Debugging Max Patches

Designing a Max patch (or any computer program) can be difficult, especially before you have a lot of experience. However, if you keep at it, you will find designing gets easier surprisingly fast. One thing that never gets any easier is finding out what is wrong when the patch does not work. Every time we encounter some problem, we have to go through the five stages of debugging:

- Denial
- Anger
- Examination
- Repair
- Testing

The first two items in this list are only partly a joke-- dealing with bugs is the most frustrating part of writing code, and I have certainly invented many new expletives in the process of finding problems. Bugs are part of the process and can't be avoided. Examination is the first productive item on the list. This means looking at every object and patch cord to see if the right parts are in the right order. When we find an apparent goof, we fix it, then test the complete patch again. Our examination can include three phases: Look over the patch. Consult the max window.
Use the tools in the debug menu.

Things to look for

I try to point out common mistakes throughout these papers. This makes for a certain amount of repetition but I'm not assuming anyone will read them all. So here yet again are the things I find most often when looking at student work:

Messages trying to be objects and vice versa.

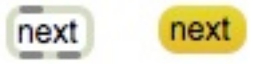


Figure 1. Object and message

Confusing the message box and object box is usually just a matter of hitting the m key when the n was needed. I encourage students to modify the message box color to make it obvious when this has happened. However, the difference between objects and messages can be confusing to the beginner because there's nothing like either in other languages. An object box represents code. It is smart, and will give you some result calculated from its input. A message box is dumb. All it will produce is the message you have typed into it.

Dots left out of math boxes

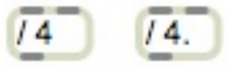


Figure 2. Integer and float

Max chooses float or integer math based on the argument in a math object. If there is no argument, integer is used. This is a difficult problem to detect because it won't stop a patch from running, it will just give wrong answers. This distinction applies to a number of objects . For instance, pack and unpack will convert input to match the type of the corresponding arguments in the list. Select will match a float to an integer argument but will not match an integer to a float argument. The math comparisons convert incoming data before the test-- therefore any float from 3.0 to 3.999 will match 3 entered as an integer.

Patch line to wrong inlet

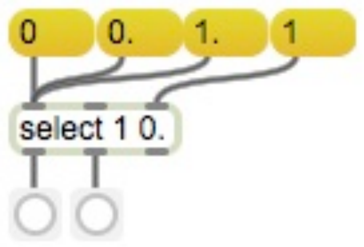


Figure 3.
Max is eager to help you make your connections-- so eager that cords sometime wind up on the wrong inlet (or outlet when you patch upwards).

Objects out of line

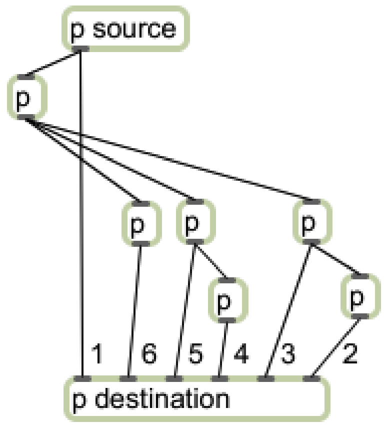


Figure 4.
Since the order of operation is dependent on right to left position of destination, it's not hard to accidently break a working patch by moving an object. This happens to me when I select more objects than I intend to. Figure 4 worked fine until the top p object was slid to the left.

Typos in objects

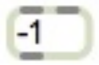


Figure 5.
Missing dots aren't the only way mistyping can really mess you up. Figure 5 was intended to be an object that subtracts 5 from the numbers that come in. It's missing a space between the minus and the 1, so it is actually an int object, and numbers will pass through unchanged. With the new auto-completion feature it's easy to get decide instead of decode.

What's in the Max window?

The Max window will tell you a lot about the health of a patch. However, the messages are cryptic and sometimes misdirect you. Here are some common things you will see:

Newobj name: no such object This comes up when you mistype an object name. It's harmless. You already know the name didn't work.
Patchcord inlet out of range This comes up when you edit a patch and inlets go away. Also harmless.
Someobject: doesn't understand somemessage This means a problem with the data being fed to someobject. Look for objects of this type and double check what is getting sent to them. These will often fill the max window.
Expr: divide by zero detected This is an illegal operation, and usually means an expr has not received data in some inlet. (It often happens to me after I edit the expr). Surprisingly, the division object does not complain about divide by zero, it just divides by 1 instead.

Of course the best things to see in the Max window are those you put there yourself. A print object will copy any message to the Max window. This can be used to follow values in the middle of a patch. You can use an argument in the print object to label the message you are printing. This is the traditional way to find where things are going wrong.

Monitor vital signs

Sometimes the quickest way to debug a patch is to attach display objects to key outputs in the middle of the patch. These let you verify the math or intermediate steps in a jitter process. The most useful display is a message box (use the right inlet). That will show not only the value but type of numbers, and is the only way to see other messages. Button objects are useful to show the rate of messages through the system. You will often discover some section of the patch is not getting triggered when it should. Jitter patches can be debugged with small pwindow objects. The best way to proceed is to start at the top of the patch, checking the output of every object in the same order as the patch execution.

Be sure to remove all of these spy objects when your patch is finished. They slow execution a surprising amount. These displays loose their effectiveness when the patch is executing quickly, because the display will only show the most recent value.

Debugging tools

The debug menu has several tools that can help you find out what is really going on in a patch. Watchpoints are the most powerful. Once the debugger is on, the context menu for a any patchcord (click the button that appears when the cord is selected) will let you put a watchpoint on that cord. There are two kinds of watchpoint:

A *monitor watchpoint* will show the values of data flowing through as in figure 6. The data is shown in two places: in a flag that appears by the patchcord and in the watchpoints window. Each watchpoint is numbered and the display in the watchpoints window shows them in order. It's easy to compare values and see if the data is being processed in the appropriate way.

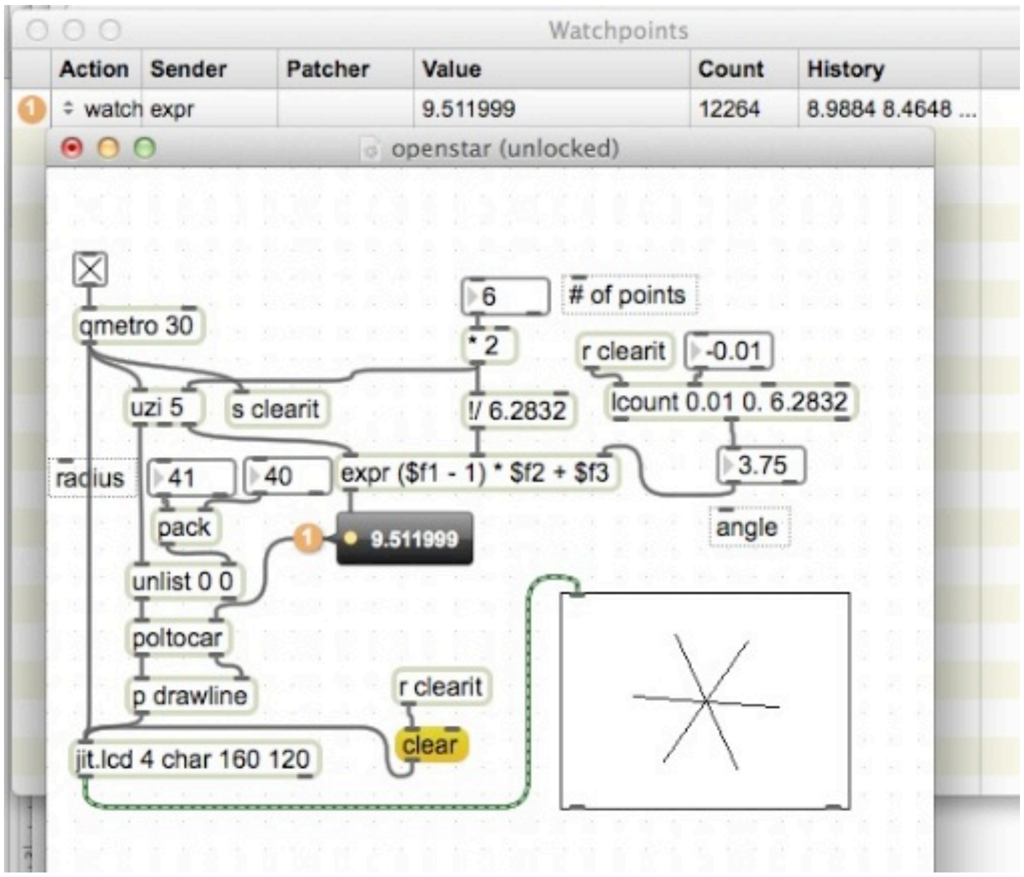


Figure 6.

A breakpoint watchpoint will stop the patch when data gets there. A little green ball will appear (cute sliding animation) to indicate where the patch is stopped as in figure 7. The debug window will show the message at the halt point. When you select step from the debug window (shift command T), execution will resume up to the next cord. Thus you can work cord by cord through your patch and see the order of operation and a listing of all messages along the way.

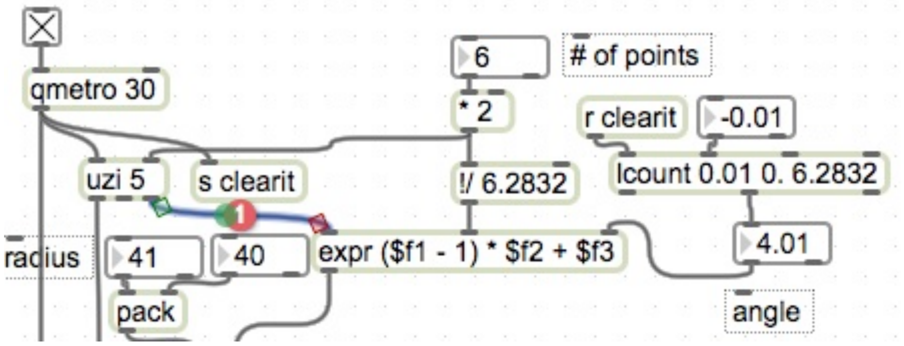
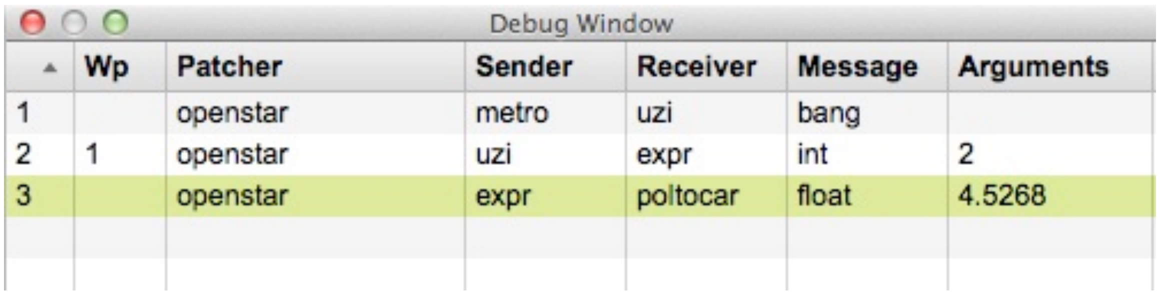


Figure 7.

The debug window is shown in figure 8. In addition to the step command, you can select continue, which will allow operation to run full speed until another break watchpoint is encountered, or autostep, which will move slowly through the patch. (Note: when you use break watchpoints with autostep, it is best to initiate action with a button instead of a metro. All of the metro bangs will be queued up, which will make it difficult to shut stepping down.)



▲	Wp	Patcher	Sender	Receiver	Message	Arguments
1		openstar	metro	uzi	bang	
2	1	openstar	uzi	expr	int	2
3		openstar	expr	poltocar	float	4.5268

Figure 8.

In addition to watchpoints, the debugging window offers probing. When this is on a vu meter and sample values will be displayed when the mouse is over an audio patch cord. The sample values won't tell you much unless they are stuck on 0, which is very important to know. There is also a matrix monitor that will show a (2 dimension) display of matrices when the mouse is over a jitter cord. (Matrix probe appears non-functional in the current build.)

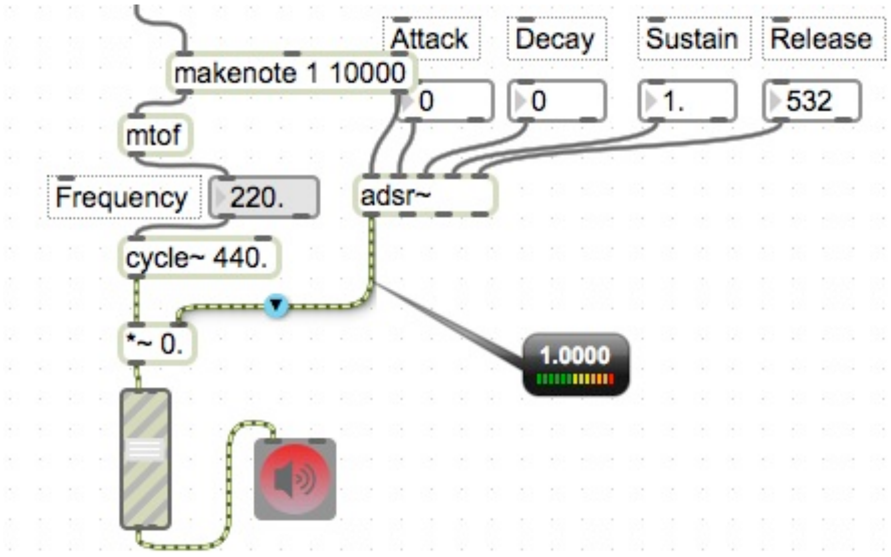


Figure 9.
These tools and a bit of patience will allow you to figure out what is working and what is not. However, this is only the first part of the problem. You also need to ensure that the patch will continue working in the future.

The Robust Patch

The reliability of any computer program diminishes exponentially with its size. In other words, if a patch is doubled in complexity, it is four times more likely to fail. When a patch is functional, you are only half done. There is still a lot of work to do to make the patch give long term, reliable service. The following sections describe the remaining parts of the job.

Initialization

It's very common to get a patch working beautifully, only to discover it is broken after it has been closed and reopened. The usual cause for this is some critical parameter is only set by a user control, or some file has not been loaded.

Critical parameters can be set with arguments in the objects, but that has two severe limitations: the value will not cascade down the patch to set any dependent values, and the user input object will not reflect the starting state of the patch. It is best to set the user controls to a working default state with a loadbang or loadmess object. There are more advanced way to manage UI settings like preset and pattr, but loadbang is the easiest to use and the most reliable.

Files can also be loaded with loadbang or arguments. In some cases you will find the user objects should not be initialized until all files are loaded in. In that case, use delays after the loadbangs. You can test your initialization by including a thispatcher object and sending it the loadbang message. It should put you right back in your initial state.

Dependence

Many of our projects require more than one patch to be open at once. In this situation, you should consider encapsulating all of the patches in a master patch or building a loader path that uses pcontrol to open them all. In any case, there should be comments in every patcher identifying the patchers it works with. You are not going to remember all of this in six months time.

Comments

I cannot overemphasize how important comments are to long term stability of a patch. Here's a scenario I have gone through repeated times.

1. I open an old patch an it doesn't work.
2. I see something really odd.
3. I "fix" the oddity.
4. It still doesn't work.
5. I find the real problem.
6. It works, but badly.
7. I restore what I removed in step 3.
8. Now it's right.

If I had commented my odd coding I would have known that it was deliberate and looked for the problem somewhere else. Comments may seem stupidly obvious, but they never hurt and can save hours of work later on.

Neatness counts

Our patches usually begin looking like a colony of drunken spiders. I am no exception, but the patches that illustrate these tutorials probably give a different impression. I revise my patches to improve readability, since the explanations will make no sense if readers can't follow the patch. If a patch is difficult to understand, any attempt to extend it (or repair it) in the future will likely break something instead.

Once you have a working patch, copy it, and clean up the copy. By "clean up" I mean:

- Pull objects together until you can see the entire patch without scrolling.
- Ensure the patch flows from the top to the bottom of the window.
- Group related objects together.
- Move groups to proper right to left order.
- Remove dead end or redundant objects.
- Encapsulate related objects when that makes sense.
- Guarantee message order with trigger objects
- Replace very long cords with send and receive.
- Segment long cords so they never cross objects.
- Adjust cord crossings to clarify destinations.
- Color code cords that originate from popular sources.
- Comment everything.

At each step, test the copy and compare it with the working version. Moving objects around very often breaks a patch.

Figure 24 and 25 illustrate the benefit of orderly patching. They are exactly the same, and both have the same bug. The patch is an arpeggiator controlled by a k-slider in polyphonic mode. Clicking on a key enters a note into the arpeggio, clicking it again will remove it. The bug is in the velocity which somehow becomes attached to the wrong note[1]. If you click a strong C, then a quiet E, you hear the E but the C becomes quiet. If you study figure 24, you will eventually spot the mistake, but I bet it is immediately apparent in figure 25.

[1] This is a real mistake I made in a real patch, not a made-up example.

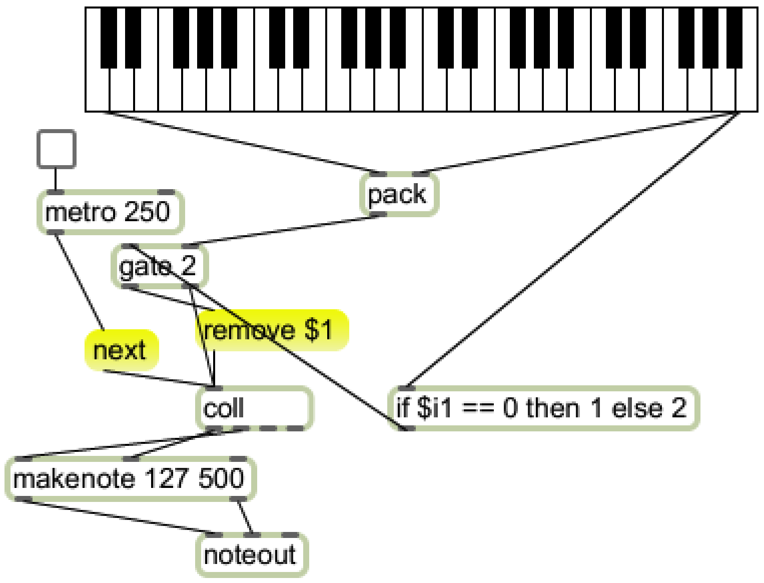


Figure 24. A messy, buggy patch.

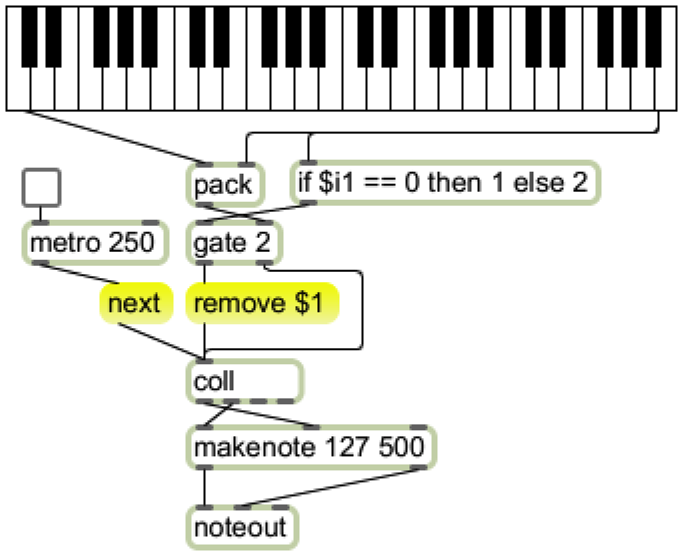


Figure 25. Cleaning up the patch makes it easier to see the bug.

The list generated by the k-slider is stored according to note number, but the coll object emits address before data- thus the pitch will be played by makenote before the velocity arrives. The velocity of the previous note is used. The problem is easily fixed with a swap object between the coll and makenote. Figure 26 shows the correction and comments that will make the operation of the patch clear.

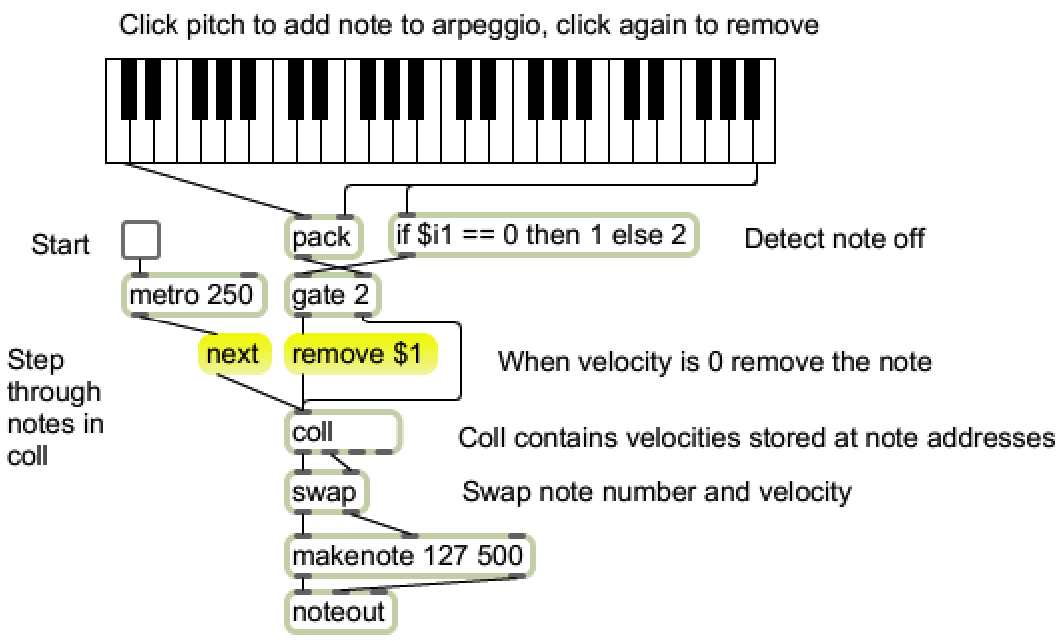

Click pitch to add note to arpeggio, click again to remove
Start
pack
if $i1 == 0 then 1 else 2
Detect note off
metro 250
gate 2
Step through notes in coll
next
remove $1
When velocity is 0 remove the note
coll
Coll contains velocities stored at note addresses
swap
Swap note number and velocity
makenote 127 500
noteout

Figure 26.

So neatness is to your advantage, it's not just the obsession of cranky teachers.

Testing

Testing is a big topic in the professional computing world. Most companies have more testers than programmers. (Cycling 74 has about a dozen employees and more than 100 testers.) A formal testing program usually includes user observation and beta deployment with real world users. We don't need to go to those lengths, but we should at least get our friends to try our patches. You'll be amazed at how quickly they will break. That's because the patch works best with your habitual working methods or you unconsciously avoid troublesome combinations. Ask your friend exactly what was going on when the break occurred. (Make notes when you test his stuff.) Do that yourself to see if it breaks again. It may take several tries to zero in on the exact problem, but once you can reproduce it you can fix it.

The best way to test code is to write an instruction manual. If the manual is any good, it will thoroughly explain what should happen with every control. Writing it will force you try everything out. You will also notice things that could use improvement.