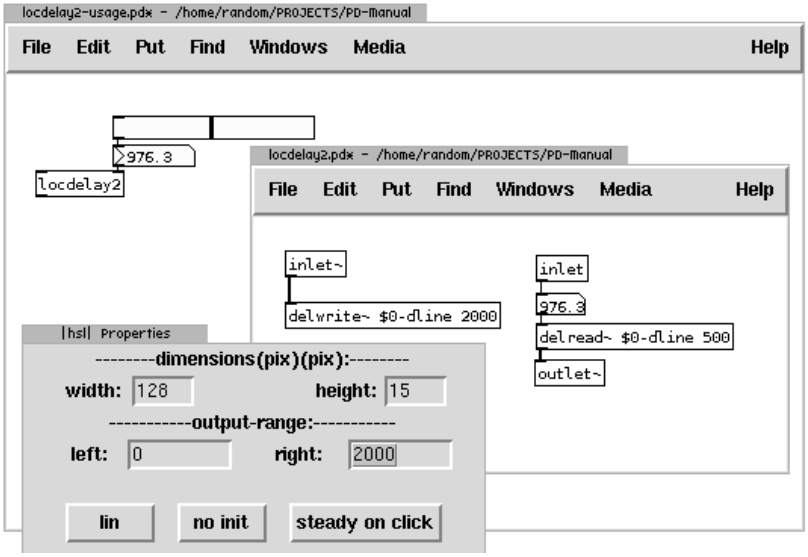

[shotline] abstraction, which has a goal of producing a ramp of values in specified time from some starting value to ending value, takes three arguments - speed, from-value and end-value. These variables are accessed inside the abstraction with $1, $2 and $3 in the [pack object]. The latter sends a list of those three arguments to message box, in which $1, $2 and $3 *represent only elements of an incoming list and not directly arguments of the abstraction*. Message box first send the second element, followed by a comma - so it resets line to that value, and then a pair of third and first element which correspond to target value and time-frame of a ramp.

# Pretty interfaces and two-dimensional data

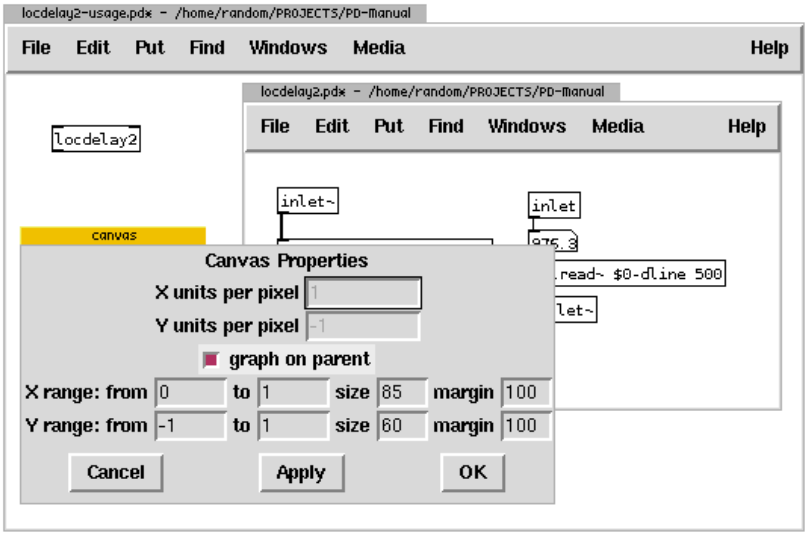
## Graph on parent

In pure data it is extremely easy to create interfaces that include sliders, buttons, number boxes, toggles, coloured backgrounds... how to use them, look at the "GUI objects", or simply right-click>help on one of them. However they still need to be connected and to use them away from the data inlets that they control, they have to be repeatedly created in order to function the way we want. Consider an example of a delay abstraction (already used above) that takes at it's second inlet a value for time of delay which we want to

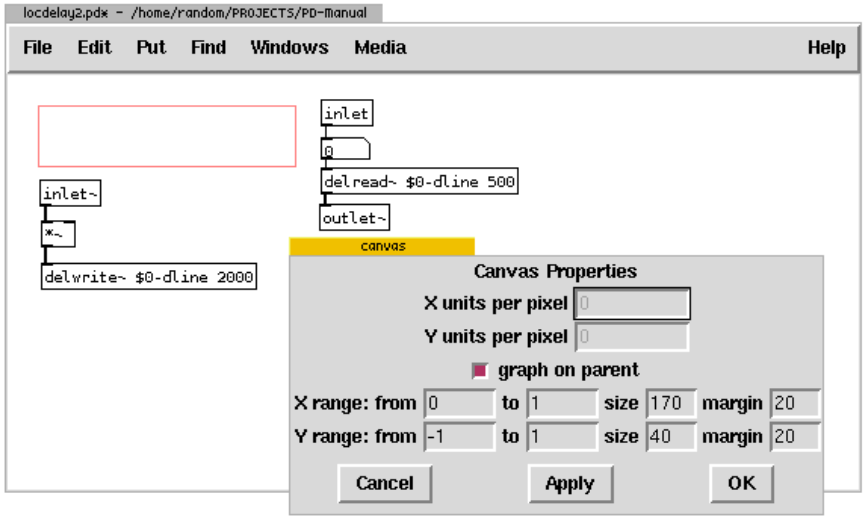
control with a slider:



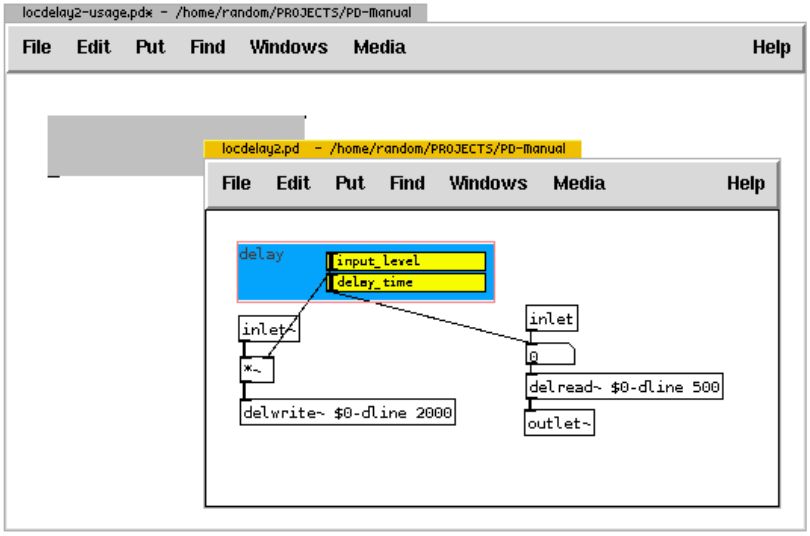

So, everytime when an abstraction like that is created, when it is desired to be controled by a slider, many steps are needed to recreate the same visual and programmatic construct. Luckily, there is a very powerful feature od PD: **graph-on-parent**. It enables a subpatch or an abstraction to have a custom appearance at the parent 'calling' patch. Instead of plain object box with the name of abstraction and arguments, it can have different size, colour, and all the gui object inside. Here's how it's done, continuing on delay: inside the abstraction or subpatch, rightclick on white underlying canvas and choose properties. Inside a dialog that appears, enable toggle for graph-on-parent:



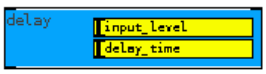

Applying this will create a grey-bordered box within the abstraction. This box represents the shape and form of the abstraction on the parent canvas (the calling patch). Whatever the size and contents of that grey box will be visible excluding connections, object boxes and message boxes. In the properties of the abstraction below the graph-on-parent option two rows of four values represent X and Y settings. Size will set the size of the box while margins will only set the position of that grey box within the abstraction. Adjusting these setting accordingly:



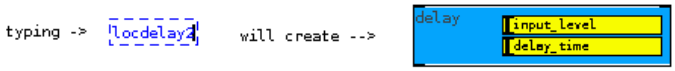

Inside the grey box it is now possible to create a suitable interface, according to users needs and aesthetic preferences needed for functional and pleasurable control of parameters. See properties of individual GUI objects (like canvas, slider, etc) and experiment what can be done with them. Simple delay abstraction in this case receives an underlying colour canvas and two sliders, one for delay-time and the other for incoming level:



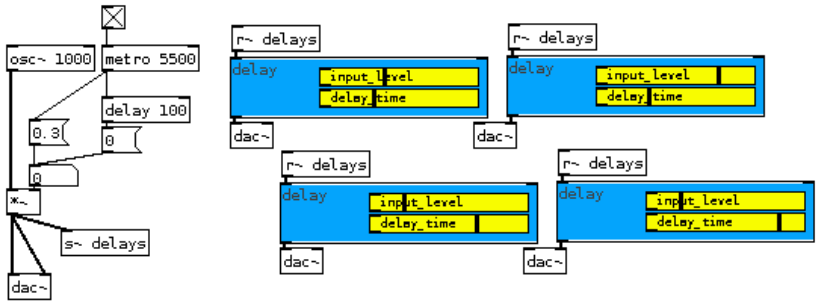

While editing the abstraction with graph-on-parent, abstraction is greyed-out on the parent canvas until the abstraction window is closed. Only then the final appearance can be seen:




The purpose of a pixel wide transparent gap between the gray border and canvas in the abstraction is to reveal inlets and outlets at the parent window - however with sizing of inlaid canvas, even black borders can be hidden. Calling this abstraction as usual - by creating an object box and typing the name of abstraction without the extension .pd - will always instantly create this GUI:



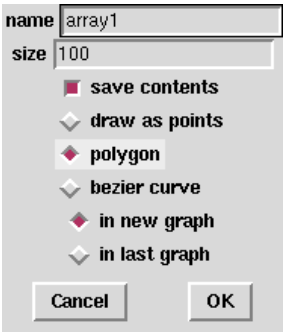

that needs nothing more than to connect to audio signals and adjusting controls:



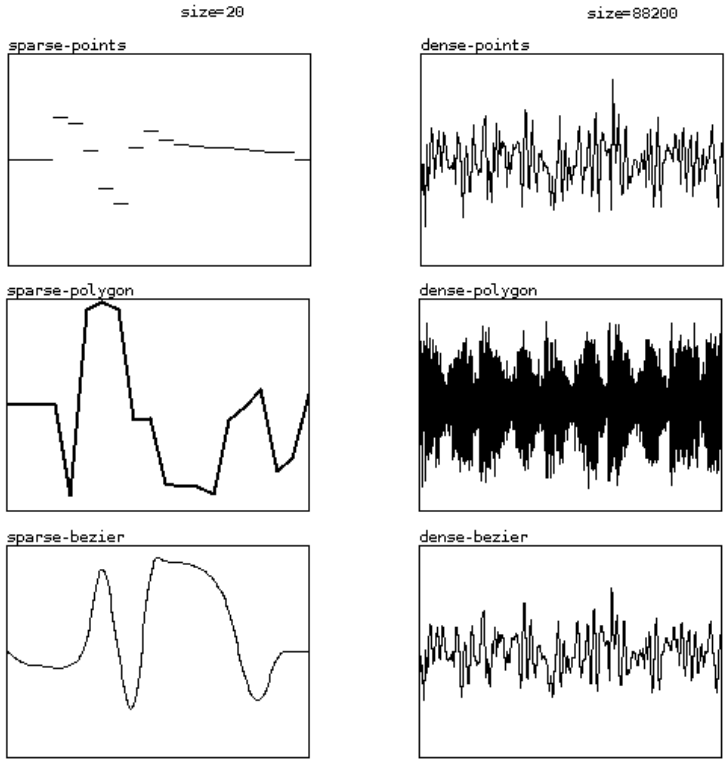

## Arrays + graphs = tables

In programming and building applications - and patches in PD are many times some kind of applications - we frequently need a way to conveniently store bigger amount of data and to be able to instanly access it. To store data in float boxes or dollarsign variables is inconvenient when more then four, five or ten numbers need to be stored. An "array" can be thought of as a container in a computer memory with neatly indexed drawers with data that can be looked up instantly. Arrays are visualy contained in graphs with X and Y axis. In PD arrays contain only numbers and together with graphs encapsulate the concept of tables. In other words, tables are graphical arrays that contain numbers.

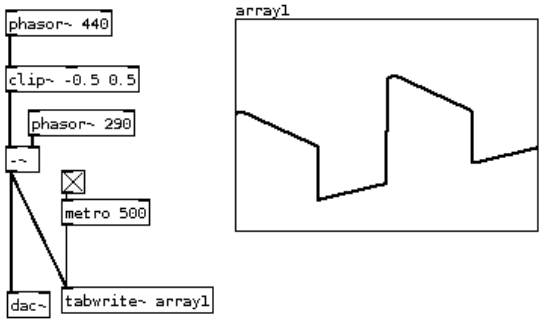
To create a table choose "Array" from "Put" menu and a dialog appears:



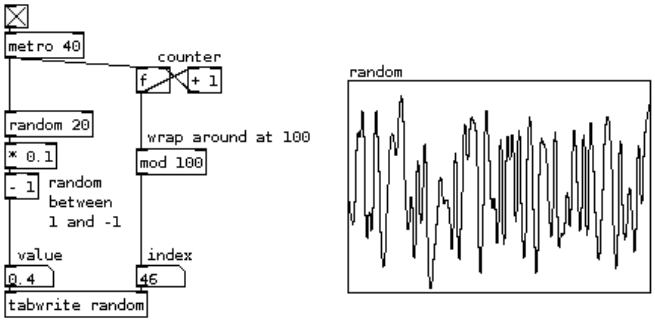

Here the name and size of array can be defined. The name of the table should be unique and $0 can be used in a name (i.e.: $0-sample1) to avoid crosstalk. Size of the array defines how many elements it will hold. If table will be used to control a 16-step sequencer only 16 elements is needed. But if it will contain a two seconds long sound sample (at 44100hz sampling rate) the array should be long 88200 elements, however the table can be resized also later. Save contents will save the contents of an array within the patch file. This can be desired if table will be used for waveforms for oscilations or to control an envelope of the sound, or undesired if soundfiles will be loaded in it - in which case the patch file (something.pd) will become rather big and despite of being a text file will contain sound-wave data. Next three options 'draw as points', 'polygon' and 'bezier curve' define how data will be visualized: as discreet points (horizontal lines), as cornered zigzagging connected lines or smoothed bezier-curved line:




In play mode, it is possible to draw inside the table with the mouse, which can be useful with few elements in a table. Sometimes tables can be used to display the waveform of sound signals. Using tabwrite~ sound signals are recorded into table. Every time a [tabwrite~] receives a bang, it will start recording (sampling) audio signal into the array, graphing it when reaching the end of array:

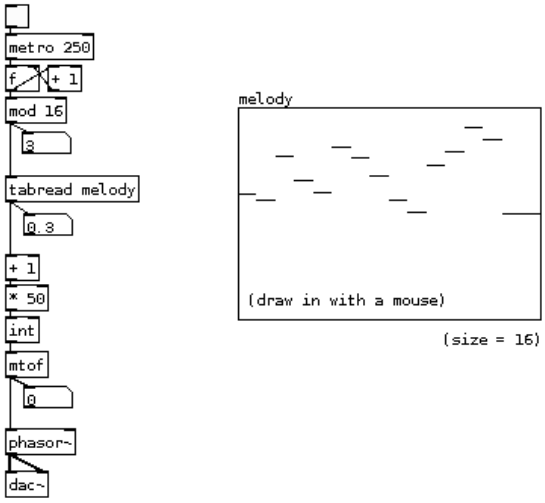



In above example, [tabwrite~] is banged every half second to continuously display the waveform produced from two [phasor~]s, and a [clip~] object. Data can be put as values into tables too, simply by sending an index number (X-coordinate) and a value (Y-coordinate) to [tabwrite] (no tilde!) object:

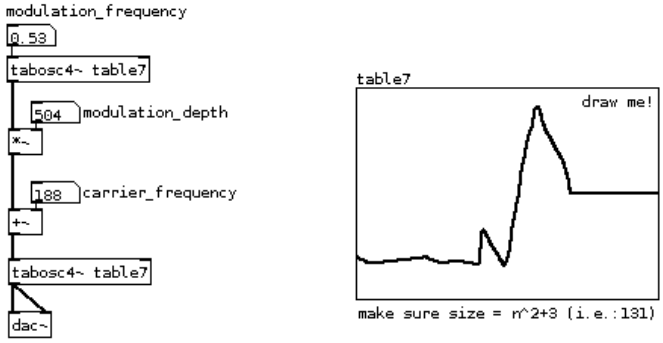



In above example for each index number (they are produced with a counter and start from beggining (0) with [mod 100] at 100) a random value between -1 and 1 is writen to a table.

Tables can be read (looked up) in two ways: to get discrete numbers, or to directly read them as audio waveforms. With [tabread] an index number is taken as an X-coordinate and value in the table (Y-coordinate) is output. In the following example an array is used in a repeating sequencer-like fashion as a simple rudimentary control for an sawtooth oscillator:



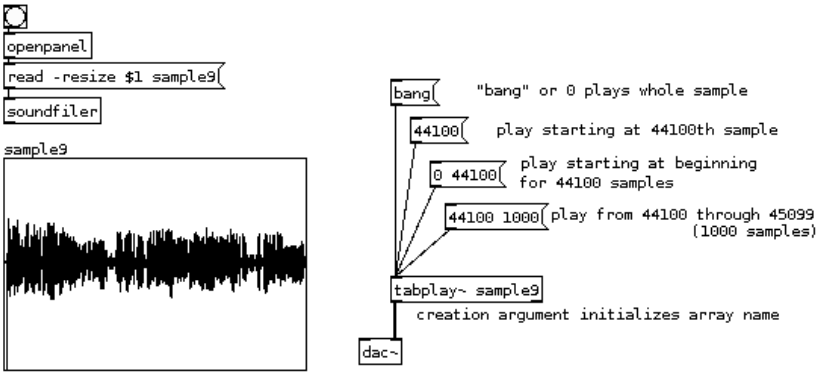

With [tabosc4~] table data is used as an oscillating waveform - like sinewave is used in sinewave oscillator [osc~] and sawtooth wave is used in [phasor~]:



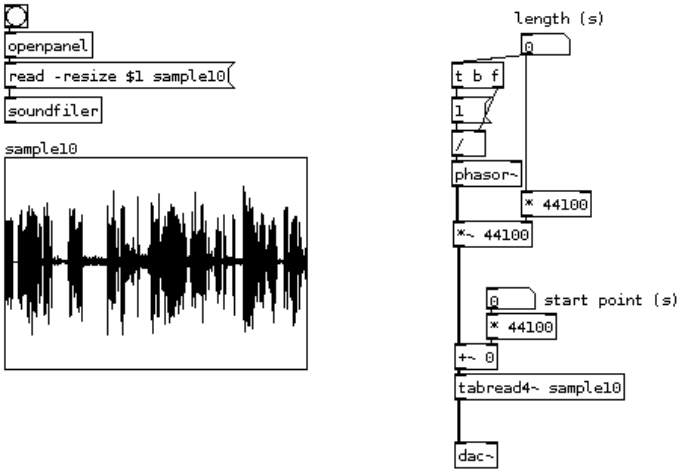

In above example an oscillating waveform from table7 is used to modulate frequency of an oscillator that is using the same waveform to synthesize sound. Changing the table in realtime will influence the modulation

and oscillation. Source for hours of fun!

Another way to read data from a table is to play it as a sound recording - which usually is, especially if array is filled with data from a sound file. For this [soundfiler] object comes handy, as is shown in the following examples. In first, array is played using simple and straightforward [tabplay~] object, which offers flexibility of playing from a specific point for a specific length. Remember, digital sound recording is, simply put, high frequency measurements (sample rate, i.e.: 44.1kHz) of sound vibrations. In PD, when soundfile is loaded into a table, every single measurement (sample) can be accessed. That is why, 44100 samples equals 1 second (in most cases).




Following to the aforementioned possibility of accessing individual samples within a sound recording that's been loaded into an array, a [tabread4~] object allows more computational flexibility. Below, [phasor~] object produces ramps (sawtooth wave) from 0 to 1 at the audio rate (commonly 44100 times in a second). If frequency of the [phasor~] oscilator is 1Hz, it will output a ramp from 0 to 1 in exactly one second. If multiplied by 44100 and sent to [tabread4~], it will read first 44100 indices (indexes) in a second and output the values as an audio signal - example below tries to demonstrates that with a twist or two:




First twist comes from an idea of changing the frequency of phasor, and this way slowing down the ramps. This would however shift the pitch of the sound - like changing speed of a vinyl record. This is prevented by multiplication with higher number of samples, which effectively turn the parameter into the length of a sample that is being looped instead of slowing it down. Looping is here because [phasor~] starts again at 0 after it has reached 1. The other twist is the starting point, which simply shifts the whole loop by adding number of

samples (seconds multiplied by 44100).

# Glossary

(Names of other glossary entries are in **bold** when they first appear in an entry, while the names of PD objects appear in *italics*.)

# Glossary Terms

## Abstraction

A resuable block of code saved as a seperate PD patch and used as if it were an **object.** Any abstraction to be used must either be saved in the same **working directory** as the PD patch it is used in, or the directory it is saved in must be included in the **path** section of the PD settings. Abstractions can be opened by clicking on them, and the **GUI elements** inside can be displayed even when closed by setting their **properties** to **Graph on Parent**. **Inlets** and **outlets** can be used to send and receive information to and from an abstraction, as well as **send** and **receive** pairs.

(Names of other glossary entries are in **bold** when they first appear in an entry, while the names of PD objects appear in *italics*.)

(Names of other glossary entries are in **bold** when they first appear in an entry, while the names of PD objects appear in *italics*.)

## ADC

Analog to Digital Converter - the line into PD from the sound card. The PD **object** for this is *adc~*.

## ADSR

(**Attack**, **Decay**, **Sustain** and **Release**) the common points of change (or breakpoints) in the **envelope** of a **note**.

## Aliasing

whenever a sound is replayed or synthesized whose **frequency** is over the **Nyquist number** (half the current **sampling rate)**, a second frequency will be heard "reflecting" off the Nyquist number downwards at the same increment in **Herz.** Example: if the sampling rate is 44,100 Hz, the Nyquist number would be 22,050. If one attempted to play a sound at 23,050 Hz, an additional tone at 21,050 Hz (the difference between the two frequencies subtracted from the Nyquist number) would be heard.

### ALSA

Advanced Linux Sound Architecture - the default set of **audio drivers** for the Linux operating system.

## Argument

A piece of information sent to an **object** which sets a parameter of that object. Arguments can be sent as **messages**, or taken from **creation arguments**. Arguments are also used to replace **variables** (often represented by **dollar signs**) in messages and objects. By using the *pack* object, multiple arguments can be sent in a message.

## Array

A way of graphically saving and manipulating numbers. It works in an X/Y format, meaning you can ask the array for information by sending it a value representing a location on the X (horizontal) axis, and it will return the value of that position value on the Y (vertical) axis. Arrays are often used to load soundfiles in PD, and are

displayed on screen in **graphs**.

## ASIO

Audio Stream Input/Output - an audio driver for low **latency** audio input and output developed by the Steinberg audio software company and available for many soundcards using the Windows operating system.

## Attack

The beginning of a **note**, which is usually triggered by pressing a key on a keyboard or by a **sequencer.** A slow attack means the sound takes longer to reach full volume than a faster attack. See also **envelope**.

## Audio Driver

Provides a system of input and output between the soundcard and applications using the soundcard. The more efficient the audio driver, the lower the **latency** of an audio system will be. Examples include **MME** and **ASIO** for Windows, **CoreAudio** for Mac OS X and **OSS**, **ALSA** and **JACK** for Linux.

## Bang

is special **message** in PD, which many **objects** interpret as "do something now!", meaning do the operation the object is supposed to do with the information it already has received in its **inlets**. Bang can be sent via a **GUI element**, the *bang* object or a message box. Bang can also be abbreviated to just *b*.

## Bit Depth

Refers to the number of bits used to write a **sample**. Each sample of 16-bit audio, which is the CD standard, is made from 16 bits which can either be 0 or 1. This gives $2^{16}$ (or 2x2x2x2x2x2x2x2x2x2x2x2x2x2x2x2 = 65,536) number of possible values that sample can have. A higher bit depth means a greater **dynamic range**. In contrast to 16 bit audio for CDs, studio recordings are first made at 24 (or even 32) bit to preserve the most detail before transfer to CD, and DVDs are made at 24 bit, while video games from the 1980s remain famous for their distinctively rough "8 bit sound". Bit depth is also referred to as **word length**.

## Buffer

a chunk of memory inside the computer used to store sound. The soundcard uses a buffer to store audio from the audio applications for playback. If the **latency** of the system is too low for the soundcard and **audio drivers**, then the buffer will be too small and the soundcard will use all the audio data in the buffer before getting more from the audio application, resulting in an interruption know as a "dropout", or **glitch.**

## Canvas

An area of pixels in the **patch** which is used to add color or graphical layout to the patch. Since PD remembers when things were put in the ptach, a canvas is placed in the patch before any other objects which must be seen on top of it. Alternately, objects to be seen on top of the canvas can be Cut and then Pasted over it.

## Clipping

Clipping occurs when a signal is too loud for the soundcard to reproduce it. This happens when the **samples** used to represent the sound go out of the range between -1 and 1 due to amplifying them. Any samples out of

this range will be **truncated** to fit within that range, resulting in **distortion**, a loss of audio detail and in **frequencies** which were not present in the original sound. The clipping point of a system is referred to as 0 dB in the **gain** scale, and the gain of any sound is measured in how far below the clipping point it is (-10 dB, -24 dB, etc).

## Cold and Hot

In PD, the left-most **inlet** of an **object** is called "hot", which means that any input to that inlet causes the object to do its function and create output at the **outlet**. Any other inlet to the right of the left-most inlet is considered "cold", which means that input to these outlets is stored in the object until it receives input on the hot inlet, at which time all the information stored in the object is acted on.

## Comment

A line of text in a **patch** which explains some part of the patch, or is a reminder to the programmer or anyone else who opens the patch later on. Comments have no actual affect on the fucntion of the patch.

## Creation Argument

Additional information given when an **object** is created. Example: making an object called *osc~ 440* would create a cosine **oscillator** (the name of the object) with a starting frequency of 440 Hz (the creation argument). See also **Argument**.

## Cutoff Frequency

The **frequency** at which a **filter** begins to affect a sound.

## DAC

Digital to Analog Converter - the line out to the sound card from PD. The PD **object** for this is called *dac~*.

## Decay

The amount of time a sound takes to go from peak volume down to it's **sustain** level (in the case of an **envelope**), or to no sound at all (in the case of a **delay**).

## Decibel

Decibel is a scale used to measure the **gain** or **loudness** of a sound. Decibel is usually abbreviated to dB and usually denotes how far under 0 dB (the **clipping** point of a system) a sound is (-10 dB, -24 dB, etc). The Decibel scale is **logarithmic**.

## Delay

The amount of time between one event and another. As an audio effect, a delay takes an incoming sound signal and delays it for a certain length of time. When mixed with the original sound, an "echo" is heard. By using **feedback** to return the delayed signal back into the delay (usually after lowering its **gain**), multiple echos with a **decay** result. The PD **objects** to create a delay are named *delwrite~* and *delread~*, and the pair must be given the same **creation argument** in order to communicate (i.e. *delwrite~ rastaman* and *delread~ rastaman*). As a setting in PD, delay changes the **latency** of the program to allow for faster response time at the expense of more **gliltches** or vice versa.

## Distortion

Distortion occurs when an audio signal is changed in some way on the level of the **samples** which produces **frequencies** not present in the original. Distortion can be deliberate or unwanted, and can be produced by driving the signal to a **clipping** point, or by using mathematical transformations to alter the shape (or "waveform") of the signal (usually referred to as "waveshaping").

## Dollar Sign

A **symbol** in PD which is used to represent a **variable** in either a **message** or a **creation argument**. Multiple dollar signs can be used, as in "$1 $2 $3". In such a case, $1 will take the first **argument** in an incoming message, $2 the second, $3 the third, etc etc. And in the message "set $1", any number sent to this message would replace $1, resulting in "set 1", "set 2", "set 3" etc depending on what number the message received. In the case of a creation argument used in an **abstraction**, one could create an abstraction named *myniceabs*, and call it in a **patch** as *myniceabs 34*, *myniceabs 66* and *myniceabs 88*. In this case, the initial **frequency** of an *osc~ $1* **object** would be set to 34 **Hz** in the first abstraction, 66 Hz in the second and 88 Hz in the third, since the creation argument of the *osc~* object sets its starting frequency. $0, however, is a special case, and is set to a unique random number for each abstraction it is used in (but it retains the same value everywhere inside that abstraction).

## Dynamic Range

Used to refer to the difference between the loudest sound that can possibly recorded and the quietest, as well as the amount of detail which can be heard in between. Sounds which are too quiet to be recorded are said to be below the **noise floor** of the recording system (microphone, recorder, sound card, audio software, etc). Sounds which are too loud will be **clipped**. In digital audio, the **bit depth** used to record the sound determines the dynamic range, while in analog electronics, the **self-noise** of the equipment also determines the dynamic range.

## Edit Mode

The mode in PD where **objects**, **messages**, **comments**, **GUI elements** and other parts of the PD can be placed on the screen and moved around. Edit mode can be switched in and out of by using the Edit menu or the Control (or Apple) and "E" keys. The opposite of Edit mode is **Play mode**.

## Envelope

A term used to describe changes to a sound over time. Traditionally, this is used to synthesize different instrumental sounds with **Attack**, **Decay**, **Sustain** and **Release** (or **ADSR**) which are triggered at the beginning of a **note**. A violin, for example, has a slow attack as the strings begin to vibrate, while a piano has a fast (or "percussive") attack which seperates it's distinctive sound (or "timbre") from that of other instruments.

## External

An **object** in PD which was not written into the core PD program by the author, Miller S. Puckette. Externals are created and maintained by the Pure Data development community, and account for many of the additional fucntions of PD, including the ability to manipulate video and 3D as well as stream MP3s and many other things. Externals are usually loaded as an **external library** at the start of a PD session by including them in the **startup flags,** although some can be loaded as single objects at anytime as long as the location where that external is saved on your system is listed in the **path** setting of PD.

## External Library

A collection of **externals** written for PD. Taken as a library, externals can be loaded at the start of a PD session by including them in the **startup flags.**

## Filter

An audio effect which lowers the **gain** of **frequencies** above and/or below a certain point, called the **cutoff frequency**. The range it allows through is called the **pass band**, and the frequencies which are reduced are called the **stop band**. A High Pass filter (*hip~*) only allows frequencies above the cutoff frequency through. A Low Pass filter (*lop~*) allows only frequencies lower than the cutoff frequency through. A Band Pass filter (*bp~*) only allows frequencies close to the cutoff frequency through. The amount by which the filter lowers the gain of frequencies in the stop band is measured in **Decibels** per **Octave**, and is affected by the **resonance** (or "Q") of the filter, which determines the amount of **feedback** the filter uses and which frequency is most emphasized by the filter.

## Feedback

Feedback occurs in any system where the output is played back into the input. 100% feedback means all of the output is returned to the input. A classic example is holding a microphone in front of a speaker. Less than 100% feedback means that the signal is decreased in some way with each pass through the system. In **delays**, the amount of feedback determines how many repetitions of the "echo" one hears until the sound **decays** to zero. In a **filter**, feedback determines the **resonance** of the filter, and how much emphasis in given to the filter's **cutoff frequency**.

## Float or Floating Point

A **number** with a decimal point, which can be positive or negative and represent a range between -8388608 and 8388608. A special notation is used for extremely large or small floating point numbers, since PD only uses up to 6 characters to represent a floating point number. Therefore, "1e+006" is a floating point number which represents "1000000" (or 1 with 6 decimal places after it), while "1e-006" represents "0.0000001" (or 1 with 6 decimal places in front of it).

## Foldover

Foldover occurs when a **frequency** higher than the **Nyquist number** is played or synthesized. See **Aliasing**.

## Frequency

Refers to number of times in one second a vibration (in many cases a sonic vibration) occurs. Frequency is measured in **Herz**, and often indicates the **pitch** of a sound which is heard. Frequency is a **linear** scale, however, while pitch is **logarithmic**. This means that a sound which is heard as one **octave** above another one is twice the frequency in Hz, while two octaves above would be four times the frequency and three octaves above would be eight times.

## Gain

Expresses the strength of an audio signal, and is expressed in **Decibels**. The scale of gain is **logarithmic**, since it expresses the physical ratio of power between one sound and another. Gain is commonly measured in digital audio systems as the amount of Decibels below 0 dB, which is the **clipping** point (-10 dB, -24 dB, etc). See also **loudness**.

## Glitch

A sonic error occurring when the computer does not have enough time to process the audio coming in or out of an audio application before sending it to the sound card. This is a result of having too low a **latency**, so that the **buffers** of the sound card are not filled up as fast as the soundcard is playing them, resulting in an temporary but audible loss of sound. Glitches can occur when other processes interrupt the processor with various tasks (such as refreshing the display on the screen, reading or writing a hard drive, etc etc).

## Graph

A graph is a graphical container that can hold several **arrays.** An array needs a graph to be displayed, so whenever you create an array from the menu, you will be asked whether you want to put it into a newly created graph or into an existing graph.

## Graph on Parent

A **property** of **subpatches** and **abstractions** where the **GUI elements** of the subpatch or abstraction are visible in the main **patch** even when that subptach or abstraction is not open. This allows for better graphic design and usability for complicated patches.

## GUI element

Graphical User Interface - visible parts of the PD **patch** which are used to control it via the mouse or to display information, such as **sliders**, **radio** buttons, **bangs**, **toggles**, **number** boxes, **VU** meters, **canvases**, **graphs**, **arrays**, **symbols**, etc.

## Hot and Cold

In PD, the left-most **inlet** of an **object** is called "hot", which means that any input to that inlet causes the object to do its function and create output at the **outlet**. Any other inlet to the right of the left-most inlet is considered "cold", which means that input to these outlets is stored in the object until it receives input on the hot inlet, at which time all the information stored in the object is acted on.

## Hradio

A horizontal **radio** button. See also **GUI element**.

## Hslider

A horizontal **slider**. See also **GUI element**.

## Herz or Hz

A term used to describe the number of times something occurs in one second. In digital audio, it is used to describe the **sampling rate**, and in acoustics it is used to describe the **frequency** of a sound. Thousands of Herz are described as KHz.

## Inlet

The small rectangular boxes at the top of **objects**, **GUI elements**, **messages**, **subpatches** and **abstractions**. They receive input from the **outlets** of the objects, messages, GUI elements, subpatches or abstractions above them. Inlets can be **hot** or **cold**.

## Integer

In PD, this is a whole **number**, without a decimal point, which can be positive or negative. See also **floating point**.

## JACK

JACK Audio Connection Kit - a low **latency** audio system designed to run on Linux and Mac OSX in combination with various **audio drivers** such as **ALSA** and **Portaudio.** On Linux, the **QJackctl** application can be used to make audio and **MIDI** connections between the soundcard, MIDI devices such as keyboards and PD. On Mac OSX, JACK is referred to as JackOSX, and the JackPilot application functions like QJackCtl, but only for audio connections.

## Latency

The amount of time needed to process all the **samples** coming from sound applications on your computer and send it to the soundcard for playback, or to gather **samples** from the sound card for recording or processing. A shorter latency means you will hear the results quicker, giving the impression of a more responsive system which musicians tend to appreciate when playing. However, with a shorter latency you run a greater risk of **glitches** in the audio. This is because the computer might not have enough time to process the sound before sending it to the soundcard. A longer latency means less glitches, but at the cost of a slower response time. Latency is measured in miliseconds.

## Linear

A scale of numbers which progresses in an additive fashion, such as by adding one (1, 2, 3, 4...), two (2, 4, 6, 8...) or ten (10, 20, 30, 40...). Another type of scale used in PD is **logarithmic**. Multiplying an audio signal, for example, by either a linear or a logarithmic scale will produce very different results. The scale of **frequency** is linear, while the scales of **pitch** and **gain** are logarithmic.

## Logarithmic

A scale of numbers which progresses according to a certain ratio, such as exponentially (2, 4, 8, 16, 256...). Another type of scale used in PD is **linear**. Multiplying an audio signal, for example, by either a linear or a logarithmic scale will produce very different results. Both scales of **pitch** and **gain** are logarithmic, while the scale of **frequency** is linear.

## Loudness

Unlike **gain**, which expresses the physical power of a sound, loudness is the preceived strength of a sound. Higher **frequencies** are perceived as louder than mid-range or lower frequencies with the same amount of gain, and the amount of perceived difference varies from person to person.

## Message

A piece of information sent to the **objects** of a **patch**, often using the message **GUI element**. Messages tell objects which functions to perform and how, and can be simply numeric, include text which describes which function to change or even contain other information such as the location of soundfiles on the computer.

## MIDI

A system of describing musical information in electronic music using numbers between 0 and 127. There are various types of MIDI messages which can be sent in and out of PD such as **note** (*notein*, *noteout*), pitchbend (*pitchin*, *pitchout*), continuous controller (*ctlin*, *ctlout*) and program change (*pgmin*, *pgmout*). MIDI messages can be sent to and from external MIDI devices, such as keyboards, slider boxes or hardware **sequencers**, or they can be exchanged with other MIDI applications inside the computer.

## MME

The default set of **audio drivers** for the Windows operating system. MME drivers do not have as low **latency** as ASIO drivers.

## Monophonic

A monophonic electronic music instrument has one **voice**, meaning that only one **note** can be played at a time. See also **polyphonic**.

## Noise Floor

The part of the **dynamic range** which represents the quietest sound which can be recorded or played back. Sounds below this level (expressed in **Decibels**) will not be heard over the background noise of the system. In digital audio, the **bit depth** used to record the sound determines the noise floor, while in analog electronics, the **self-noise** of the equipment also determines the noise floor. Typical computer soundcards can have an analog noise floor between approximately -48 dB and -98 dB.

## Note

In electronic and computer music, a note is represented on the **MIDI** scale by two numbers between 0 and 127 (the amount of keys available on the MIDI keyboard). A note is triggered either by pressing a key on the keyboard or by a sequencer. A MIDI note has two values: it's **pitch** (the musical note it plays, expressed as a **frequency** which has been assigned to that note) and it's **velocity** (how hard the key is pressed, which determines how loud the note is heard). Notes also have an **envelope**, which determines the change in volume that note has over time.

## Number

a **GUI element** used to display and store numbers. The number2 GUI element can also save numbers when that function is set in its **properties**.

## Nyquist Number

A number which is half the **sampling rate** of the application which is being used, and represents the highest possible **frequency** which can be played back without **aliasing**. The Nyquist number is expressed in **Herz**. Example: if the sampling rate is 44,100 Hz, the Nyquist number would be 22,050. If one attempted to play a sound at 23,050 Hz, an aliased additional sound at 21,050 Hz (the difference between the two frequencies subtracted from the Nyquist number) would be heard.

## Object

The most basic building block of a PD **patch**. Objects have a names, which could be considered the "vocabulary" of the PD language, and the name of the object determines its function. Objects can take **creation arguments** to modify their functions at the time they are created. They receive information via **inlets** and send output via **outlets**. Objects with a tilde (~) in their name are audio generating or processing objects, otherwise they are objects to manipulate data (for example, an object named + would add two numbers

together, and an object named +~ would add two audio signals together). To see the documentation help file of any object, right click with the mouse, or use the Control (or Apple) key with a mouseclick.

## Octave

The interval between one musical note and another with 12 semitones (or 12 **notes** in the **MIDI** scale) between them, which is seen in acoustics as half or double the **frequency**. While frequency is a **linear** scale, however, while pitch is **logarithmic**. This means that a sound which is heard as one **octave** above another one is twice the frequency in Hz, while two octaves above would be four times the frequency, three octaves above would be eight times higher, and one octave below would be half the frequency.

## Oscillator

An audio generator which produces a continuous, repeating waveform. A cosine oscillator (*osc~*) produces a pure sinus wave with no harmonics, while a sawtooth or ramp oscillator (*phasor~*) produces a richer sound with many harmonics. Other shapes for a waveform include square, pulse or triangle. Each waveform is defined by a mathematical function, and each shape has its own harmonic spectrum.

## OSS

An outdated system of **audio drivers** for the Linux operating system, replaced by **ALSA**.

## Outlet

The small rectangular boxes at the bottom of **objects**, **GUI elements**, **messages**, **subpatches** and **abstractions**. They send output to the **intlets** of the objects, subpatches, abstractions, messages and GUI elements below them.

## Pass Band

The range of **frequencies** allowed through by a **filter**.

## Patch

The document in which you build structures within PD. One patch can contain many **objects**, **comments**, **GUI elements**, **messages**, **subpatches** and **abstractions**. If another patch is saved in the same **working directory** or in another directory listed in the **path** setting, then it can be used in the main or parent patch as an abstraction. Patches are saved as simple text files with the names and locations of all the contents listed inside. Patches are always saved with the .pd extension.

## Path

Is a setting of PD which determines two things. The first is the directories on your computer which PD searches to load **externals**, and the second is the directories where PD searches to find **abstractions** used in patches. Path can be set with **startup flags**, or by entering the directories in the startup settings using the main window of PD.

## Pitch

A part of a **note** in the **MIDI** specification which determines what pitch is heard when the note is played. It is represented by a number between 0 and 127, with each number representing a key on the MIDI keyboard. The relation of pitch to **frequency** is **logarithmic**. This means that a sound which is heard as one **octave** (+ 12

MIDI notes) above another one is twice the frequency in Hz, while two octaves (+ 24 MIDI notes) above would be four times the frequency, three octaves (+ 36 MIDI notes) above would be eight times, and one octave below (- 12 MIDI notes) would be half the frequency.

## Play Mode

The mode in PD where the **GUI elements** and other parts of the PD can be manipulated with the mouse. This is often when the **patch** is being played. Play mode can be switched in and out of by using the Edit menu or the Control (or Apple) and "E" keys. The opposite of Play mode is **Edit mode**.

## Polyphonic

A polyphonic electronic music instrument is capable of playing multiple **notes** at a time, allowing for chords and other musical techniques. The number of notes it can play is determined by the number of **voices** it has. See also **monophonic**.

## Portaudio

A Free and Open Source set of **audio drivers** for Linux and Mac OS X.

## Property

All the **GUI elements** in PD have a menu where their properties can be changed. This is accessed by using the right-click mouse button, or the Control (or Apple) key and a mouseclick. Under properties, the graphical appearance and function of the GUI element can be changed.

## Radio

A **GUI element** set of buttons which, when clicked, send the number of the box which was clicked to the **outlet**, or display numbers received by its **inlet**. Radio boxes can be vertical or horizontal, and the number of boxes seen can be changed in the **properties**.

## Real-time

A system where changes can be made in the program even as it is running, and the user can see or hear the results immediately. The opposite would be a non-real-time system, where data must be compiled or rendered by the computer in order to hear or see results.

## Release

The amount of time it takes for the **gain** of a **note** to reach zero after the key on the keyboard has been released. See also **envelope**.

## Resonance

The **frequency** in a filter or other system of **feedback** which is most emphasized, resulting in that frequency being the loudest.

## Sample

In digital audio, a sample is the smallest possible element of a recorded sound. In CD audio, for example, it takes 44,100 samples to make one second of recorded sound, and so we can say that the **sampling rate** is

44,100 **Herz**. Samples also have a **bit depth** which determines the **dynamic range** that is possible to record and playback. Common bit depths are 8 (for old video games), 16 (for CD audio), 24 (for studio recording and DVDs) or 32 (for sounds inside the computer). In electronic music, a sample is also a prerecorded piece of sound which is played back by a **sampler**.

## Sampler

An electronic music intrument which plays back a recorded sound (or **sample**) whenever it is sent a **note**. The **pitch** of the note determines how fast or slow the sample is played back, which emulates the pitch changes in other instruments. Samples can be looped (played over and over) and one-shot (played once).

## Sampling Rate

The rate at which the computer records and plays back sound, which is measured in **Herz** representing the number of **samples** per second. CD audio is recorded and played at 44,100 Hz (or 44.1 KHz), while DVD audio runs at 96,000 Hz (or 96 KHz) and cheap consumer gadgets like voice recorders, video games, mobile phones, toys and some MP3 players often use a rate of 22,050 Hz (22.05 KHz) or even less. The sampling rate determines the highest **frequency** which can be recorded or played, which is expressed by the **Nyquist number**, or half the sampling rate. Sounds higher in frequency than the Nyquist rate will be **aliased**. Playing back sounds at a different sampling rate then they were recorded at will result in hearing that sound at the "wrong speed".

## Sequencer

A **MIDI** device or application used to store **notes** which are sent to a **synthesizer** or **sampler**. Sequencers often play notes back at a rate specified in Beats per Minute.

## Self-noise

The amount of analog noise a piece of electronic equipment produces without any further input, often due to parts of its circuitry or electromagentic interference. Self-noise is measured in **Decibels**. The self noise of the equipment determines the **noise floor**. Professional or semiprofessional sound equipment often produces less self-noise than cheaper, consumer-grade equipment. Typical computer soundcards have self-noise which results in a noise floor between approximately -48 dB and -98 dB.

## Send and Receive

A method of communicating between **objects** in a **patch** without the connecting cables. The objects *send* and *receive* are used, with a shared **creation argument** which sets the "channel" they transmit on, for example *send volume* and *receive volume*. The names of the objects can be abbreviated to *s* and *r*, and a pair for audio signals also exists (*send~* and *receive~*, or *s~* and *r~*).

## Shell

The text-only interface to your computer, where commands are typed in order to start programs and get information. On Linux and Mac OSX, this is often called the "terminal". On Windows, it is referred to as the Command Prompt or (now obsolete) as the DOS Prompt.

## Slider

A **GUI element** which sends a number to its **outlet** when it is moved with the mouse, or display numbers received by its **inlet**. Sliders can be horizontal or vertical, and when they are created have a typical **MIDI** range of 0 to 127. This range can be changed under the **properties**.

## Startup Flag

When starting PD from the **shell**, the startup flags are used to pass information to PD about how it should run, what **audio drivers** it should use, how many channels, what **patch** to open at startup, which **external libraries** to load and what **paths** to use to find **externals** and **abstractions**.

## Stop Band

The **frequencies** which are reduced by a **filter**.

## Subpatch

A graphical enclosure in a **patch** used to conceal parts of the patch which are not always used. Subpatches can be opened by clicking on them, and the **GUI elements** inside can be displayed even when closed by setting their **properties** to **Graph on Parent**. **Inlets** and **outlets** can be used to send and receive information to and from a subpatch, as well as **send** and **receive** pairs.

## Sustain

The level of **gain** a **note** holds after the **attack** and **decay**. The note holds this gain level until the key is **released**. See also **envelope**.

## Symbol

Any part of a **message** which is not a number. Single words or locations of data on the computer are common symbols, and there are a variety of **externals** which can be used to construct more complicated symbols.

## Synthesizer

A sound producing device or application which receives **notes** and plays sound based on these notes.

## Table

Like a **graph**, a table is a way of using an **array** in a PD **patch**. In this case, it is used like an **abstraction**, with a **creation argument** which gives the name of the array. For example, if you create an **object** named *table mytablename*, then inside the *table* object you will find an array named "mytablename" inside its own **graph**.

## Toggle

A **GUI element** which sends either a zero or a non-zero number (typically 1) to its **outlet** when clicked, or displays zero or a non-zero number  received by its **inlet**. Its function can be changed under its **properties**.

## Truncate

When a number goes out of a certain set of allowed boundaries, it will be truncated. This means that any numbers out of that range will be replaced by the closest number still within that range (either the highest or lowest). In a digital audio signal, this is called **clipping**.

## Variable

A type of "placeholder", often within a **message** and written as a **dollar sign**, which is meant to be replaced with other information. For example, in the message "$1 $2 $3", there are three variables to be replaced with actual information.

## Vector Based Graphics

The graphical system used by PD to display **patches** where every element on the screen is defined by a set of numbers describing their appearance rather than an image, and every change to these elements means that the computer must recalculate that part of the screen.

## Velocity

A part of a **note** in the **MIDI** specification which says how hard the key of the keyboard was pressed, and in turn determines the **gain** of that note when it is played. It is represented by a number between 0 and 127.

## Voices

A **polyphonic** electronic music instrument can play as many simultaneous **notes** as it has voices. A **monophonic** instrument, on the other had, can only play one note at a time and is said to have one voice.

## Vradio

A vertical **radio** button. See also **GUI element**.

## Vslider

A vertical **slider**. See also **GUI element**.

## VU

A **GUI element** in PD which is used to display the **gain** of an audio signal in **Decibels**.

## White noise

A random signal **(or process)** with a flat power spectral density. In other words, the signal's power spectral density has equal power in any band, at any centre frequency, having a given bandwidth. White noise is considered analogous to **white light** which contains all frequencies.

An infinite-bandwidth, white noise signal is purely a theoretical construction. By having power at all frequencies, the total power of such a signal is infinite. In practice, a signal can be "white" with a flat spectrum over a defined frequency band.

## Word Length

See **bit depth**.

## Working Directory

In PD this is the directory which the patch you are working in has been saved to. Any **abstractions** used in that patch must either be saved to that directory, or the directory in which those abstractions have been saved

must be added to the **path** setting in the startup preferences.

A list of the core Pure Data objects, followed by lists of objects included in various external libraries.

Contents:

* Core Pure Data

* IEMLib

* Zexy

* MaxLib

* PDP

*PiDiP

* GEM

# Core Pure Data

The "reference" section of the documentation should contain a patch
demonstrating how to use each of Pd's classes.  As of version 0.29, a complete
list of "object" classes follows.  Not included in this list are messages,
atoms, graphs, etc. which aren't typed into object boxes but come
straight off the "add" menu.

---------- GLUE ----------
bang -  output a bang message
float -  store and recall a number
symbol -  store and recall a symbol
int -  store and recall an integer
send -  send a message to a named object
receive - catch "sent" messages
select -  test for matching numbers or symbols
route -  route messages according to first element
pack -  make compound messages
unpack -  get elements of compound messages
trigger - sequence and convert messagess
spigot -  interruptible message connection
moses -  part a numeric stream
until -  looping mechanism
print -  print out messages
makefilename - format a symbol with a variable field
change -  remove repeated numbers from a stream
swap -  swap two numbers
value -  shared numeric value
---------- TIME ----------

delay - send a message after a time delay
metro - send a message periodically
line - send a series of linearly stepped numbers
timer - measure time intervals
cputime - measure CPU time
realtime -measure real time
pipe - dynamically growable delay line for numbers
------------------------------ MATH ---------------------------------
+ - * / pow arithmetic
== = > < >= <= relational tests
& && | || % bit twiddling
mtof ftom powtodb rmstodb dbtopow dbtorms convert acoustical units
mod div sin cos tan atan atan2 sqrt log exp abs higher math
random lower math
max min greater or lesser of 2 numbers
clip force a number into a range
------------------------------ MIDI ---------------------------------
notein ctlin pgmin bendin touchin polytouchin midiin sysexin - MIDI input
noteout ctlout pgmout bendout touchout polytouchout midiout - MIDI output
makenote - schedule a delayed "note off" message corresponding to a note-on
stripnote - strip "note off" messages
------------------------------ TABLES----------------------------------
tabread - read a number from a table
tabread4 - read a number from a table, with 4 point interpolation
tabwrite - write a number to a table
soundfiler - read and write tables to soundfiles
------------------------------ MISC ---------------------------------
loadbang - bang on load
serial - serial device control for NT only
netsend - send messages over the internet
netreceive - receive them
qlist - message sequencer
textfile - file to message converter
openpanel - "Open" dialog
savepanel - "Save as" dialog
bag - set of numbers
poly - polyphonic voice allocation
key, keyup - numeric key values from keyboard
keyname - symbolic key name
------------------------- AUDIO MATH ---------------------------------
+~ -~ *~ /~ arithmetic on audio signals
max~ min~ - maximum or minimum of 2 inputs
clip~ - constrict signal to lie between two bounds
q8_rsqrt~ - cheap reciprocal square root (beware -- 8 bits!)
q8_sqrt~ - cheap square root (beware -- 8 bits!)
wrap~ - wraparound (fractional part, sort of)
fft~ - complex forward discrete Fourier transform
ifft~ - complex inverse discrete Fourier transform
rfft~ - real forward discrete Fourier transform
rifft~ - real inverse discrete Fourier transform
framp~ - output a ramp for each block
mtof~, ftom~, rmstodb~, dbtorms~, rmstopow~, powtorms~ - acoustic conversions
------------------------- AUDIO GLUE ---------------------------------
dac~ - audio output

```
adc~ -      audio input
sig~ -      convert numbers to audio signals
line~ -     generate audio ramps
vline~ -    deluxe line~
threshold~  detect signal thresholds
snapshot~ - sample a signal (convert it back to a number)
vsnapshot~  deluxe snapshot~
bang~ -     send a bang message after each DSP block
samplerate~ get the sample rate
send~ -     nonlocal signal connection with fanout
receive~ -  get signal from send~
throw~ -    add to a summing bus
catch~ -    define and read a summing bus
block~ -    specify block size and overlap
switch~ -   switch DSP computation on and off
readsf~ -   soundfile playback from disk
writesf~ -  record sound to disk
-------------------- AUDIO OSCILLATORS AND TABLES ------------------------
phasor~ -   sawtooth oscillator
cos~ -      cosine
osc~ -      cosine oscillator
tabwrite~ - write to a table
tabplay~ -  play back from a table (non-transposing)
tabread~ -  non-interpolating table read
tabread4~ - four-point interpolating table read
tabosc4~ -  wavetable oscillator
tabsend~ -  write one block continuously to a table
tabreceive~ read one block continuously from a table
-------------------- AUDIO FILTERS ------------------------
vcf~ -      voltage controlled filter
noise~ -    white noise generator
env~ -      envelope follower
hip~ -      high pass filter
lop~ -      low pass filter
bp~ -       band pass filter
biquad~ -   raw filter
samphold~ - sample and hold unit
print~ -    print out one or more "blocks"
rpole~ -    raw real-valued one-pole filter
rzero~ -    raw real-valued one-zero filter
rzero_rev~  rzero~, time-reversed
cpole~, czero~, czero_rev - corresponding complex-valued filters
-------------------- AUDIO DELAY ------------------------
delwrite~ - write to a delay line
delread~ -  read from a delay line
vd~ -       read from a delay line at a variable delay time
------------------------------ SUBWINDOWS ---------------------------------
pd -      define a subwindow
table -   array of numbers in a subwindow
inlet -   add an inlet to a pd
outlet -  add an outlet to a pd
inlet~, outlet~ - signal versions of inlet, outlet
------------------------------ DATA TEMPLATES -----------------------------
struct -    define a data structure
```

drawcurve, filledcurve - draw a curve
drawpolygon, filledpolygon - draw a polygon
plot - plot an array field
drawnumber - print a numeric value
------------------------------ ACCESSING DATA ----------------------------
pointer - point to an object belonging to a template
get - get numeric fields
set - change numeric fields
element - get an array element
getsize - get the size of an array
setsize - change the size of an array
append - add an element to a list
sublist - get a pointer into a list which is an element of another scalar
scalar - draw a scalar on parent
------------------------------ OBSOLETE ----------------------------
scope~ (use tabwrite~ now)
namecanvas
template (use struct now)

## IEMLIB

contents of iemlib Release 1.15 from December 2003

============================== DSP~ **==========================**

------------------------- filter~ -------------------------------
FIR~ finite impuls response filter, with array-coefficients

maverage~ moving average filter, (IIR + delay)

ap1~ allpass 1.order
ap2~ allpass 2.order
bpq2~ bandpass 2.order with Q-inlet
bpw2~ bandpass 2.order with bandwidth-inlet
bsq2~ bandstop 2.order (notch) with Q-inlet
bsw2~ bandstop 2.order (notch) with bandwidth-inlet
hp1~ highpass 1.order
hp2~ highpass 2.order
lp1~ lowpass 1.order
lp2~ lowpass 2.order
rbpq2~ resonance-bandpass 2.order with Q-inlet
rbpw2~ resonance-bandpass 2.order with bandwidth-inlet

hml_shelf~ high-middle-low shelving-filter with freq- and gain-inlets
lp1_t~ lowpass 1.order with time_constant inlet
para_bp2~ parametrical bandpass 2. order with freq-, Q- and gain-inlet

hp2_butt~, hp3_butt~, hp4_butt~, hp5_butt~, hp6_butt~, hp7_butt~,
hp8_butt~, hp9_butt~, hp10_butt~
highpass 2.3.4.5.6.7.8.9.10.order with butterworth characteristic
hp2_cheb~, hp3_cheb~, hp4_cheb~, hp5_cheb~, hp6_cheb~, hp7_cheb~,

hp8_cheb~, hp9_cheb~, hp10_cheb~
   highpass 2.3.4.5.6.7.8.9.10.order with chebyshev characteristic
hp2_bess~, hp3_bess~, hp4_bess~, hp5_bess~, hp6_bess~, hp7_bess~,
hp8_bess~, hp9_bess~, hp10_bess~
   highpass 2.3.4.5.6.7.8.9.10.order with bessel characteristic
hp2_crit~, hp3_crit~, hp4_crit~, hp5_crit~, hp6_crit~, hp7_crit~,
hp8_crit~, hp9_crit~, hp10_crit~
   highpass 2.3.4.5.6.7.8.9.10.order with critical damping
lp2_butt~, lp3_butt~, lp4_butt~, lp5_butt~, lp6_butt~, lp7_butt~,
lp8_butt~, lp9_butt~, lp10_butt~
   lowpass 2.3.4.5.6.7.8.9.10.order with butterworth characteristic
lp2_cheb~, lp3_cheb~, lp4_cheb~, lp5_cheb~, lp6_cheb~, lp7_cheb~,
lp8_cheb~, lp9_cheb~, lp10_cheb~
   lowpass 2.3.4.5.6.7.8.9.10.order with chebyshev characteristic
lp2_bess~, lp3_bess~, lp4_bess~, lp5_bess~, lp6_bess~, lp7_bess~,
lp8_bess~, lp9_bess~, lp10_bess~
   lowpass 2.3.4.5.6.7.8.9.10.order with bessel characteristic
lp2_crit~, lp3_crit~, lp4_crit~, lp5_crit~, lp6_crit~, lp7_crit~,
lp8_crit~, lp9_crit~, lp10_crit~
   lowpass 2.3.4.5.6.7.8.9.10.order with critical damping

vcf_hp2~, vcf_hp4~, vcf_hp6~, vcf_hp8~
   highpass 2.4.6.8.order with freq- and Q-signal-inlets
vcf_lp2~, vcf_lp4~, vcf_lp6~, vcf_lp8~
   lowpass 2.4.6.8.order with freq- and Q-signal-inlets
vcf_bp2~, vcf_bp4~, vcf_bp6~, vcf_bp8~
   bandpass 2.4.6.8.order with freq- and Q-signal-inlets
vcf_rbp2~, vcf_rbp4~, vcf_rbp6~, vcf_rbp8~
   resonance-bandpass 2.4.6.8.order with freq- and Q-signal-inlets

------------------------ arithmetic~ ----------------------------
addl~        signal-addition with line~
divl~        signal-divison with line~
mull~        signal-multiplication with line~
subl~        signal-subtraction with line~

------------------------- converter~ ----------------------------
prvu~        peak and rms VU-meter interface
pvu~         peak VU-meter interface
rvu~         rms VU-meter interface
unsig~       signal to float converter

------------------ t3~ - time-tagged-trigger --------------------
-- inputmessages allow a sample-accurate access to signalshape --
t3_sig~         time tagged trigger sig~
t3_line~     time tagged trigger line~

--------------------------- misc~ -------------------------------
fade~        fade-in fade-out shaper (need line~)
iem_blocksize~     blocksize of a window in samples
iem_samplerate~     samplerate of a window in Hertz
int_fract~     split signal-float to integer- and fractal-part
LFO_noise~     downsampled 2-point interpolated white noise
mp3play~     mp3 stereo player

peakenv~ peak envelope shaper
pink~ pink noise
round~ round signal-float to nearest integer
sin_phase~ output phase-difference of 2 sinewaves in samples

========== control **==========**

---------- gui (included into millers pd) ----------

bng bang, display and generate a bang-message
cnv canvas, colored background and text
hdl horizontal dial, for multiplex usage
hradio horizontal radiobutton, only float in/out
hsl horizontal slider
nbx numberbox, the second
tgl 2 state toggle
vdl vertical dial, for multiplex usage
vradio vertical radiobutton, only float in/out
vsl vertical slider
vu vu-meter, display rms- + peak-level in dB

---------- float operating ----------
1p1z float-message-filter 1.order
db2v db to rms
dbtofad midi-db to fader-characteristic
fadtodb fader-characteristic to midi-db
fadtorms fader-characteristic to rms
rmstofad rms to fader-characteristic
round_zero round numbers near zero to zero
speedlim reduce speed of a numeric stream
split3 part a numeric stream into 3 ways
split part a numeric stream into 2 ways (like moses)
transf_fader partial linear characteristic diagram (like table)
v2db rms to db
wrap wraparound

---------- symbol operating ----------
mergefilename merge a list of symbols together
splitfilename divide a symbol into 2 parts
stripfilename strip n characters of a symbol
unsymbol convert a symbol- to a anything-message

---------- anything operating ----------
any store and recall any message (like f, or symbol)
iem_append append a message to any messages (obsolete: merge_any)
iem_prepend prepend a message to any messages (abbr. pp or prepend)

---------- init ----------
default replace initial-argument, if it is zero
dollarg receive parent initial-arguments (abbr. $n)
dsp control audio-engine, calculate dsp-performance (aka. dsp~)
float24 store a 24-bit accurate float-number
init initialize a message via loadbang (abbr. ii)
once any message pass through only the first time

------------------------ counter ----------------------------
exp_inc  exponential increment counter (bang triggered)
for++  incremental counter (triggered by internal metro)
modulo_counter  endless loop counter (bang triggered)

------------------------- misc -------------------------------
add2_comma  add a comma-separated message to a messagebox
bpe  break point envelope controller
f2note  frequency to midi+cents+note
gate  interruptible message connection (like spigot)
iem_i_route  variation of route (abbr. iiroute)
iem_receive  catch "sent" messages (receive-name-input) (abbr. iem_r)
iem_route  improvement of route
iem_sel_any  control a message-box with multiple content
iem_send  send messages to named object (send-name-input)(ab. iem_s)
pre_inlet  output an identifier-message and then the incoming message
prepend_ascii  output an identifier-message and then the incoming message
soundfile_info  output header-info of a wav-file
toggle_mess  control a message-box with multiple content (abbr. tm)

------------------- parameter handling ------------------------
iem_pbank_csv  parameter memory manager (csv-format) (like textfile)
list2send  array of send-objects
receive2list  array of receive-objects

--------------- t3 - time-tagged-trigger  ---------------------
----------- a time-tag is prepended to each message -----------
----- so these objects allow a sample-accurate access to ------
---------- the signal-objects t3_sig~ and t3_line~ ------------
t3_bpe  time tagged trigger break point envelope
t3_delay  time tagged trigger delay
t3_metro  time tagged trigger metronom
t3_timer  time tagged trigger timer

-------------- obsolete ---------------------------------------
post_netreceive
pre_netsend

==========================================================================

## ZEXY

These are the objects that come with the zexy-external

============================== DSP~ **==========================**

--------------------------- IO~ -------------------------------
sfplay  play back (multi-channel) soundfiles
sfrecord  record (multichannel) soundfiles

---------- generators~ ----------
dirac~ dirac-pulse
step~ unity step
noish~ downsampled noise (hold)
noisi~ downsampled noise (interpolate)

---------- processing~ ----------
limiter~ a limiter/compressor module
quantize~ quantizes signals
swap~ bytes swap a 16bit-signal
blockmirror~ time-reverse a signal-vector (1,2,...,64 -> 64,63,...,1)
blockswap~ swap the upper and lower half of a signal-vector
z~ samplewise delay

---------- analytic~ ----------
sigzero~ detects whether a signal is zero throughout the vector or not
pdf~ probability density function
envrms~ like env~, but outputting rms instead of dB
avg~ arithmetic mean of 1 signal-vector
tavg~ arithmetic mean between two bangs
dfreq~ frequency detector

---------- sigbinops~ ----------
>~, <~, ==~, &&~, ||~ logical operators
abs~ absolute value of a signal
sgn~ signum of a signal

---------- misc~ ----------
nop~ no-operation
pack~ convert a signal to a list of floats
unpack~ convert a list of floats to a signal
matrix~ matrix-multiply m IN-signals to n OUT-signals
multiline~ multiply a number of signals with scalars (interpolated)
multiplex~ multiplex 1-of-n inlets to 1 outlet
demultiplex~ demultiplex 1 inlet to 1-of-n outlets

========== control **==========**

---------- basic ----------
nop no-operation
repeat repeat a message several times
lister store lists (like "float" for floats)
repack (re)pack atoms to packages of a given size
packel get a specified element of a list
drip extract the atoms of a package (opt. scheduled)
length get the length of a list
niagara split 1 packages into 2
glue append a package to another (glue them together)
segregate segregate the input to various outlets, depending on the type
any2list convert "anythings" to "lists"
list2int cast each float of a list to integer
atoi ascii to integer
strcmp compare lists as strings

list2symbol  convert a list into a single symbol
symbol2list  convert a symbol to a list

---------- advanced ----------
tabdump  dump out a table as a list of floats
tabset  set a table with a list of floats
makesymbol  concatenate lists to formatted symbols
date  get system date
time  get system time
index  map symbols to indices
msgfile  a powerful "textfile" derivative
demultiplex  demultiplex the input to a specified outlet
lpt  write to the (parallel) port (linux only)
operating_system  get the current OS


---------- maths ----------
mavg  moving average filter for floats
mean  get the mean value of a list of floats
minmax  get minimum and maximum of a list of floats
sort  shell-sort a list of floats
urn  unique random numbers
prime  test whether a number is prime or not
wrap  wrap the float-input between to boundaries
.  scalar multiplication of vectors (=lists of floats)
deg2rad  convert between degree and radiant
rad2deg  convert between radiant and degree
cart2pol, pol2cart, cart2sph, sph2cart, pol2sph, sph2pol  convert between coordinate systems (cartesian, polar, shperic)

---------- matrix ----------
matrix  create/store/... matrices
mtx_element  set elements of a matrix
mtx_row  set rows of a matrix
mtx_col  set columns of a matrix
mtx_ones  matrix with all elements==1
mtx_zeros  matrix with all elements==0
mtx_eye  identity matrix
mtx_egg  identity matrix (from upper-right to lower-left)
mtx_diag  diagonal matrix
mtx_diegg  diagonal matrix (from upper-right to lower-left)
mtx_diag  get the diagonal of a matrix
mtx_trace  get the trace of a matrix
mtx_transpose  transpose a matrix
mtx_roll  column-shift a matrix
mtx_scroll  row-shift a matrix
mtx_pivot  pivot-transform a matrix
mtx_resize  resize a matrix (evtl. with zero-padding)
mtx_size  get the size of a matrix
mtx_inverse  get the inverse of a matrix
mtx_add, mtx_+  add 2 matrices (or an offset to 1 matrix)
mtx_sub, mtx_-  subtract 2 matrices (or an offset from 1 matrix)
mtx_mul, mtx_*  multiply 2 matrices (or a factor with 1 matrix)
mtx_.*  multiply 2 matrices element by element

mtx_./ divide 2 matrices element by element
mtx_mean get the mean value of each column
mtx_rand matrix with random elements
mtx_check check the consistency of a matrix and repair
mtx_print print a matrix to the stderr

==========

## MAXLIB

maxlib 1.5 Music Analysis eXtensions LIBrary
written by Olaf Matthes <olaf.matthes@gmx.de>

MUSIC/MIDI ANALYSIS
chord chord detection
beat beat tracking
borax music analysis
rythm beat detection
score array01 score following
pitch pitch information
gestalt "gestalt" of music
edge detect rising/falling edge
tilt measure tilt of input

MATH

divmod calculate / and %
divide / for several inputs
minus - for several inputs
multi * for several inputs
plus + for several inputs
average average of last N values
history average over last N seconds
match match input to list of numbers
scale scale input to output range
delta calculate 1st or 2nd order diff.
wrap wrap a number in a range [obs!]
rewrap wrap it back and forth

ROUTING/CHECKING

split split according to range [obs?]
nroute r. according to Nth elem.
unroute opposite of route
limit limiter for floats
listfunnel Max's funnel for lists
nchange s change that accepts any kind of input

(REMOTE) CONTROL
dist send to list of recieve objects
netdist same for netreceive
remote send to one receive object

netrec netreceiev with extra info about sender
netserver bidirectional communication (client/server based)
netclient

TIME

pulse a "better" metro
speedlim lets input through every N miliseconds
iso play sequence of MIDI notes
ignore ignore too fast changing
step a line object that steps
velocity velocity of input in digits per second
sync extended trigger object
timebang send a bang at given time of day
pong bouncing ball model
temperature amount of input changes per time

BUFFER

lifo last in first out for floats
fifo first in first out for floats
listfifo first in first out for lists
arraycopy copy from one array to another

OTHER/EXPERIMENTAL

subst self-similar substitution
mlife cellular automaton

RANDOM

gauss
linear
expo
beta
cauchy
poisson
bilex, arbran array01 array02

==========

## PDP

This is a list of all pdp objects and abstractions with a minimal description.
Take a look at the patches in the doc/ directory for more info.
(Messy doc & test patches can be found in the test/ directory.)

general purpose pdp modules:

pdp_del a packet delay line
pdp_reg a packet register

pdp_snap takes a snapshot of a packet stream
pdp_trigger similar to pd's trigger object
pdp_route routes a packet to a specific outlet
pdp_loop a packet loop sampler (packet array)
pdp_description output a symbol describing the packet type
pdp_convert convert between packet types

image inputs/outputs:

pdp_xv displays images using the xvideo extension
pdp_glx displays images using opengl
pdp_v4l reads images from a video4linux device
pdp_qt reads quicktime movies

image processors:

pdp_abs absolute value
pdp_add adds two images
pdp_and bitwize and
pdp_bitdepth set bit depth
pdp_bitmask apply a bit mask
pdp_bq spatial biquad filter
pdp_bqt temporal biquad filter
pdp_cog gaussian blob estimator
pdp_constant fills an image with a constant
pdp_conv horizontal/vertical seperable convolution filter
pdp_cheby chebyshev color shaper
pdp_chrot rotates the chroma components
pdp_flip_lr flip left <-> right
pdp_flip_tb flip top <-> bottom
pdp_grey converts an image to greyscale
pdp_grey2mask converts a greyscale image to an image mask
pdp_hthresh hard thresholding
pdp_mul multiplies two images
pdp_mix crossfade between 2 images
pdp_mix2 mixes 2 images after applying a gain to each of them
pdp_noise a noise generator
pdp_not bitwize not
pdp_or bitwize or
pdp_plasma plasma generator
pdp_pointcloud convert an image to a point cloud
pdp_positive sign function that creates a bitmask
pdp_randmix crossfades 2 images by taking random pixels
pdp_rotate tiled rotate
pdp_scale rescale an image
pdp_sign sign function
pdp_sthresh soft thresholding
pdp_zoom tiled zoom
pdp_zrot tiled zoom + rotate
pdp_zthresh zero threshold (x<0 -> 0)
pdp_xor bitwize xor

dsp objects

pdp_scope~ a very simple oscilloscope
pdp_scan~ phase input scanned synthesis oscillator
pdp_scanxy~ x,y coordinate input scanned synthesis oscillator

utility abstractions

pdp_pps computes the packet rate in packets/sec

image abstractions

pdp_agc automatic gain control (intensity maximizer)
pdp_blur blurs an image
pdp_blur_hor horizontal blur
pdp_blur_ver vertical blur
pdp_contrast contrast enhancement
pdp_dither a dither effect
pdp_phase applies an allpass filter to an image
pdp_phase_hor horizontal allpass
pdp_phase_ver vertical allpass
pdp_motion_blur blurs motion
pdp_motion_phase phase shifts motion
pdp_offset add an offset to an image
pdp_alledge an all edge detector
pdp_conv_emboss emboss
pdp_conv_sobel_hor horizontal sobel edge detector
pdp_conv_sobel_ver vertical sobel edge detector
pdp_conv_sobel_edge sum of squares of hor and ver
pdp_saturation change colour saturation
pdp_sub subtract 2 images
pdp_invert inverse video
pdp_gain3 set 3 channel gains independently
pdp_gradient converts a greyscale to colour using a colour gradient
pdp_png_to convert a png file (on disk) to a certain packet type
pdp_tag tag a packet (to use it with route)

matrix processors

pdp_m_mv matrix vector multiply
pdp_m_mm matrix matrix multiply
pdp_m_+=mm matrix matrix multiply add
pdp_m_LU compute LU decomposition
pdp_m_LU_inverse compute matrix inverse from LU decomp
pdp_m_LU_solve solve a linear system using LU decomp

matrix abstractions

pdp_m_inverse compute matrix inverse

SEPARATE LIBRARIES:

cellular automata
(pdp_scaf)

pdp_ca computes a cellular automaton (as a generator or a filter)
pdp_ca2image convert a CA packet to a greyscale image (obsolete: use pdp_convert)
pdp_image2ca convert an image to a CA packet (black and white) (obsolete: use pdp_convert)

3d drawing objects
(pdp_opengl)

3dp_windowcontext a drawable window
3dp_draw draw objects (cube, sphere, ...)
3dp_view viewing transforms (rotate, translate, ...)
3dp_light light source
3dp_push push a matrix (modelview, texture, ...)
3dp_dlist compile a display list
3dp_snap copies the drawing buffer to a texture packet
3dp_mode set the current matrix mode
3dp_toggle set some opengl state variables

3d drawing abstractions (pdp_opengl)

3dp_mouserotate connect to 3dp_windowcontext to rotate the scene
3dp_blend turn on accumulative blending mode

==========

## PiDiP

pdp_ascii an ASCII art renderer
pdp_canvas a video canvas
pdp_charcoal charcoal effect
pdp_cmap a color mapper
pdp_colorgrid a color picker
pdp_compose a video compositor
pdp_capture screen capture to video utility
pdp_ctrack a color tracker
pdp_disintegration disintegration effect
pdp_ffmpeg~ a video streamer towards a ffserver
pdp_form a geometric forms adder
pdp_i/pdp_o PD to PD streaming objects
pdp_imgloader load an image and blend it with a video source
pdp_live~ a video stream decoder (at least from ffserver)
pdp_mgrid a grid-based motion detector
pdp_morphology: pdp_binary
pdp_erode
pdp_dilate
pdp_hitandmiss
pdp_distance

+ patches: closing, opening, skeletization, thinning, thickening
pdp_mp4live~ a quicktime stream emitter (darwin, quicktime)
pdp_mp4player~ a quicktime stream receiver (darwin, quicktime)
pdp_pen free hand drawing object
pdp_rec~ a quicktime file recorder
Video: jpeg, yuv2, divx, dv, yuv2
Audio: twos, raw
pdp_shape shape detection object
pdp_spigot a video signal router
pdp_spotlight a spotlight especially made for cabaret
pdp_text a text addition object
pdp_theorin~ threaded theora/ogg files reader
pdp_theorout~ theora/ogg files recorder
pdp_transition transition between two video sources
pdp_qt quicktime movie reader
pdp_qt~ quicktime movie reader with audio
pdp_yqt a quicktime movie reader with less functionnalities
pdp_qt but with less functionnalities
codecs : jpeg, yuv2, divx, dv, yuv2
no compressed headers!
pdp_aging, pdp_baltan, pdp_cycle, pdp_dice, pdp_edge,
pdp_intrusion, pdp_lens, pdp_mosaic, pdp_nervous, pdp_puzzle,
pdp_quark, pdp_radioactiv, pdp_rev, pdp_ripple, pdp_shagadelic
pdp_simura, pdp_spiral, pdp_transform, pdp_underwatch, pdp_vertigo
pdp_warhol, pdp_warp : the port of effecTV to PDP.

## GEM

Controls
gemhead - the start of rendering chain
gemwin - the window manager
gemmouse - outputs the mouse position and buttons in the GEM window
gemkeyboard - outputs the keycode of a key pressed when you are in the GEM window (there might be different keycodes in Windows/Linux)
gemkeyname - outputs a symbolic description of a key pressed when you are in the GEM window (there might be different symbols in Windows/Linux)
gemorb - outputs the position, rotation, and buttons for a Space Orb
gemtablet - outputs the pen position, pressure, and buttons in the GEM window

Manipulators
accumrotate - accumulate a rotation
alpha - enable/disable alpha blending
ambient - set the ambient color with a vector
ambientRGB - set the ambient color with 3 discrete values
camera -
color - set the color with a vector
colorRGB - set the color with 3 discrete values
depth - enable/disable depth testing
diffuse - set the diffuse color with a vector

diffuseRGB - set the diffuse color with 3 discrete values
emission - set the emissive color with a vector
emissionRGB - set the emissive color with 3 discrete values
linear_path - generate a path from an array of points
ortho - change the view to orthogonal, with the viewport the size of the window
polygon_smooth - turn on anti-aliasing for the objects below
rotate - rotate with an angle and vector
rotateXYZ - rotate with 3 discrete values
scale - scale with a vector
scaleXYZ - scale with 3 discrete values
separator - push the OpenGL state for the rest of the chain and pop when done
shininess - set the shininess of an object
specular - set the specular color with a vector
specularRGB - set the specular color with 3 discrete values
spline_path - generate a spline from an array of knots
translate - translate with a vector
translateXYZ - translate with 3 discrete values

Geos
circle - render a circle
colorSquare - render a colored square (evtl. with color gradients)
cone - render a cone
cube - render a cube
cuboid - render a box
curve - render a Bezier curve
curve3d - render a surface
cylinder - render a cylinder
disk - render a disk
imageVert - make pixel colors to a height field map
model - render an Alias|Wavefront model
multimodel - render a series of Alias|Wavefront models, render by number
newWave - render a wave (that is evolving over time)
polygon - render a polygon
primTri - a triangle primitive
rectangle - render a rectangle
ripple - a rectangle with distorted (over time) texture-coordinates
rubber - a grid where you can move one of the grid-points
slideSquare - render a number of sliding squares
sphere - render a sphere
square - render a square
teapot - render a teapot
text2d - render 2-D text (a bitmap)
text3d - render 3-D text (polygonal)
textextruded - render an extruded 3D-text
textoutline - render outlined text (polygonal)
triangle - render a triangle

Particles
part_head - The start of a particle group
part_color - Set the range of colors for the new particles
part_damp - set the damping for particles
part_draw - Apply the actions and render the particles.  Accepts a message "draw line" or "draw point" to change the drawing style.
part_follow - Particles will follow each other like a snake

part_gravity - Have the particles accelerate in a direction
part_info - get the information (position, color, size,...) of each particle
part_killold - Remove particles past a certain age
part_killslow - Remove particles below a certain speed
part_orbitpoint - Orbit the particles around a specified point
part_render - render the remaining gem-tree as particles.
part_size - Set the size of new particles
part_source - Generate particles
part_targetcolor - Change color of the particles toward the specified color
part_targetsize - Change size of the particles toward the specified size
part_velocity - Set the velocity domain (distribution like CONE and the appropriate arguments)
part_vertex - emit a single particle

Nongeos
light - make a point light
world_light - make a light at infinity

Pixes
pix_2grey - convert rgb pixels to grey (still an RGBA image)
pix_a_2grey - convert rgb pixels to grey based on alpha channel
pix_add - add two pixes together
pix_aging - super8-like aging effect
pix_alpha - set the alpha value of a pix
pix_background - let through only pixels that differ from a static "background" image
pix_backlight - a backlight photo effect
pix_biquad - 2p2z-filter for subsequent images
pix_bitmask - apply a bitmask to a pix
pix_blob - get center of gravity
pix_buf - buffer a pix
pix_buffer - storage room for pixes (like [table] for floats)
pix_buffer_read/pix_buffer_write - put/get pixes into/from a pix_buffer
pix_chroma_key - color keying (like "blue-box")
pix_coloralpha - set the alpha-channel of a pix as a mean-value of the color-components
pix_colormatrix - recombine the RGBA-channels with matrix-operation
pix_color - set the color of a pix (leaving alpha alone)
pix_colorreduce - reduce the number of colors (statistically)
pix_composite - composite two pixes together
pix_convolve - convolve a pix with a kernal
pix_coordinate - set the texture coordinates
pix_crop - get a sub-image of a pix
pix_curve - apply color-curves onto a pix
pix_data - get pixel data information
pix_delay - frame-wise delay
pix_diff - get absolute difference of two pixes
pix_dot - rasterize a pix with big dots
pix_draw - draw a pix
pix_dump - dump the pixel-data as a long list of floats
pix_duotone - reduce the number of colors by thresholding
pix_film - use a movie file as a pix source for image-processing
pix_flip - flip the pixels of a pix
pix_gain - apply a gain to a pix
pix_grey - convert any pix into greyscale colorspace
pix_halftone - rasterize a pix like it was printed in a newspaper
pix_histo - get the histogram of a pix

pix_hsv2rgb - transform a pix from HSV-colorspace into RGB-colorspace
pix_image - load in an image file
pix_imageInPlace - load a series of image files directly into texture-buffer, display by number
pix_info - get information about the pix (like dimension, colorspace,...)
pix_invert - invert a pix
pix_kaleidoscope - as if you were looking at the pix through a kaleidoscope
pix_levels - level adjustment
pix_lumaoffset - y-offset pixels depending on their luminance
pix_mask - mask a pix based on another pix
pix_metaimage - recompose an image out of smaller versions of itself
pix_mix - mix to pixes together
pix_motionblur - motionblur an image
pix_movie - use a movie file as a pix source and load it immediately into the texture-buffer
pix_movement - set the alpha-channel with respect to the change between two frames
pix_multiply - multiply two pixes
pix_multiimage - load in a series of image files, display by number
pix_normalize - normalize a pix
pix_offset - add an offset to a pix (wrapping instead of clipping)
pix_pix2sig~ - interpret a pix as 4 (RGBA) audio-signals
pix_posterize - posterization photo effect
pix_puzzle - shuffle an image
pix_rds - generate a Random Dot Stereogram out of the image (aka: Magic Eye (tm))
pix_rectangle - generate a rectangle in a pix buffer
pix_refraction - break up an image into coloured "glass-bricks"
pix_resize - resize a pix to next power of 2
pix_rgb2hsv - transform a pix from RGB-colorspace into HSV-colorspace
pix_rgba - transform a pix of any format into RGBA
pix_roll - (sc)roll through an image (wrapping)
pix_rtx - swap time-axis and x-axis
pix_scanline - take every nth line of the original image
pix_set - set the pixel-data with a long list of floats
pix_sig2pix~ - interpret 4 audio-signals as (RGBA) image-data
pix_snap - capture the render window into a pix
pix_snap2tex - capture the render window directly as a texture
pix_subtract - subtract two pixes
pix_tIIR - time-base Infinite-Impulse-Response filter (for motion-bluring,...) with settable number of poles/zeros
pix_takealpha - take the alpha channel of one pix and put it into another pix
pix_texture - use a pix as a texture map
pix_threshold - apply a threshold to a pix
pix_video - use a video camera as a pix source
pix_write - capture the render window to disk
pix_zoom - zoom into a pix (using OpenGL)

openGL there are more than 250 objects that form a complete wrapper around the openGL set of functions (as defined in the openGL-1.2 standard).
each openGL-function is prefixed with "GEM", eg: [GEMglVertex3f] is wrapped around glVertex3f.

MarkEx
alternate - alternate between two outlets
average - average a sequence of numbers
change - only output on change
counter - count bangs
invert - non-zero numbers to zero, zero to 1

multiselect/multisel - a select object which accepts a list in the right inlet
oneshot - send a bang, then block until reset
randomF / randF - floating point random numbers
strcat - string concatentation
tripleLine - do a line with three numbers
tripleRand - random with three numbers
vector+ / v+ - add a scalar to a vector
vector- / v- - subtract a scalar from a vector
vector* / v* - multiply a vector by a scalar
vector/ / v/ - divide a vector by a scalar
vectorpack / vpack - attach a scalar to the end of a vector
rgb2hsv - convert a list of three floats from RGB to an HSV value
hsv2rgb - convert a list of three floats from HSV to an RGB value
abs~ - absolute value of a signal
reson~ - resonant filter

# PD Links

## Pure Data Software

PureData.org: http://www.puredata.org/

PD Downloads: http://www.puredata.org/downloads

Pure Data CVS: http://www.puredata.org/dev/cvs

PD Extended Installers: http://at.or.at/hans/pd/installers.html

Miller S. Puckette's PD page: http://www-crca.ucsd.edu/~msp/software.html

Desire Data: http://desiredata.goto10.org/

## Externals

PD Downloads: http://www.puredata.org/downloads

Pure Data CVS: http://www.puredata.org/dev/cvs

GEM: http://gem.iem.at/

PDP: http://zwizwa.fartit.com/zwikizwaki.php?page=PureDataPacket

PiDiP: http://ydegoyon.free.fr/pidip.html

Unauthorized PD: http://ydegoyon.free.fr/software.html

PMPD: http://drpichon.free.fr/pmpd/

## Linux Distributions with PD

Dyne:bolic: http://www.dynebolic.org/ 

Pure Dyne: http://puredyne.goto10.org/

Ubuntu Studio: http://ubuntustudio.org/ 

PlanetCCRMA: http://ccrma.stanford.edu/planetccrma/software/

PD Gentoo Overlay: http://pd-overlay.sourceforge.net/

## Tutorials & Examples

Pd community patches: http://www.puredata.org/community/patches

Pure Data Documentation Project: http://www.davesabine.com/eMedia/PureData/tabid/145/Default.aspx

Theory and Techniques of Electronic Music by Miller Puckette:
http://www.crca.ucsd.edu/~msp/techniques.htm

Music making tutorials: http://www.obiwannabe.co.uk/html/music/musictuts.html 

Practical synthetic sound design in Pd:
http://www.obiwannabe.co.uk/html/sound-design/sound-design-all.html

PD Repertory Project: http://crca.ucsd.edu/~msp/pdrp/latest/doc/

PD Network Video Loop: http://www.yourmachines.org/tutorials/pdvideolan.html

# Getting Help

Pure Data Mailing List (Search): http://lists.puredata.info/pipermail/pd-list/

Pure Data Mailing List (Subscribe): http://lists.puredata.info/listinfo/pd-list

Pure Data Forum: http://puredata.hurleur.com/

# License

# Authors

*AudioStreaming*
© adam hyde 2005, 2006, 2007, 2008
Modifications:
corey fogel 2007
Derek Holzer 2008
Felipe Ribeiro 2007
Heiko Recktenwald 2006

---

*ConfiguringPD*
© Derek Holzer 2006, 2008
Modifications:
adam hyde 2007, 2008
Georg ... 2008

---

*Credits*
© adam hyde 2006, 2007
Modifications:
Derek Holzer 2006, 2008

---

*DataflowTutorials*
© Derek Holzer 2006, 2008
Modifications:
Luka Princic 2008

---

*InstallingDebian*
© adam hyde 2008
Modifications:
Derek Holzer 2008

---

*InstallingOSX*
© Derek Holzer 2006, 2008
Modifications:
Daniel Prieto 2007
Maarten Brinkerink 2007

---

*InstallingUbuntu*
© adam hyde 2008
Modifications:
Derek Holzer 2008

---

*InstallingWindows*
© adam hyde 2006, 2008
Modifications:
Derek Holzer 2008

---

*Introduction*
© adam hyde 2006, 2008
Modifications:
Derek Holzer 2006, 2007, 2008
Evelina Domnitch 2007

*ListofObjects*


*PureGlossary*

Modifications:
adam hyde 2008
michela pelusio 2007

*PureLinks*


*SimpleSynth*


*StartingPD*

Modifications:
adam hyde 2008
corey fogel 2007
Daniel Prieto 2007

*TheInterface*

Modifications:
Daniel Prieto 2007

*TroubleShooting*

Modifications:
adam hyde 2008

*WhatIsDigitalAudio*

Modifications:
adam hyde 2008

*WhatIsGraphicalProgramming*

Modifications:
adam hyde 2008
Maarten Brinkerink 2007


Free manuals for free software

# General Public License

Version 2, June 1991



**Preamble**

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

**TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION**

**0.** This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without

limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

**1.** You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

**2.** You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

> **a)** You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
>
> **b)** You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
>
> **c)** If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

**3.** You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

**a)** Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

**b)** Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

**c)** Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

**4.** You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

**5.** You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

**6.** Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

**7.** If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

**8.** If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

**9.** The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

**10.** If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

**NO WARRANTY**

**11.** BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

**12.** IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**END OF TERMS AND CONDITIONS**