Plug-in Confidential: How I Built My First Plug-in

Gregory Taylor

gregory@cycling74.com

```
<SFX: Evening insect noises, crackling campfire,
distant howling wolves, and the sound of a cheap
boombox playing an Edith Piaf cassette in an RV about
30 yards away.>
```

It seems like only yesterday that I wrote my first plug-in—Well, I guess it was a big yesterday ago at least. Of course, back then some folks thought that writin' plug-ins was propellerhead work, plain and simple. Lots of other otherwise normal folks figgered that using Max and MSP was just too gawldarned Eee-So-Tear-Ick for anybody who didn't ride and shoot with them IRCAM rangers.

But I showed 'em, all right – I showed 'em all. And I can show you, too. Stop gnawin' on that carrot and give a listen, now.

```
<EFX: Campfire codger morphs into some nondescript
bald guy sitting at a table in a Starbucks Coffee
place in a nondescript city in some country where
there are Starbucks coffee places.>

<SFX: Espresso machine noises, distant howling
children, and the sound of a cheap stereo system
playing Antigone Rising in the corner.>
```

True Confessions

Okay. Let's be upfront about this - I do, in fact, work alongside the Cycling '74 people who brought you the 74 mindbendingly cool effects that are Pluggo from time to time. But to put it politely, I'm no more a Max/MSP whiz because I work with Cycling '74 people than your average custodian is qualified to do thoracic surgery because they work at the Mayo Clinic. Should you be thinking of entertaining any notions to the contrary, I expect that looking at the patches I'll show you here (and the clever things or good programming techniques that the examples won't show you) should dispel that notion.

And another thing. This document was originally written 5 years ago, in the wilder and woolier days when building plug-ins was just a bit more difficult than it is now. Heck, there was no Max/MSP for Windows systems back then, and there wasn't even a way to create plug-ins for RTAS (That's the plug-in format that ProTools users use, in case you were wondering). So thing have gotten both easier (in terms of the way you actually build plug-ins) and more interesting (you can now create a plug-in patch on a Macintosh, save

it as a plug-in, and then take the same patch and do the same thing on your Windows system to create a Windows plug-in). Some things haven't changed though—you will still need to know enough about Max/MSP to not stare at a patch like a cow at a passing train. But that's what things like tutorials and documentation are for, and Max/MSP has both of those things in abundance. Then, as now, there is no substitute for learning Max/MSP.

I should also point out that the passage of years has not necessarily made me a better Max/MSP programmer—I am one of those people who build what I need to do my work, and I am neither cursed nor blessed with the fiddly urge to tweak things ad libitum once I have what I need. While I hope that this (very slightly updated) "Diary of a Pluggista" brings your some hope and encouragement and maybe a little amusement, thanks in advance for not guffawing too loudly at what I'm about to show you—my point about showing you everything I've done is a way of saying that it doesn't matter whether or not you're a power user. The most important thing is to do it, and you can always learn to clean up stuff and be more efficient later on. Since my point is to encourage "regular" people to roll up their sleeves and to start making plug-ins, I thought it best to show you the whole process, warts and all – the implication being that if I can do this, you can, too.

This diary isn't intended to replace the lucid, succinct, and distilled Pluggo Developer's Guide, "Developing Plug-ins in Max/MSP for Pluggo." This document isn't quite like one of those tutorial things where some really smart guy starts with a simplified and imaginary task that just happens to show you in some kind of bottom-up order how things work. This is a story of a real guy – me – writing a real plug-in. If I simplified anything, it's to make it easier to follow the patch cords. It's a travelogue. If you want clear and concise stuff with nothing extraneous (such as the foibles of everyday life), read the manual pages. They're great. I wish I'd read 'em more clearly before I started. However, if you want to look over my shoulder while I mess around, read on.

Why Would You Want To Write Plug-ins?

There's a really simple reason that I'd want to try and use Max/MSP to create my own plug-in – no one else is likely to do it for me anytime soon.

Part of that is a result of what you used to have to know in order to even try writing plug-ins, and part of is connected to the plug-in market itself. Since you'd have to at least be a card-carrying member of the Coder Cabal (and it probably didn't hurt if you also had a DSP tag on your Coding License, too) to even think of creating plug-ins under the Old Dispensation, your special qualifications made the plug-in something you could charge people big bucks for - a cursory bit of web-surfing ought to tell you real quickly how much money we're talking about. Plug-in writers in the pre-Pluggo world were being paid to do stuff that was too tough for regular people.

But there's another side effect to that – having these clever DSP programmer guys writing plug-ins for money also meant that the plug-ins they wrote clustered around a rather

limited set of effects. When you think of it as a business proposition, it makes sense – why spend lots of time writing a phase shifter that takes your birthdate and uses the character string as a control input when you could write another parametric EQ plug-in that lots more people would pay you lots more money for? Even if you did get the thing done in record time, there was still the question of what you spent the time doing. So, instead of spinning off quick and idiosyncratic plug-ins, the Old Dispensation plug-in writer did the (arguably) smart thing: he put in the time getting tweaky with Stealth Bomber instrument panel graphics because it made the plug-in look like something you ought to pay a squillion clams for.

That's all well and good. I've got some of those things with the whizzy faux metal front panel graphics in my plug-ins folder, too. But in the course of my work, I kept running into either the limitations of the plug-ins, or I found myself muttering, "I wish I could just [fill in your minor variation or unusual idea of choice here]." And every time I did, I was reminded of the awful truth: the Publisher's Clearing House people were probably far more likely to film me looking at a fat check in dumb amazement than I was likely to have some Coder Cabal plug-in writer insert my precise little tweak or write my odd little effects idea.

Enter Pluggo

Pluggo dropped into the world that I've just described with a bang. I already owned a copy of MSP (a serial number in the low 600s, so I'm only semi-hip) when the announcement went out, so I downloaded the version of Pluggo for MSP owners from the website and installed it.

I'll skip over the next week or so, since they're mostly snapshots of me messing around with the plug-ins one after the other in slack-jawed delight. We should also omit the tales of me running the *Feedback Network* plug-in for entire evenings at a time with no input whatsoever, just listening to the network hum and squeal through *Warpoon's* "Empty Jazz Club" setting (try it.). I did manage to learn a few things in the course of playing around – I figured out the modulation and audio routing and synchronization stuff. This is probably the single unique thing that Pluggo did that really rung my bell - I really liked having the audio input modulating the effects in real time.

And I glanced at the plug-in developer's folder - kind of. It had a couple of MSP tutorials on writing plug-ins that I skimmed (I didn't really read 'em in great detail, as the reader will see). But that was about it until one morning when I woke up with An Idea and A Mission.

Me? With an Idea and a Mission? I'd sooner have figured I'd be translating hieroglyphics than writing plug-ins. In part, this comes from having looked at the original company of 74 quirky/useful/amazing plug-ins that came with Pluggo and being a little awed by the concentration of intelligence. Could I have possibly imagined myself doing any of the

clever things or designing any of the cool interfaces that jhno or David Zicarelli or Adam Schabtach did? Frankly, no.

Having finished my first plug-in and having a couple of other ones in various states of development now, I see that my problem was one of perspective: I was looking at plug-ins that were the end of a long process of refinement and imagining them to be first tries created by people whose talents I couldn't even approximate. I also imagine that the process of making the darned things was insanely difficult for anyone but the kind of person who eats, sleeps, and breathes Max/MSP.

Guess what? Those were mistaken assumptions.

So, what happened to make me into a plug-in author? Simple. I was puttering around one afternoon and realized that there was something I wanted – a certain kind of plug-in. I could imagine the thing because I know enough Max/MSP to have used the little patch/object before, and I knew what sorts of things I might do with it (playing with using Pluggo's modulators made that part easy). It seemed like a simple little thing that maybe nobody else would think of or find interesting, something that I could do. So I got the developer's material stuff and the tutorial patches, and got to work.

This document is a record of that work – how it started, how it went in fits and starts, and what I learned along the way.

Daddy, Where Do Plug-Ins Come From?

There are a couple of places to start when you think about creating plug-ins, but the first part is always the same – desire. While your desire to have the thing is the reason you'd put in the effort to actually mess about and do it, there are a number of places I could have started from, and the same is probably true for you:

1. Take something you've already made with MSP that you like, and make it into a plug-in. That way, all you're really tangling with is learning the Pluggo development stuff. After I'd finished with my first plug-in, I actually sat down and read the whole Pluggo Developer's Guide and discovered that the author had been nice enough to actually give you a list of what Max/MSP stuff you can't use (it's Appendix A). Fortunately, my initial idea didn't run aground on this one.

2. Take a patch from somewhere that does something really cool and make a plug-in out of that (you get to learn the thing you're interested in and Pluggo development stuff). The next time you download an elegant example of how to do something from the Max mailing list and you're fooling around with it and comparing it to what you thought you had, think of making it into a plug-in.

3. Start with some process you normally use for your work – say, in the recording studio – and turn it into a plug-in. The familiarity with the procedures you've used for a long

time is what you start from, and you structure the experience of teaching yourself something about Max/MSP around that. Then you add the Pluggo stuff.

Interestingly, the American composer Carl Stone told me that that's exactly how he learned MSP – he simply made things which replaced the equipment he used to compose and tour with all the time. Luke DuBois, erstwhile member of the Freight Elevator Quartet, sat down and made himself something that'd look and act like the wonderful but unwieldy Serge modular synth he knew and loved so well. This seems to be a good strategy.

In my case, I've often made these huge walls of noise for background stuff by taking a couple or reverb boxes cranked all the way up and then EQ'd the signal coming outta the first one into the second, and EQ'd the output from the second. It makes a great wall of slowly blooming noise for use as background noise clouds or convolution raw material. So guess what my current plug-in (in progress) is?

Introducing Messrs. Navier and Stokes to Pluggo

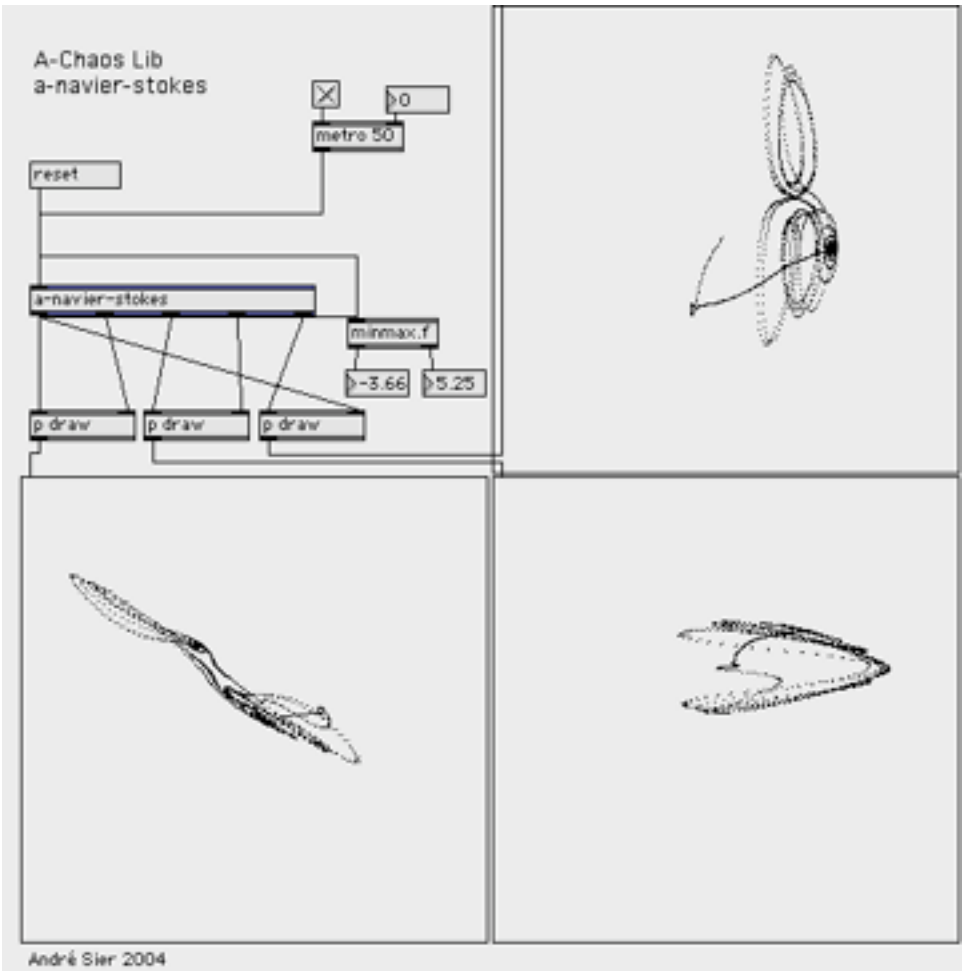
One of my dark little secrets is that I learned nearly everything I know about Max/MSP by imitation – I opened up the help files, copied the stuff I needed out, got rid of what I didn't need, and messed around with what I had until I understood enough of what it did to do what I wanted. Then, I broke the patch and wound up learning how things worked as I struggled to fix what I'd broken. There – I've said it, and I feel a lot better.

So if you're thinking that my interest in chaos and stuff like the Navier-Stokes equation comes from my screamin' Math background, forget it. I got into chaos by downloading something from the net and goofing around with it.

In the 90s, Richard Dudas (who does eat, sleep, and think in MSP, by the way) put together and uploaded a collection of chaotic objects based on the work of Mikhail Malt, which began its life as part of IRCAM's PatchWork software. More recently, André Sier ported these objects, along with others, to OSX and Windows. I commend them to your attention. The package of documents, objects, and patches that come with this document contains a copy of the Chaos object I used. At some point, you might want to download the whole set of them from the Cycling '74 Share area at

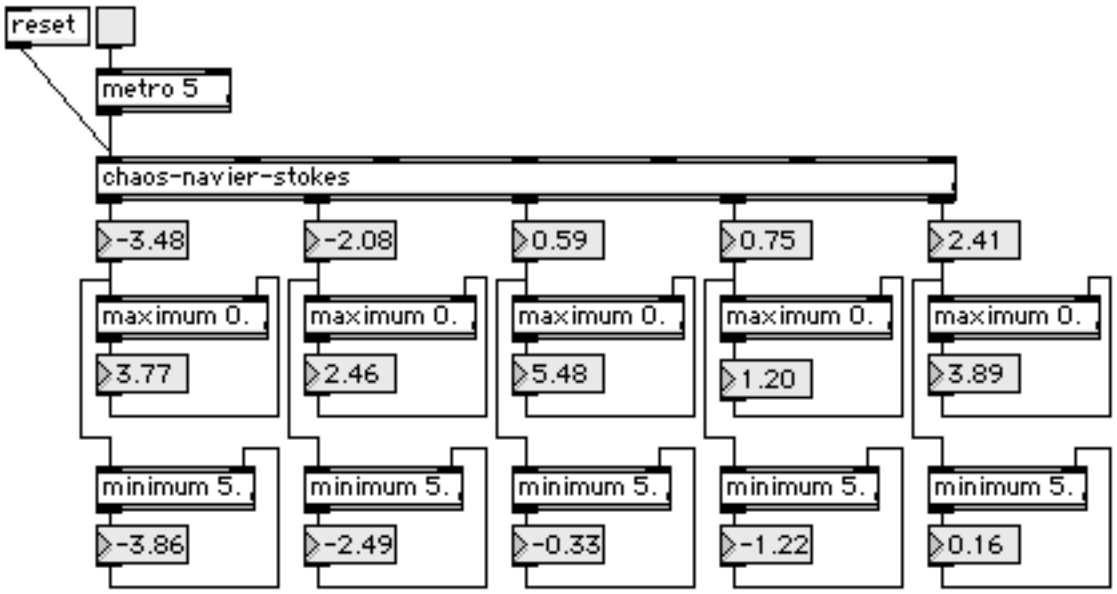
http://www.cycling74.com/twiki/bin/view/Share/AndreSier

One of the objects in his ChaosCollection file that really caught my fancy is **a-navier-stokes**, which drew these lovely pictures from the output of the Navier-Stokes equation:

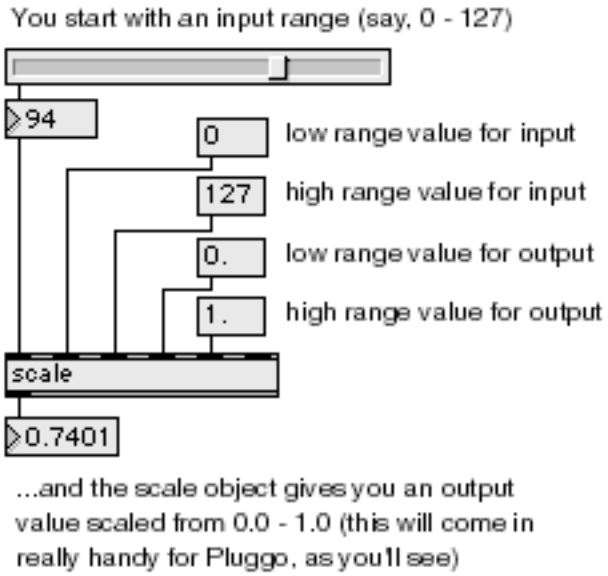


I initially downloaded the ChaosCollection because I was curious, and because Richard and André had helpfully included both the chaotic objects and examples of how they could be implemented as a Max patch (really useful for looking at how you do complex things with the **expr** object, by the way). So I horsed around with the help file patches and drew lots of little demo pictures. It was fun, but I began to be curious about the maximum and minimum ranges for the outputs of the equation. While they were fun to look at, the pictures wouldn't tell me that. So, in true Max form, I started to tinker - I'd took the original help file for the **a-navier-stokes** object and then disconnected the whizzy graphics stuff. I set up this patch, turned it on, and then went out to dinner and a movie.

It probably ran for 6 hours. When I came back, I had a set of maximum and minimum values for the vanilla object. I figured that if I could take that information and scale the outputs into the 0-127 range, I'd have a cool thing that I could do lots of stuff with.



Hardly even before I turned my mind to making a little patcher that did scaling, a friend on the Max/MSP mailing list turned me onto the **scale** object. One look should suggest why I liked it – it did exactly what I wanted:



I sat down with my list of maximum and minimum values and my newfound **scale** object and knocked out a simple all-purpose chaotic widget to do interesting tasks like moving the faders around on my mixing board, or racing up and down the scale in the form of a MIDI santoor.

So I had this cool little chaotic widget sitting around. I think that the last time I actually used it was to mix and pan a bunch of talk radio programs I'd run into my mixing board, with occasionally hilarious results. This happened before Pluggo ever came into my life. It was just a simple thing that I'd built and used.

But on that fateful Saturday, something hit me – **I could take that cool thing I'd slapped together and use it to control Pluggo plug-ins**. I could maybe write a modulator plug-in. And with that, I became a plug-in developer - while I was faint-hearted when it came to imagining myself thinking up cool MSP DSP objects, I could definitely imagine myself doing this – after all, I'd done lots of the work already.

Preparations For The Journey

So I sat down and skimmed the Pluggo Developer's Guide stuff. I checked the list in Appendix A to make sure that none of the Max objects I wanted to use weren't included. No problem. I noticed that there were tutorial files full of examples I could borrow, and help files I could loot and cut and paste, if need be. Some stuff didn't make sense to me – probably because I was reading about them instead of using them, and some of it looked pretty straight ahead. I made a simple plan for myself – I'd make a quick, dirty, and simple chaotic modulator program I could imagine, and then add some stuff if I felt like it. As long as I had the **a-navier-stokes**-based stuff sitting around in my Max patches folder, I figured I'd start with that. Here's what I decided:

1. Although chaotic attractors exhibit interesting and different behaviors depending on their initial variables (the classy term for it is "Sensitive Dependence on Initial Condition"), I'd just use the **a-navier-stokes** object in its vanilla form with nothing but the default variables. No presets.

2. I wanted some kind of visual display that I could use to track the value of the scaled outputs. I wanted to keep things dead simple – I'd just use hsliders to display all the equation's outputs. While the egg slider interface was pretty cool, I thought I'd better make my own. Fortunately, I had the P3 Tutorial, which provided examples of how that worked.

3. There were a few housekeeping things about the **a-navier-stokes** object. I wanted to be able to adjust the rate at which the **a-navier-stokes** object spit out values, and figured that I'd just start the thing calculating the moment it was started up, and to provide a reset button (after all, the original help file had it….).

4. Finally, I realized that there **was** something that I wanted to add to my original little chaotic widget object that I didn't already have – the ability to narrow the range over which the slider's outputs move back and forth. Since I figured that would be something I'd always be tweaking on a very individual basis, I didn't need to set up any presets, either.

With these modest expectations, I set to work.

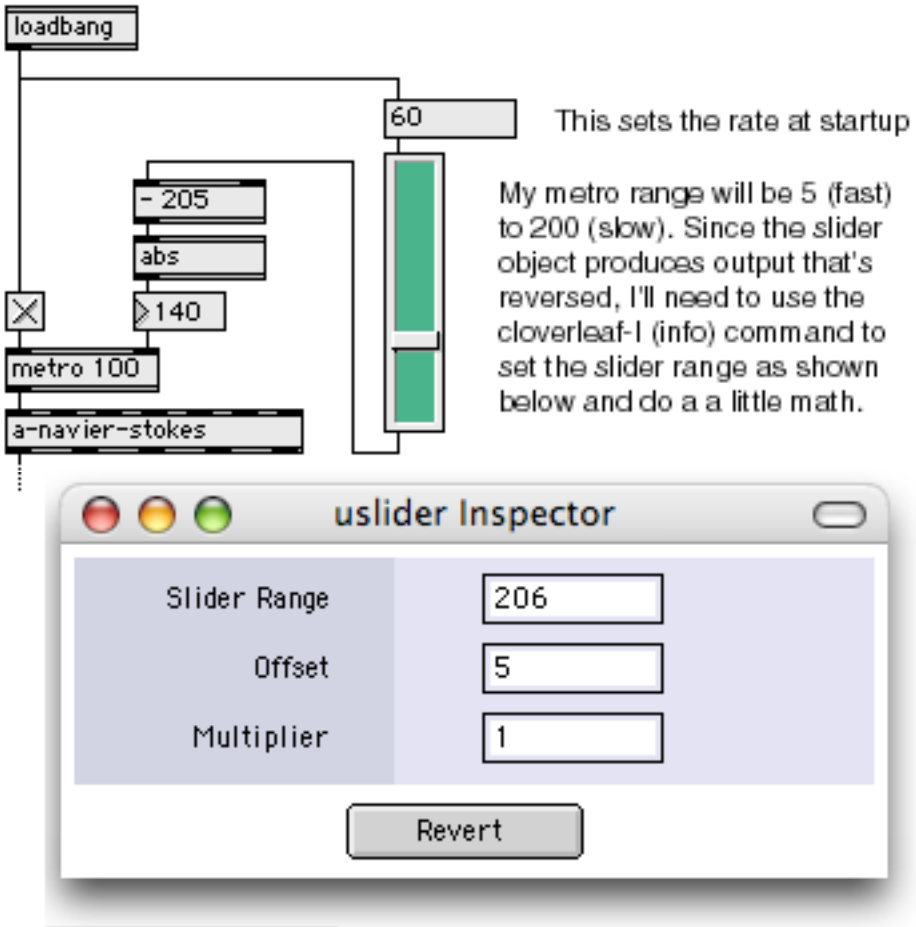
This is how the patch starts
this is how the patch starts
this is how the patch starts
not with a whimper but a loadbang.
-T.S. "Max" Eliot, *The Hollow Men*

Since I'd decided to go for a really simple bare bones plug-in, some of the next stuff was really simple. I opened a new Max/MSP patch, dropped my little chaotic widget in, and

stripped off all the scaling and MIDI stuff. Now, I was ready to start to add or modify the pieces I'd need for my user interface.

First, there was the question of how fast I wanted my plug-in to kick out the values I'd be using to modulate other plug-ins. I decided to set an arbitrary range for the **metro** object that controlled how fast the **a-navier-stokes** object kicked out values of 5 (fast) to 200 (slow). I could have chosen anything, of course. What I needed to do was to set up a slider object to let me set the **metro** rate. But the vanilla slider object worked a little differently from what I needed – I wanted the bottom end of the slider to produce high numbers (for a slow rate) and the top end to produce low numbers (for a rapid rate of calculations). The normal slider produces values from 0 (at the bottom) to 127 at the top. No problem there – Max lets you set the total range and an offset for the slider object. I just added a little math to the slider output (some subtraction and an absolute value operation) to set up my rate slider.

I also set up the **loadbang** object so that it would automatically set a starting level I liked and start the metronome working. Here's what I've added to part of my chaotic widget patch:



NOTE: I've included real live, gen-you-wine copies of the Max patches I used to make my plug-in in its stages of development, so you could just open up the patches and examine them for edification or laughs. But they are sort of complicated, since there are 5 sets of sliders and plugmod objects, and stuff is hooked together so that all the user interface objects are in one tiny area of the patch. That makes things really tough to figure out, especially if you've got any degree of trepidation when it comes to looking at and figuring out a Max patch. So the examples you'll see here are slightly simplified compared to what's in the Max files and visually arranged so that the flow is a little clearer. If you're the kind of person who doesn't go pale when you click on the "unlock" button of a Max patch and a rat's nest of patch cords appears, then go crazy with the originals.

Next, the fun began in earnest – my bright idea of constraining the output ranges for each of the five calculated outputs meant that I had some work to do. A constrained output range would mean that I'd have to be able to choose a different low and high range for my output – something other than 0-127. I remembered that I'd seen the Max **rslider** used in a couple of Pluggo plug-ins, so I figured that I'd go with what seemed to work. I could use the **rslider** object to set the high and low output ranges that I'd send to the **scale** object, and then route the output from the **scale** object to an **hslider**, which would display the output behavior over time.

It became quickly apparent from my skim of the Pluggo Developer's Guide that the values I'll need to work with will have to be expressed in the 0.0 –1.0 range. That means that I'd need to take whatever output that the **a-navier-stokes** object kicks out, and then scale that so it fell within the constrained range from my rslider.

And I've gotta do it five times – once for each of the **a-navier-stokes** object outputs. Looks like this could be a job for a patcher, which I'll call NS_scale. I want the patcher to have the following inputs:

1. The maximum and minimum values that I know the **a-navier-stokes** object will kick out (the ones I got back in the old days that I used for my MIDI patch, remember?). I'll use the **loadbang** object and a message (which I'll unpack and route) to set those values at startup time.

2. The maximum and minimum values from my constraining **rslider**. The **rslider** kicks out values in the 0 – 127 range, of course.

3. A value to scale. The arrival of that value should be the thing that kicks off the calculation.

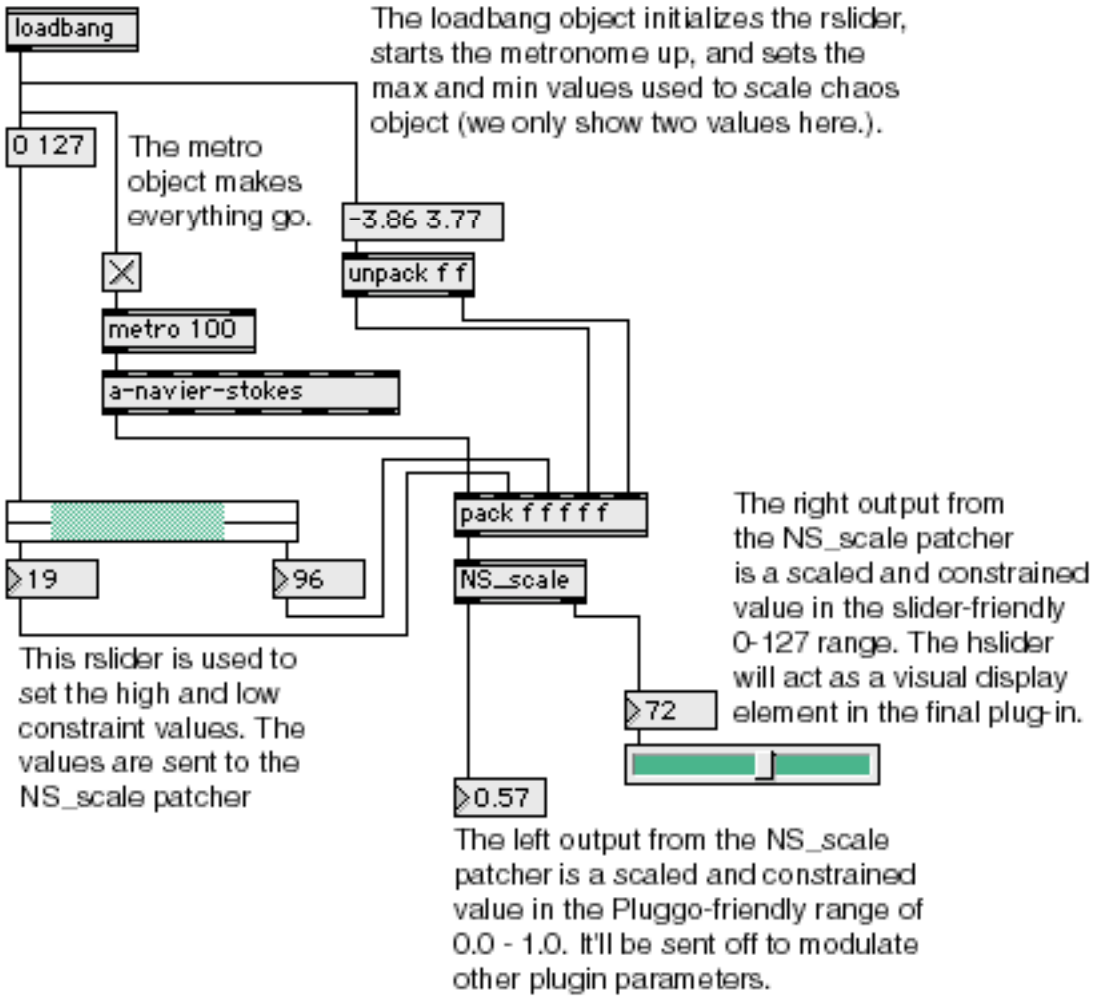
For outputs, I need:

1. A scaled output value in the 0.0 - 1.0 range for Pluggo stuff (that's what I'd feed to the other plug-ins)

2. An output in the 0-127 range to drive the **hslider** I use for display.

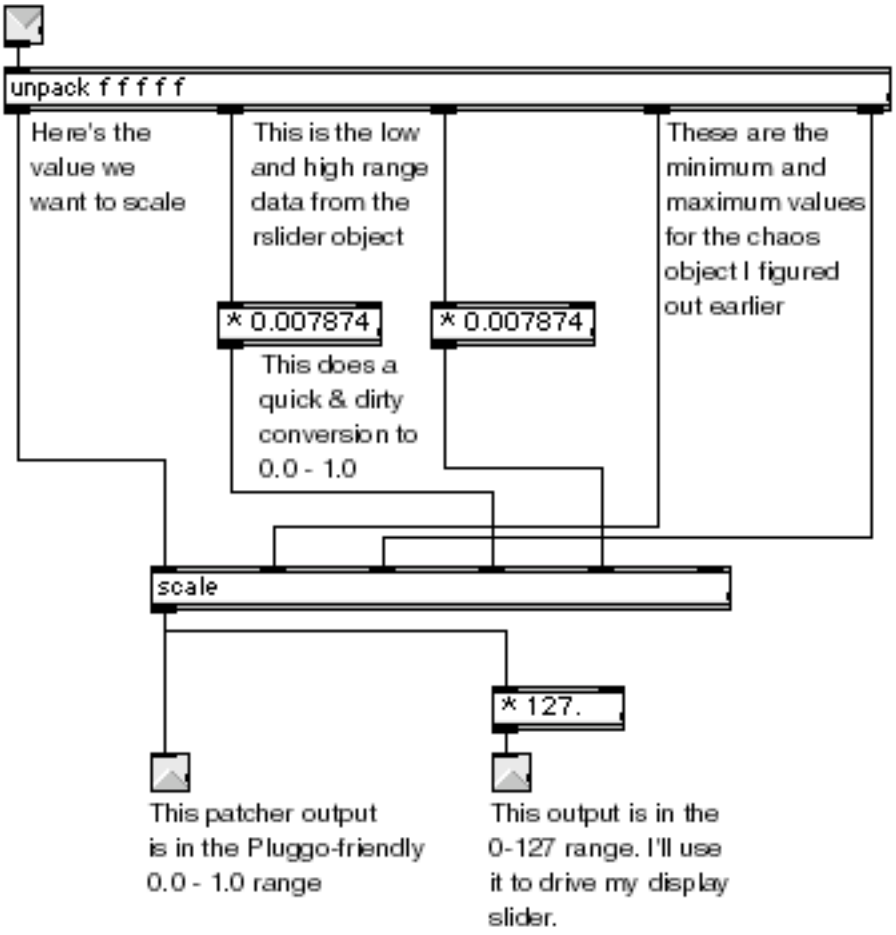
My smarter friends remind me regularly that good Max programming habits involve keeping the number of patcher inputs and outputs to a reasonable and clear minimum. Setting up my patcher so that I can use the **pack** object to feed my input data through a single inlet (which is then unpacked and worked on) allows me to control the order in which things arrive and are updated in addition to looking a little cleaner.

Here's a simplified version of what I had in mind:

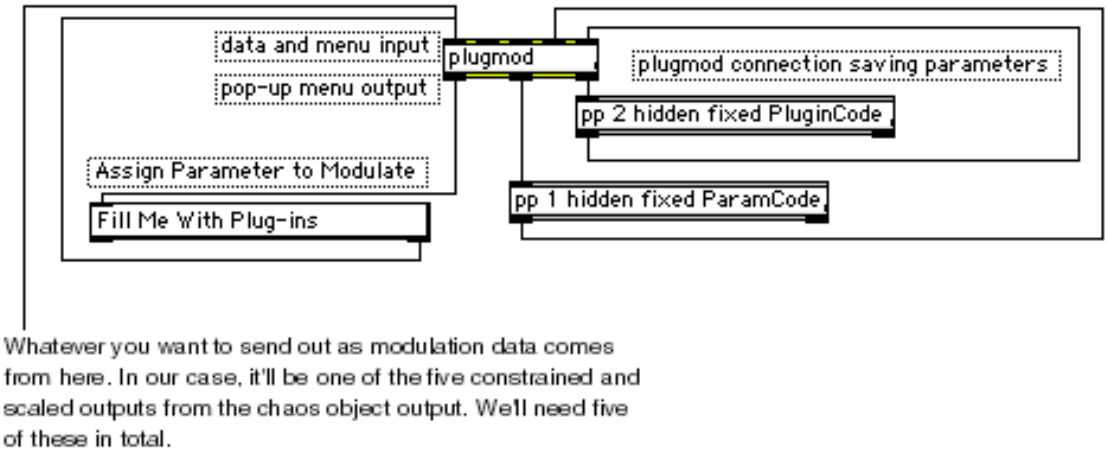


NOTE: This example shows only one of the five outputs of the **a-navier-stokes** object for the sake of clarity. The copies of the Max patches I've included use one **rslider**, one NS_scale patcher, and one **hslider** for each of the **a-navier-stokes** object's outputs.

With that in mind, it wasn't too difficult to slap together my NS_scale patcher:

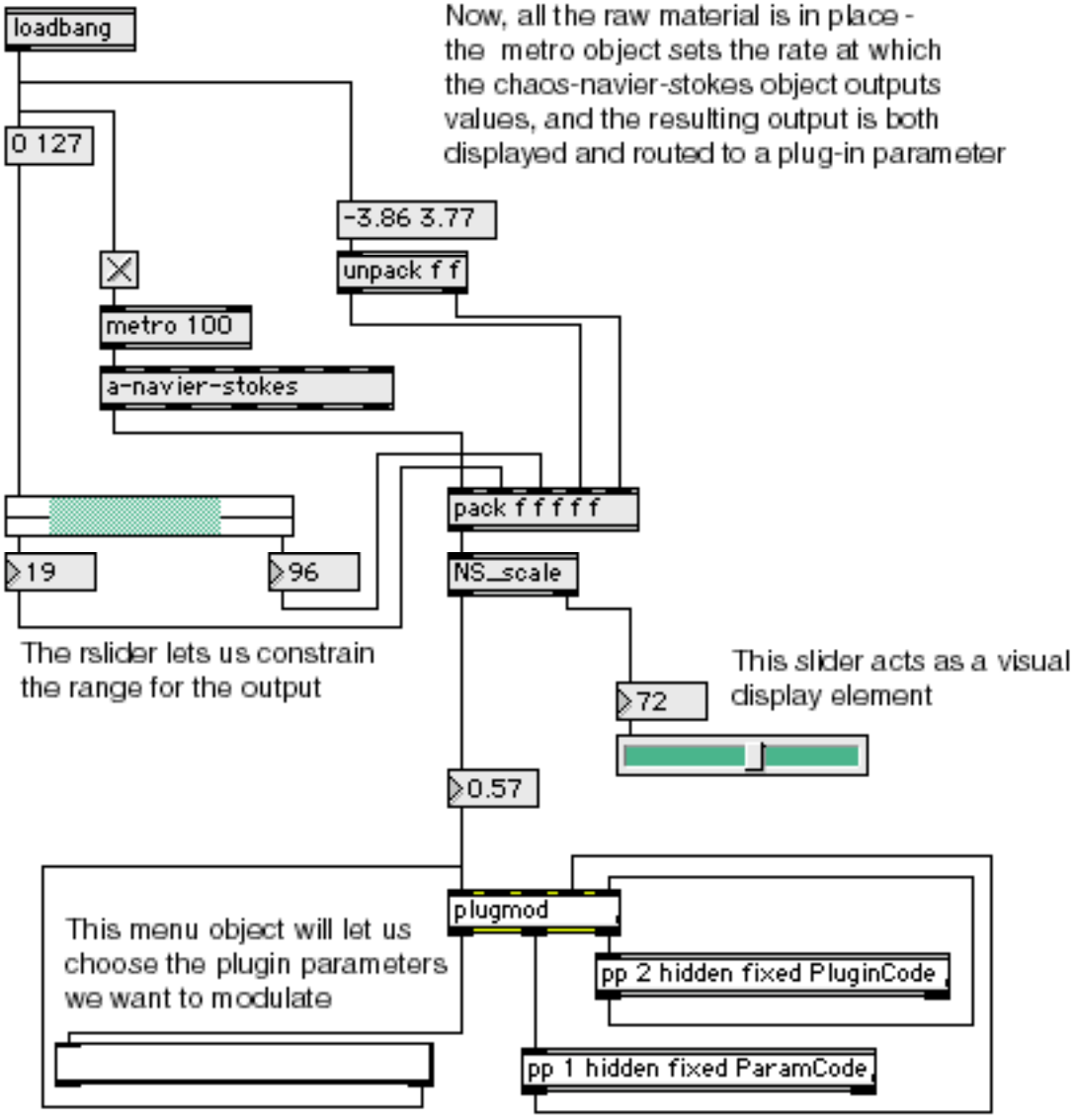


Now, how do I set things up so that I route the output value to a plug-in? Once again, we resort to time-saving intelligent plunder. The Pluggo Developer's Guide tutorial P5 contains an example of how you handle output to modulate other plug-ins using the **plugmod** object. Here's what the basic stuff looks like:



In the tutorial, only one modulator menu was used, and I needed five – one for each of my chaos object's outputs. In tried and true fashion, I simply copied the stuff I needed from the tutorial, and pasted it into my patch. There is one piece of housekeeping I had to take care of – each **plugmod** object and connected menu needed a uniquely numbered pair of plug-in parameters (in the example above, they're **pp** objects 1 and 2. So I simply pasted my five copies into the patch, and renumbered the **pp** boxes in pairs (pp 1 and pp 2 for the outputs of NS_scale patcher 1, pp 3 and pp 4 for the output of NS_scale patcher 2, and so on). Finally, I connected the data and menu input on each of my five plugmod objects to the scaled and constrained outputs of my NS_scale object (which outputs data from its left output in the 0.0 – 1.0 range that I need).

Here's an example of the whole data flow with the new additions. Again, I'm only showing you the output for a single output of my chaos object:



In Your (Inter)face –Making a Front Panel

I was pretty sure that I now had the nuts and bolts of a plug-in ready to go, so I turned my attention to what I'd have to do to set up the user interface part. I decided at the start that I wanted to use the **rslider** and **hslider** objects to control and display the output of the **a-navier-stokes** object, so I needed to make my own user interface instead of going with the more traditional Pluggo Egg Slider interface. I took a little break and took a look at David Zicarelli's third tutorial in the Pluggo Developer's Guide, where he sets up a user interface using a bunch of sliders – just like I was planning on doing.

The tutorial suggested that you clear out some space and move all the interface sliders and buttons and displays together. So I took all the component parts and started moving them around to make a simple front panel's worth of stuff in the upper left corner of my Max/MSP patch window (the tutorial told me I'd have to calculate how much space my interface panel needed in pixels, and I thought it'd be easiest to just start from 0,0, the upper left-hand corner). Moving the interface objects around so that they were all grouped together and all the other stuff was "offstage" made a rat's nest of patch cords out of my patch, of course. And I did remember that I'd need to select every single patch cord which stretched across my front panel and set it for "Hide on Lock" (cloverleaf-K) so that the campsite would be clean when I closed the patch. If you're feeling really brave, I've included the real first-run patch and the NS_scale patcher in the Version_1 folder for your edification. I made no attempt to clean it up or comment it or reroute the wires (although I did for the final version) – I include it for the curious. In the interest of cleanliness, I'll only show you the clean and locked version here.

The tutorial 3 spent a lot of time describing the contents of the **plugconfig** object, since that's where you put information that describes what kind of interface you're using, and how large it is. So in my usual fashion, I dropped a **plugconfig** object into my patch, and double-clicked it. When you do that, you get a copy of the vanilla plugconfig file, which looks like this:

```
#C useviews 1 1 1 1;
#C numprograms 64;
#C preempt 1;
#C sigvschange 1;
#C sigvsdefault 32;
#C autosize;
#C defaultview Interface 0 0 0;
#C dragscroll 1;
#C noinfo;
#C uniqueid 0 4 165;
#C initialpgm 1;
```

My reading of the third Pluggo tutorial suggested that I'd need to change a few things for my plug-in:

First, I wasn't going to use the traditional Pluggo Eggslider interface at all, so I'd need to turn it off. Changing the first line to `#C useviews 0 1 1 1;` does this. It turns off the default view, and will use the front panel I'm making

I wanted to change the number of default programs my plug-in would use. Since I wasn't using any parameter presets at all, I thought I'd keep the number to two, so that I could set up two different versions of my plug-in's settings and toggle between then while I was using my plug-in. That's a simple number change – the line reads `#C numprograms 2;`

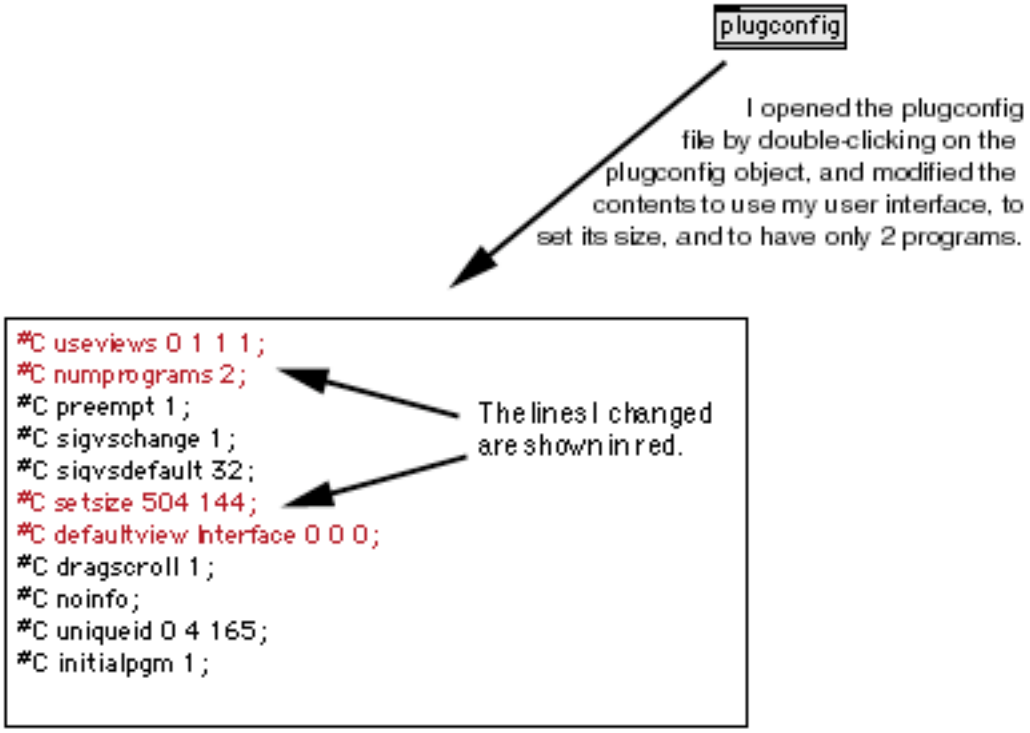
Next, I needed two things to define the front panel graphics for my user interface – a size for my user interface window, and a default view area that describes where the upper left-hand corner of my user interface panel would be.

First, I needed to specify how large it was going to be. The line `#C autosize;` in the plugconfig file set a default interface size. I couldn't use the default size – all those horizontally grouped sliders meant that I had gone for the Prairie Style interface design – low and horizontal.

I guessed at my front panel size by getting out a ruler and measuring it (really). It looked about 7" wide and 2" high, so I calculated that at 72 pixels/inch and used a figure of 504 x 144. With those numbers in mind, I changed the line to read `#C setsize 450 144;` to match the window size I wanted.

The other thing I needed to do was to specify where the "window" that had all my interface objects in it would be. The Pluggo Developer's Guide said this was a real case of trial and error. I figured that the easiest thing to do would be to assume that the very top upper left-hand corner of my patch would be where I'd locate the upper left-hand corner of my user interface (I was keeping it simple, remember). So I added `#C defaultview Interface 0 0 0 0;` to my plugconfig file.

Here's what my new plugconfig file looked like. I've highlighted the changes I made from the original vanilla file in red:

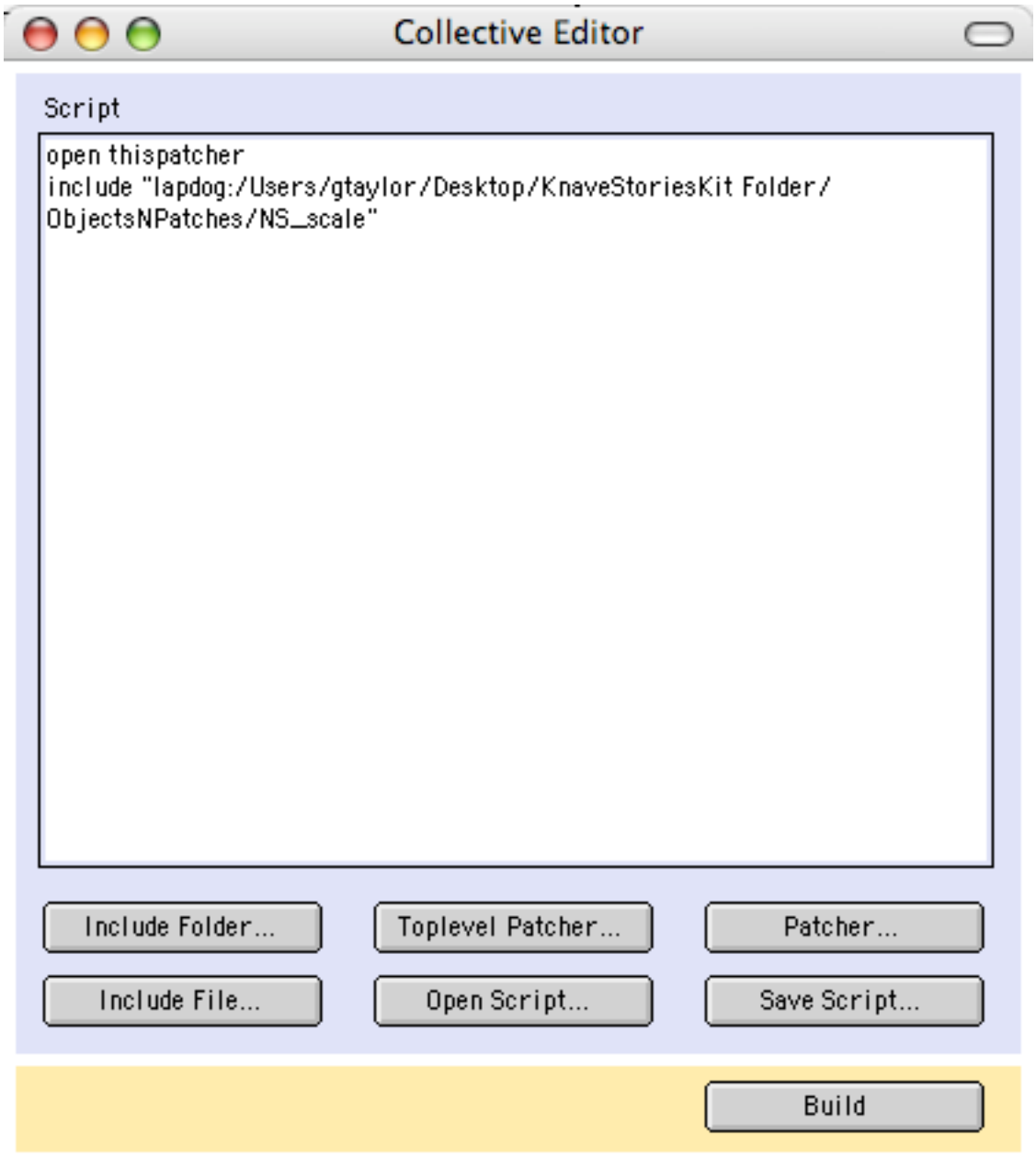


So I set up the plugconfig file, saved it, and figured I was ready to go.

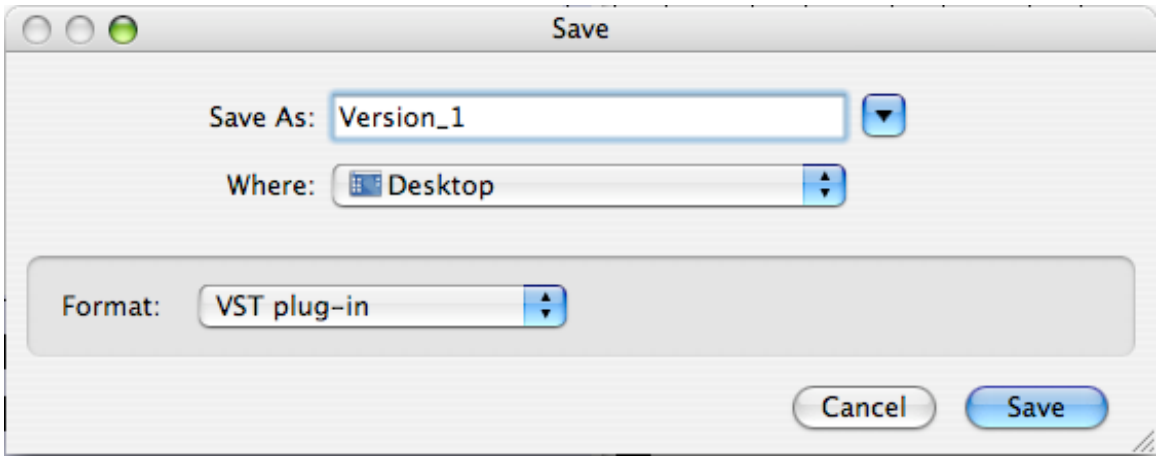
Adding Only What You Need

When you save a Max patch as a VST plug-in (It works as an Audio Units OR and RTAS patch. Don't worry), Max has the ability to remove selected external objects we're not using at all from the resulting plug-in file, which saves a bunch of disk space and lets Pluggo host a squillion plug-ins in your sequencer. The default situation is that *all* external objects are removed from the resulting plug-in file unless you specify otherwise. But if you're going to make a plug-in that contains a Max or MSP object that not everyone else has, you've got a problem. In the case of the plug-in I wrote, there were, in fact, a Max object that I was using that weren't vanilla Max/MSP distribution objects: the a-navier-stokes object from Andre Sier that started the whole thing off. So that means I could tell that there is a certain non-vanilla Max external object I want to include in resulting plug-in. I have a similar problem, occasioned by my developing the nice NS_scale patcher that's used to handle the range constraints and scaling. No one else in the world is going to have that patch, so I'll have to find a way to include it when my plug-in is made.

I used the *Build Collective / Application / Plug-In....* command from Max's file menu. I clicked on the *Include File…* button in the collective save dialog and then selected the NS_scale file. Here's what my collective script file looked like:



I clicked Build and the Save window appeared. I chose VST plug-in from the Format pop-up menu, and presto! A plug-in is born!



Gettin' Tweaky Wid It

So I saved my patch as a VST Plug-in, and dropped the resulting file into my VST plug-ins folder.I started up Peak, and loaded my new plug-in. What I got didn't exactly thrill me. The thing sure seemed to be running like a champ, and it looked like the window was the right size, but the controls weren't in the right place. Arrrgh! What did I do wrong? This one took lots of time to figure out, and it's important:

IMPORTANT THING THAT THE MANUAL DIDN"T MAKE CLEAR

The variables you put in the plugconfig file in terms of the "location" of your front panel are relative – 0 0 0 is understood to mean the upper left hand corner of the window at the point you save and close the file. If you've got a big workspace and scroll to the bottom right corner, the *Plugmaker* will assume that 0 0 0 is the upper left-hand corner of your Max window - NOT the upper left-hand corner of the whole workspace. Take a look at the window in the example that shows you the contents of the plugconfig file. Now, look at the relative placement of the user interface objects. Look familiar? That's what I mean. If you've got a large file and you don't know this, it will make you crazy until you figure it out.

So I reopened my file, and scrolled to the upper left hand corner of the workspace, saved my file and closed it, saved it as a VST plug-in again, dropped it into the Plug-ins folder and launched Peak. This time, things were looking much better, but I'd failed to take into account that I'd need to leave some blank space at the top of the window to make room for the View and Meter bar that Pluggo gives me. So, it was back to the cycle of opening my patch, moving stuff around to tweak the way the plug-in looks, making sure the frame view is set to the far upper-left hand corner of the patch window, saving, etc. Actually, this tweak loop wound up taking some time – I discovered, to my surprise, that I liked messing around with the interface design. You might discover that this is the case for you, too (uh, maybe not). After a bunch of iterations, I had it! The front panel looked great, and I could pull down each of the menus to assign modulator outputs for any plug-ins I had on my mixing board. I'd done it. I'd actually written a plug-in that really worked.

Is That All There Is?

How did I celebrate the completion of my very first plug-in?

First, I cracked open a celebratory microbrew. Next, I put my cool plug-in to work - I spent a pleasant evening listening to my plug-in modulate jhno's Feedback Network (with the output run through the "Empty Jazz Club" program from Warpoon, natch), and taught it to create huge swirling clouds of chaff with the Rye plug-in.

It just seemed a shame to saddle this cool thing with a silly name like Version_1, though. A couple of minutes with my cold one, a pen, and a napkin rewarded me with a pleasing anagram for Navier-Stokes – KnaveStories. But did my initial flush of success outlast the beer? Well, yeah. A little. But almost the second that I fired up Version_1 for real, I started to see things that I wanted to add:

The most obvious one was that I thought it'd be really cool to be able to use PluggoSync to drive the plug-in instead of a metronome.

Playing with my plug-in meant that I spent some time staring at the other 74 Pluggo plug-in front panels. While using my plug-in to modulate all those other Pluggo plug-ins, I noticed something – a number of them were using the **multislider** object for display instead of my bank of hsliders. I hadn't messed around with the **multislider** object very much, so I opened the help file for it and took a look. Wow! It's not that hard to use (just take a list of incoming variables, pack 'em up, and you're in business), and there's one big advantage - I can set it to accept the same output range as Pluggo expects (0.0 – 1.0).

When I first started my plug-in, I'd decided that I really didn't think I needed any parameter presets. But what I did notice was that while I used the range sliders a lot, the settings were very specific to the plug-ins. While I really didn't think I needed to add any parameters, I thought it would be a good idea to add a switch that would reset the rsliders to their maximum range.

Oh yeah - I'd like to be able to stop and start my plug-in from the front panel.

Throwing in the (Kitchen) Sync

I was only halfway through my celebratory cold libation when I decided on my first official plug-in improvement. I wanted my plug-in to be able to use the output of the *PluggoSync* plug-in to trigger the **a-navier-stokes** object calculations. Unlike what I'd done so far, this one actually involved sitting down and taking a serious look at the manual pages in the Pluggo Developer's Guide, but proved to be a snap.

In case you're not familiar with it, *PluggoSync* is another modulator/control plug-in that lets you keep other plug-ins in sync – either with other plug-ins, or your host sequencer. Here's the front panel of *PluggoSync*:

The stuff that interests me is at the bottom of the panel – I can take a tempo (set either as an Internal BPM clock or as an audio sync pulse using that purple arrow at the top of the panel), divide it down by one of 5 values I can set (in the Beat Divider column), and output them. From the standpoint of someone (like me) who writes Max patches, it's pretty familiar – *PluggoSync* actually outputs Max messages using the **send** object. There are a bunch of them, but all you need to do is figure out which of *PluggoSync's* output messages you want, and then set up a **receive** object to receive the output from *PluggoSync's* **send** objects.

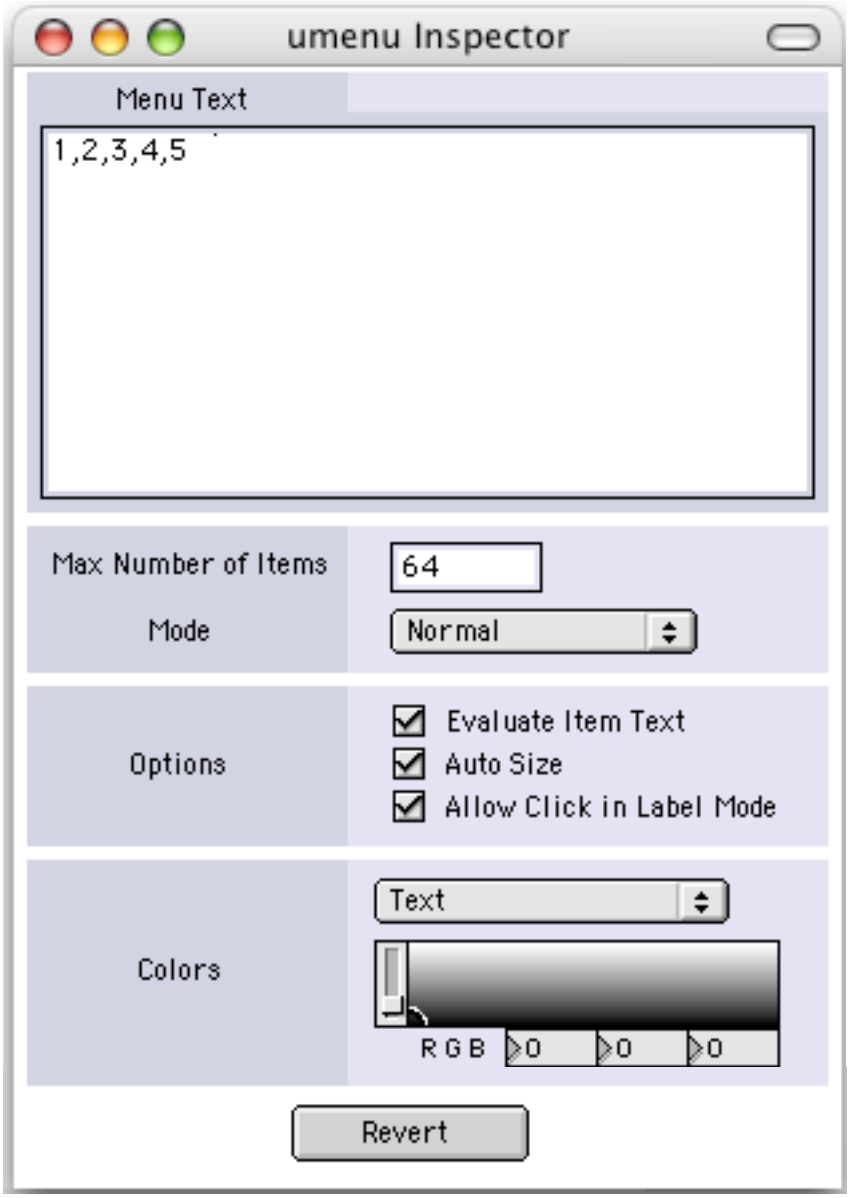
According to the manual pages, there is a **send** object associated with each of the five *PluggoSync* messages. Unsurprisingly, they're named

plugsync_1
plugsync_2
plugsync_3
plugsync_4
plugsync_5

In order to use any of these messages in my plug-in, all I need to do is to use a **receive** object with the name corresponding to the information I want. I can then route the incoming bangs to the **a-navier-stokes** object to trigger the calculation.

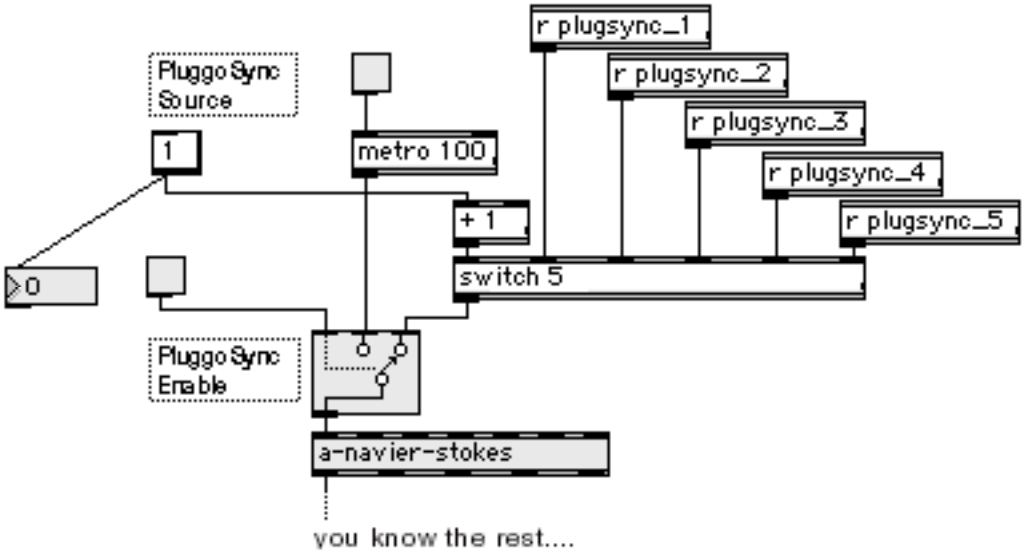
I think I'll use the menu object to select the number of the *PluggoSync* output I want to use – a quick and simple box with 5 choices. I drop the menu item into my patcher, and a window opens up to set up the contents.

All I want is 5 numbers:

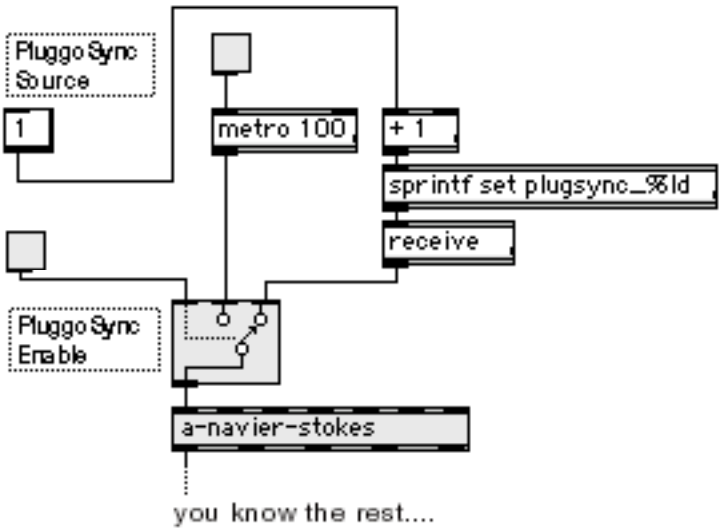


So now I've got a little pull-down box with 5 choices. What do I do with it? Hmmm…I've got five different **send** objects and a signal whose messages I want to receive and route. I could take the output of a menu object and use the index value of the menu object along with the gate object to choose which input to route. And by the way –

I'll need to add one to the output of the menu object, since the first menu element has an index of 0. It looks like this:

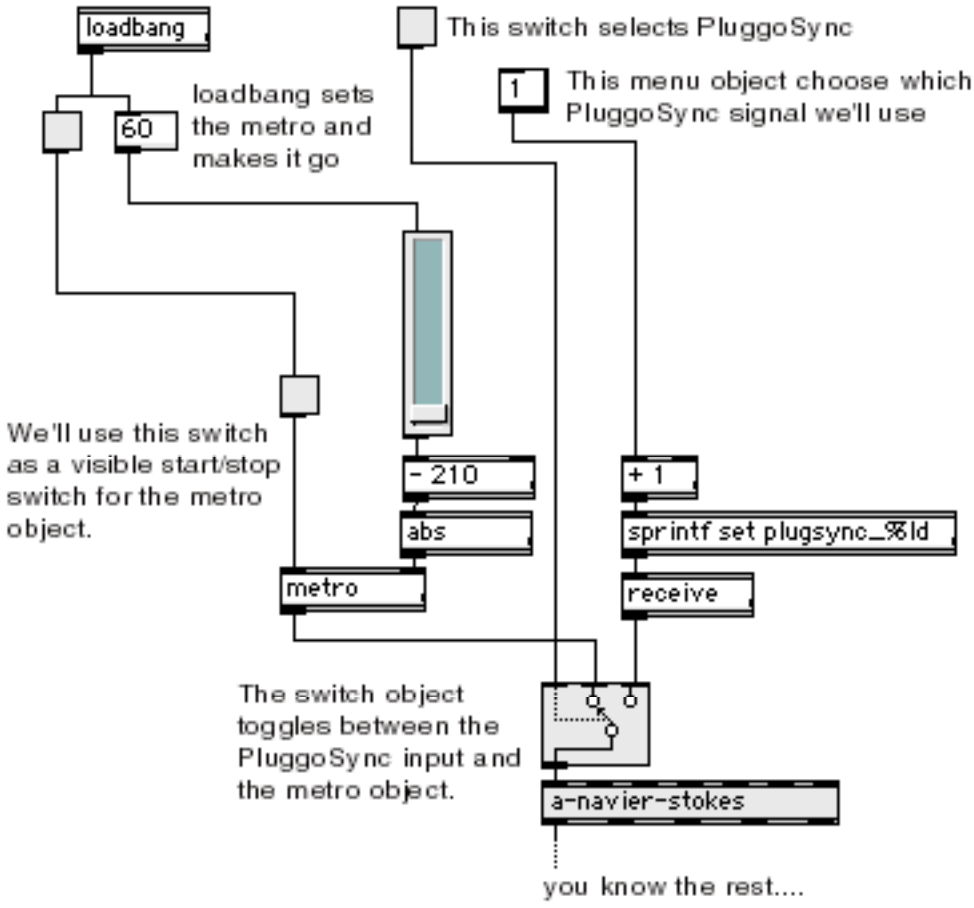


That would work, but a cooler and smarter way to do it involves using the Max **sprintf** object and the **receive** object's "set" message together. It takes the output of my menu box, and then constructs a message to the **receive** object that says set plugsync_1, so that my **receive** object is getting that particular plugsync message. Look ma, only one input!



Finally, I need to set things up to let me toggle between the metronome and the PluggoSync input. There are other ways to do this, but I'll just use a switch object.

Here's what my new *PluggoSync* input and control routing scheme looks like now:



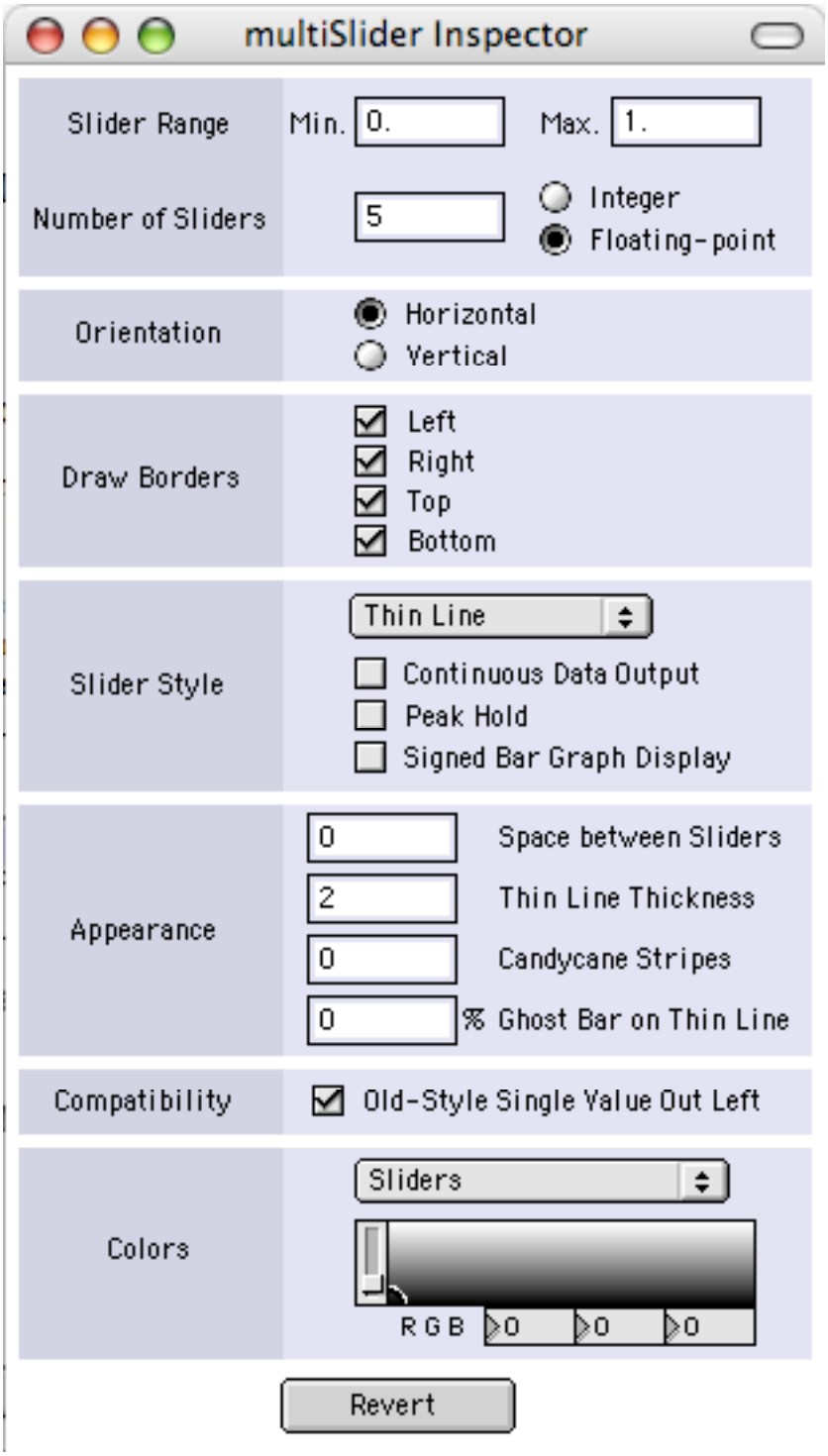
This is presented in a simpler form that you'll find in the actual KnaveStories patch, of course. The clever reader will also notice that I added a second switch box between the **loadbang** object and the **metro** object. I'll use this bang to act as an on/off switch – placing it so that it's on my front panel and visible, naturally.

At this point, I resumed what I'd done when I first made my interface panel – moving and rearranging the interfaces switches and sliders in the upper left-hand corner of my patch. In fact, adding the new stuff meant that I had to change my plugconfig file to make a larger front panel space – that was merely a case of changing the line `#C setsize 450 144;` with some larger horizontal and vertical values. I'll omit the interface design tweak loop details here. If you're really curious, you can open the plugconfig file in the KnaveStories_patch file and see for yourself.

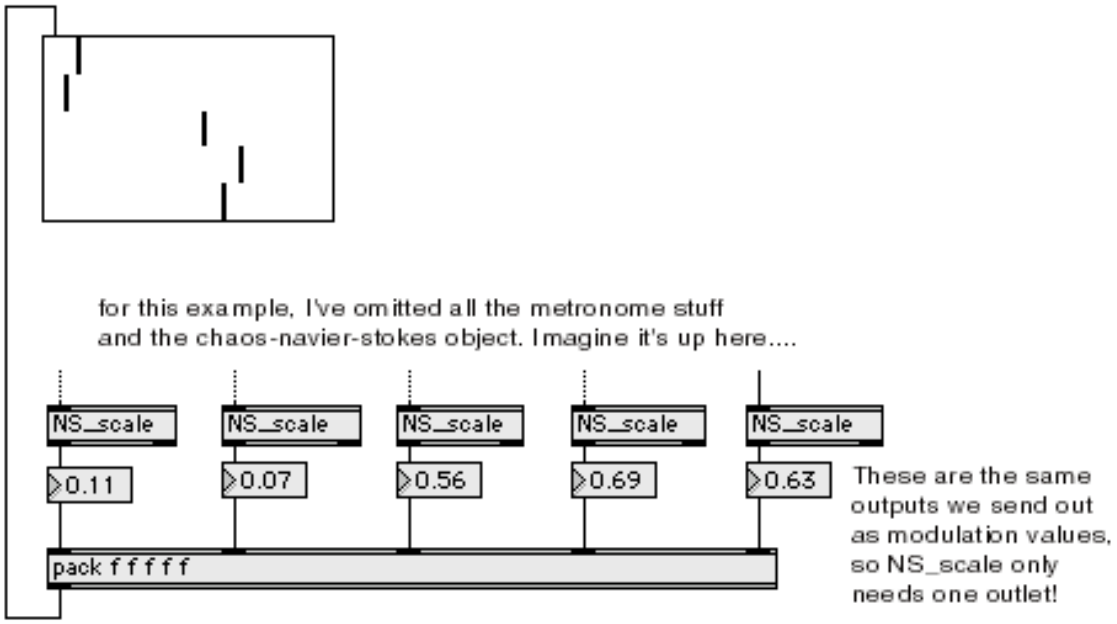
Adding A New Visual Display

This particular improvement to my patch couldn't have been much simpler. I just chose a **multislider** object from the Max menu and dropped it onto my patch.

I used the Info command (cloverleaf-I) to bring up the multislider's configuration menu, and set up my display object to replace my five hsliders with a single object that horizontally displayed 5 outputs as floating point values in the range 0.0 – 1.0 (Now I get it - that's why everyone uses the **multislider** object for data display:



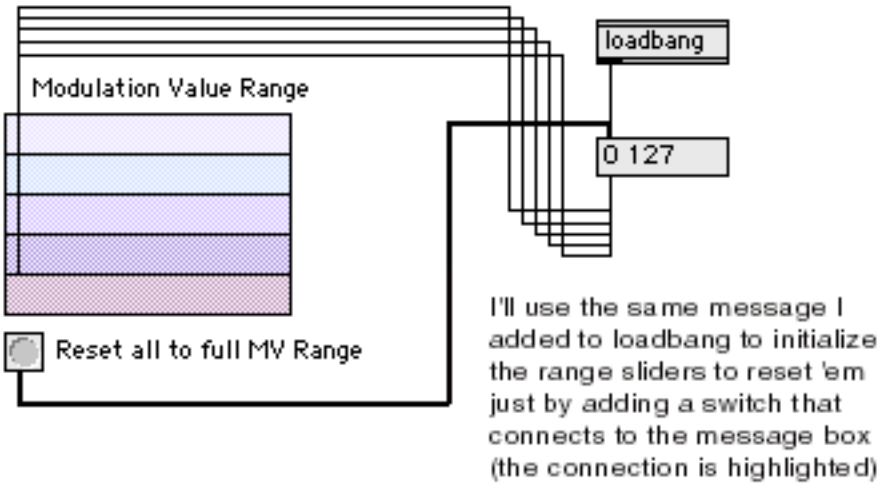
My new **multislider** now takes a string of five floating point values, and then displays them. So all I have to do is to add a **pack** object to the floating point outputs of my NS_scale object and route that to the **multislider** – sort of like this:



Home, Home on the Range

While I'm using no parameter presets whatsoever, a couple of hours of fiddling with my Version 1 patcher suggested that it'd be nice to have a switch that reset the range of all five of my range sliders to their original range.

Since I'm already using the **loadbang** object to set that range at startup, all I really need to do is to add another switch that uses the same message:



The astute reader will also notice that I've skootched the range sliders together. That's one of those aesthetic tweaky choices. Now that I've got my cool **multislider** object, it just seemed like the right thing to do. Your mileage may vary.

A Panel is Worth a Thousand Words

Before I send my plug-in out into the world, I think I'd like to add one of those snappy info panels like jhno's plug-ins have. It'd also be a good chance to credit Richard Dudas for his original long-ago work with the ChaosCollection that Andre used to make the OSX version (history matters), while I'm at it.

I wasn't sure how big my info panel picture should be, so I took a guess by measuring the size of the place my info panel would be on my monitor screen with a ruler (don't laugh – I really did this), and then figured that it'd be pretty close to that in size.

I fired up my copy of Photoshop, got out this sequence of fortune-telling pictures I took in Tokyo, and made an info panel, which I saved as a PICT file called KnaveStories_Info.pict:

The Pluggo Developer's Guide couldn't have been clearer about how I should add this info panel to my plug-in. I opened my Max patch file and double-clicked the **plugconfig** object to bring up the plugconfig script. I added the reference to the file by changing the line

```
#C noinfo;
```

(which says that there is no special info panel) with a new line that told *Plugmaker* to add my new info panel PICT file:

```
#C infopict KnaveStories_Info.pict;
```

I closed the plugconfig file text window and clicked Save to store my changes.

But I also needed to include that new picture as a part of my patch. The Pluggo Developer's Guide also describes this procedure – I need to save my patch as a collective file. I took a brief break and read through Tutorial P2. It was all there.

I used the Build Collective / Application / Plug-In.... command from Max's file menu. Just as I did before, I clicked on the Include File… button in the collective save dialog and then selected the NS_scale file. But this time, I repeated that same procedure and included my new PICT file.

Just as before, I clicked Build and the Save window appeared. I chose VST plug-in from the Format pop-up menu, saved it with my snappy new anagram name, and it was done.

My sequencer is launched, the KnaveStories plug-in is inserted in my console, and a look at the info panel tells me that all is well with the world. It's done (at least for now).

Coffee and Cointreau After Dinner – Some Final Thoughts

There you have it. That's how I turned into a plug-in developer, armed with only a burning vision, a passable grasp of Max/MSP, a tutorial, and some patches to plunder. If you look in the materials that accompany this document, you'll find copies of both my Version_1 and KnaveStories plug-ins, which you should feel free to mess around with and enjoy to your heart's content. Since I've also given you the Max patches themselves, you're free to root around in them, too (a quick look should convince you of the logic of showing more conceptual examples, I think).

This was a story about how I did my first plug-in, and then made my second one in an attempt to improve it – the only closure you're likely to get from me here is for that part of the story. Somehow, adding that silly little info panel to plug-in two made it seem like a real thing that I'd made – writing "Vincent" or "Greggus facit" on the lower right-hand corner.

So I wondered what to do next. As you can imagine, the answer was "Write more plug-ins, silly. Where else do you uh…want to go today?"

But there was another still, small voice:

> *Remember how you went on about plug-in writers in the Old Dispensation only writing that small number of plug-ins that'd make 'em lots of money and how you thought that left you out in the cold? What if there's someone else out there who might think it'd be cool to have a modulating plug-in whose outputs are both seemingly random but also have some sense of correlation? What about them?*
>
> *And while I've actually got your attention, remember that you occasionally find it daunting that so many tutorials imply that you're being guided by a superior intelligence whose knowledge you can only dream of instead of looking over some guy's shoulder? What if you just swallowed your pride and let other folks see your work, warts and all?*
>
> *And one more thing; how long has it been since you called your mom?….*

I'll delete the rest of my still, small voice's soliloquy to protect my privacy and pride. But the little voice had a point. So that's why you've got these plug-ins and this document and all this other stuff in your hand.

At one point, I was corresponding with, uh…a fellow plug-in designer. jhno. You may know his work, perhaps. I was asking one of those silly big questions about what kinds of cultural paradigms Max/MSP or Pluggo might represent, in addition to merely being a way to work. jhno said something that I keep going back to, and he says it's okay to quote him here:

> okay, i have some chocolate and alcohol and caffeine swilling around inside me - birthday gourmet with my friend dani at absinthe. so i feel the gift of tongues.
>
> pluggo is a manifestation of the evolutionary, creative urge to transcend the constraints of established systems in order to open new paths into the chaos of the unknown, where we might find beauty and insight. in pluggo we see the human desire to connect, to bring disparate individuals and communities together by drawing their interest toward each other. this act is a raising of consciousness, or increase in awareness.
>
> in practical terms: whether or not pluggo users end up making their own plug-ins via msp, there is something worthwhile in their realization that they can. whether or not they use the esoteric machinations of slice-n-dice or warpoon in their music, there is an artistic beauty in seeing that such techniques exist. this perception expands your notion of what is possible. the fact that you can use all these tools in meaningful ways integrates art and tool.
>
> pluggo, as a collection of meta-art, is an artifact unto itself - but at the same time, pluggo as a system allows new ways to reach into other tools that already exist. here i am referring to vst hosting, plugmod, mas support -

```
architectural features that allow previously disparate
systems to work together. in the best case, the utility of
this interconnection will help people in general realize
the value of cooperation, a principle that resonates with
software systems as much as it does in human interaction.

cooperation, at its most abstract, is humanity's saving
grace, that delivers us from the isolated nihilism of the
selfish gene. the mathematics of genetics show that it
works, just as the mathematics of the capitalist
marketplace have shown that cooperation works in business
and software. as in biology, there will always be predators
and cheaters - it is not an ideal, unlimited altruism.
therein lies the value of intelligence, the development of
cognition and memory - both biological and corporate - to
more finely discern the character of the environment and
its denizens.

so, to sum up: pluggo is a love chord.
```

I definitely couldn't have said that better.

Things I Could Do Next

Of course, there are other things I could do besides sharing my stuff with the world, too. If you'll recall, one of the assumptions that fell by the wayside as I started actually making my own plug-in was this idea that my plug-in sprang fully written and fully formed, like Pallas Athena from the head of Zeus. I hope I've laid that to rest. When Pluggo is a list of things someone else gives us, we tend to think of them as a fixed set (e.g. the Oblique Strategies) instead of something that we add to and whose additions are themselves evolving ideas or advice. So I'll do my part to lay that to rest by being explicit – Here 's a list of the next things I could do:

1. In thehelp patch for the **a-navier-stokes** object, he graphically rendered the outputs of the equation by showing you the cross-products of the variables in the equation. My pluggo object works only with the individual variables themselves. Wouldn't it be cool to take the output of the NS_scale object and add a switch which toggled between the **a-navier-stokes** objects' variables and their cross products. The resulting patterns are very different (I'll let you set that up and take a look at what I mean).

2. The **a-navier-stokes** object I used in this plug-in runs with a default set of initial values. Changing the initial values for equations like the Navier-Stokes will often produce different effects: The range of the outputs change radically, the rate at which the output settles into a more or less regular pattern, and so on. So far, I haven't added any presets to my plug-in. I could sit down and come up with other interesting initial sets of variables, and use those for my plug-in presets. It'd also be a good opportunity for me to work through the tutorial stuff and do something I haven't done before.

3. I could let a bunch of people who're much better Max/MSP programmers see exactly what I've done – warts and all – and then ask for their advice. Sure, it might be embarrassing to reveal to the general populace what potential howlers I may have committed. Of course, the plug-in does work. The feedback I got would probably be a great opportunity for me to develop some new good habits, and maybe pick up some programming tricks. If I get any, it'd not only make my plug-in more efficient, but I'd to probably learn something, too.

4. You might be looking at the code examples I put up and saying, "Yikes! That's the dumbest way to do this that I ever saw." Or "I cannot live without lots of **send** objects and tons of subpatches." At any rate, I'm sure that there are more efficient ways to do things than what I've chosen. Great. Email me with your better solution – I'll fix up and improve the patch, and maybe learn something, and everyone else will get a better patch. Go for it.

5. I've now done all the hard work for working with a chaos object. Richard Dudas wrote a whole collection of other chaos objects that Mr. Sier also ported, so I could take what I already have and do some replacing and cutting and pasting and make a whole bunch of other chaotic modulators based on other objects.

6. Then there's that idea I had about making a patch out of my favorite technique for creating huge walls of undulating oatmeal – two reverbs cranked all the way up with some filtering between 'em and at the output. Hmmm…wonder what kind of reverb objects are out there?

```
<SFX: The sounds of a bunch of annoyed Starbucks
employees making lots of noise in the hope that we'll
see they're trying to close up. Distant sirens and
assorted urban mayhem noises, and an old Hijokaidan
recording playing on the store's sound system. Is it
warbly? Who can tell?>
```

So there you have it – literally. You've got your own copy of KnaveStories. I hope that this has been instructive, but my real hope is that some of you will go poking around in the patches I've given you and give me some feedback and advice. You can reach me via email at gregory@cycling74.com. I hope this little diary encourages you to take heart, and to see that you don't have to be perfect to become an audio plug-in developer. The journey of a thousand miles begins with a single step.

Start walkin'—at the rate I'm sauntering, you'll have plenty of time to catch up.

Gregory Taylor
Madison, WI, September 1999 – January 2006