PD programming conventions

for multiple users of large PD patches

Programming conventions are important because:

- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly (and allow the original authors to refresh their memory more quickly).
- Current estimates suggest 80% of the lifetime cost of a piece of software goes to maintenance in the industry. Outside industry, 'cost' translates to time spent adjusting software.

Comments

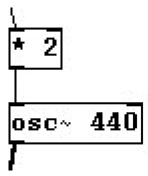
- Concise comments should be used to clarify the purpose of the code.

- Excess comments should be **avoided**. Where possible, the code should be made clear enough to make comments unnecessary. *When the code is altered or tweaked, it is likely the comments would quickly get out of date. Unnecessary commenting will distract from the useful commenting.*

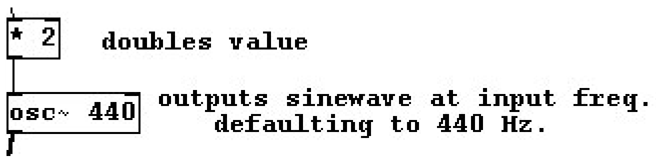
- Although one paragraph-sized comment at the start of a module may be helpful to describe its high-level purpose, large sentences in the code should be avoided where possible. *If they seem necessary, the code may be too complicated and should be broken down further or coded more simply.*

- It is not necessary to comment on the function of individual objects. *Anyone using the patch will either know what they mean or can look it up in PD documentation.*

Good:

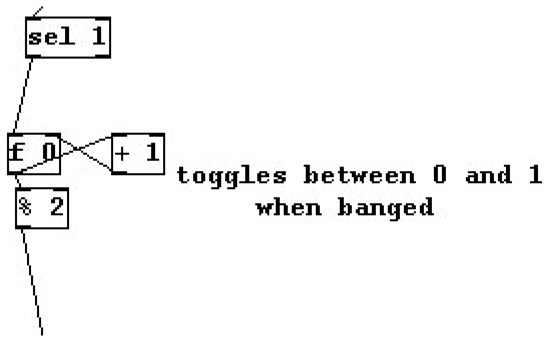


Bad:

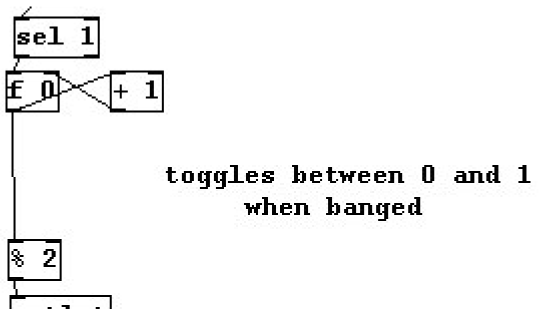


- When commenting on the combined function of a group of objects, these objects should be physically moved closer together and/or separated from other objects. *They will appear to the eye as a distinct group with a combined purpose, and it will be obvious that the comment refers to them.*

Good:

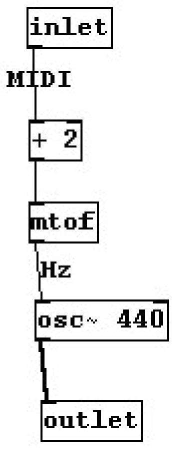


Bad:



- When the data flowing is of a particular type, this should be emphasised by a comment **over** the the first connector after the data-type is changed or first connector on screen.

Good:



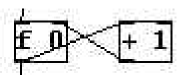
- Avoid using the words 'DEBUGGING', 'ABSTRACTION' and 'DUMMY' except as directed below, so they can be searched for and processed appropriately at the test/integration stage.

Data Flow

- Data should generally flow from the top of the screen to the bottoms of the screen. *This makes it easy for the eye to follow.*

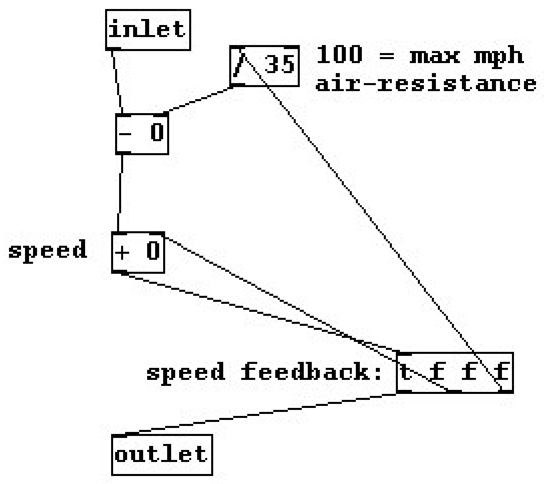
- A common exception to the above is in the 'counter' set-up or similar, where the feedback only goes back one object. In this situation, the object feeding back should be placed beside its preceding object.

OK:

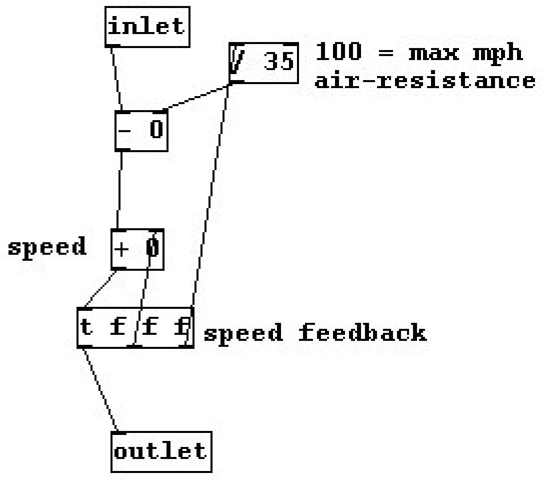


- If feedback over more objects is required, the object that sends its data 'up' the control flow should be set to one side of the rest of the objects. *This makes it the unusual data flow obvious at a glance.*

Readable:



Unreadable:



Subpatches and Abstractions

- All modules should be initially developed as subpatches. *i.e. typing* '`pd subpatchName`' *in an object. This makes them easier to transfer from programmers to testers/integrators, and prevents the need to keep track of more files than necessary..*

- If a module is intended to be used at various different points of the finished work, it should be clearly labeled with a comment of 'ABSTRACTION' in block capitals at the top-left corner. *This should make its purpose clear at the testing/integration stage.*

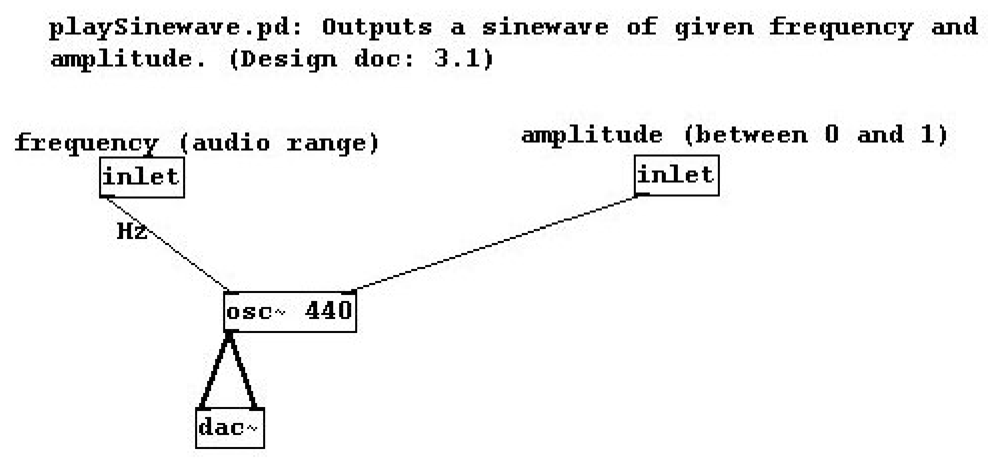
- The subpatch should be given a name that clearly describes its purpose in 'camel case' (first word in lower-case, each subsequent word with just an initial capital but no spaces, e.g. `thisIsCamelCase`)

- If there are too many inlets or outlets for the width of a subpatch, adding underscores (e.g. [exampleSubpatch_________]) may add clarity. However, this is not suitable for abstractions. In abstractions, it would be important to retype the same number of underscores each time, which would be fiddley.

- A comment at the top of the subpatch should describe:

- the purpose of the subpatch.
- how it relates to the design specification (using reference numbers if appropriate).
- the part of the program from which it is called (unless it is called from many places).

Example:

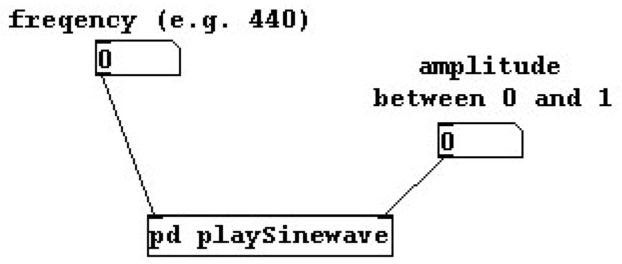

playSinewave.pd: Outputs a sinewave of given frequency and amplitude. (Design doc: 3.1)

- `inlet` and `outlet` objects should be clearly labeled with the type of information they will receive and pass on.

- Avoid more than 20 objects in a subpatch, unless the objects are in a particular repetitive pattern which will make them easier to comprehend. If necessary use further subpatches. *Too many objects may seem off-putting or unclear to someone unfamiliar with how they work.*

- When you are checking the subpatch works, before passing it to testing/integration, use the inlets and outlets you have created, rather than cluttering your subpatch. These do not need to be deleted afterwards, as they may be useful in testing/integration. *Accidentally misconnecting messages, number-boxes or other in your subpatch could lead to false test results, and removing them afterwards could cause errors.*

Example:



Sends and Receives

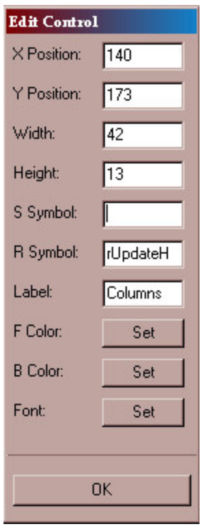
- Sends and receives should be **avoided** where possible. *Direct connections are easier to follow, and, in large patches, the send/receive names may be accidentally reused, causing hard-to-track bugs.*

- When the best design solution involves global sends and receives:
- names should be kept on a centralised list and included in the master patch or in any documentation.
- names should clearly describe the information they sent, using camel case (e.g. `thisIsCamelCase`).

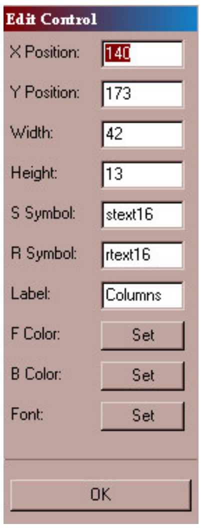
- Sends and receives required to communicate with GrIPD should begin with a small-case ‘r’ or ‘s’ depending on whether they’re being received in the interface or sent from the interface respectively. *This adds clarity and reduces chances of using the same name twice.*

- If a GrIPD object's send or receive is not being used, it should be deleted from the parameter settings. *For tidiness, and perhaps to save computational time?*

Good:


Edit Control
X Position: 140
Y Position: 173
Width: 42
Height: 13
S Symbol:
R Symbol: rUpdateH
Label: Columns
F Color: Set
B Color: Set
Font: Set
OK

Bad:

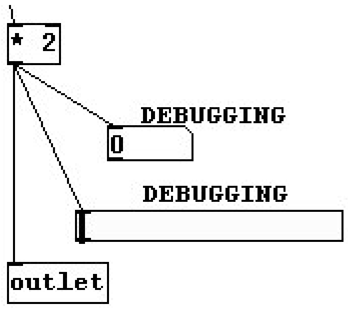

Edit Control
X Position: 140
Y Position: 173
Width: 42
Height: 13
S Symbol: stext16
R Symbol: rtext16
Label: Columns
F Color: Set
B Color: Set
Font: Set
OK

- If a send and receive must be used internally within one module:
- they should be off to one side of the rest of the objects, and vertically aligned. *It is easier to see where they lead to.*
- they should be preceded by '`$0-`' so that they will not interfere with other modules. This makes it uniquely identified, even if the same name is used in other modules.
- they should be labeled with the comment 'ABSTRACTION' in the top left-hand corner of the subpatch. '`$0-`' *only works in abstractions, not sub-patches.*

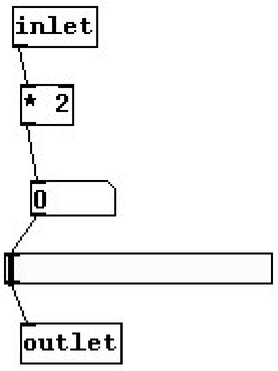
Debugging

- You may want to add number-boxes, symbols, graphical objects or print objects while developing the program. If so:
- Avoid making them inline. *This slows processing and thus potentially changes the functionality of the module. Also they are more time-consuming and fiddley to delete if they are inline.*
- Mark with with a comment saying 'DEBUGGING' in uppercase. *If you forget to delete them, they can be easily found at the test/integration stage by searching for the string.*

Good:



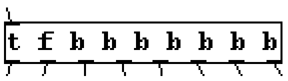
Bad:



General

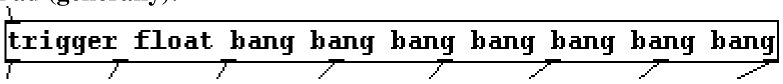
- If your module is reliant on another module not yet coded, include dummy-code as necessary, but mark it clearly in a comment as ‘DUMMY’
- Use abbreviations where they are available, such as those provided in the table below. These leave more room in the patches, and lists of them can be more easily scanned once familiar with them. *A mixture of abbreviated and unabbreviated objects is harder to follow than either wholly one or the other. However, using the full forms of the words may sometimes (conversely) be beneficial in decluttering the patch..*

Good (generally):



```
t f b b b b b b b
```

Bad (generally):



```
trigger float bang bang bang bang bang bang bang
```

PD abbreviations:

Keyword	Abbreviation
trigger	t
send	s
receive	r
int	i
float	f
bang	b
symbol	s
list	l
anything	a