Demystifying Max/MSP

A guide for musicians approaching programming for the first time

by
Paul Schuette

Table of Contents

The World of Computer Programming

Max Pre-Basics

Lesson 1: Playing a Sound File

Lesson 2: Decision Making 101

Lesson 3: Simple Sequencing

Lesson 4: Live Sound

Lesson 5: A Serving of Amplification Issues with a Smattering of Variable Handling on the Side

Lesson 6: Introduction to User Control

Lesson 7: Effects

Lesson 8: Synthesis

Lesson 9: More on Flow Control

Lesson 10: Randomness

Lesson 11: Cleaning up Your Patches

Lesson 12: Alternate Controlers

Lesson 13: Jitter

Lesson 14: Building Applications

Object Index

Table of Contents

The World of Computer Programming

Max Pre-Basics

Lesson 1: Playing a Sound File

Lesson 2: Decision Making 101

Lesson 3: Simple Sequencing

Lesson 4: Live Sound

Lesson 5: A Serving of Amplification Issues with a Smattering of Variable Handling on the Side

Lesson 6: Introduction to User Control

Lesson 7: Effects

Lesson 8: Synthesis

Lesson 9: More on Flow Control

Lesson 10: Randomness

Lesson 11: Cleaning up Your Patches

Lesson 12: Alternate Controlers

Lesson 13: Jitter

Lesson 14: Building Applications

Object Index

Demystifying Max/MSP

Objective

My early experiences with Max/MSP were during my undergraduate years of college. Professors, who were primarily music teachers with no programming background, walked us through a few of the tutorials that come with the program. These teachers intuitively saw that Max could have extraordinary benefits, yet their abilities only allowed them to provide us with an introduction at best. More importantly, they were unable to present the program to us in a way that would enable us to use it to our own creative ends. This being the case, the purpose of this book is to provide musicians, and, specifically, music teachers with a resource that allows them to present the abilities of Max/MSP to their students despite their own level of programming experience.

The line which these teachers offered to those of us who were thirsty to know more, was "Read the tutorials." This is a line that the Max community still stands by, however, this can only be an affective means of learning Max if you already possess some basic programming skills. Programming languages are languages, so image the following scenario: You are trying to learn a spoken language, like French, but you have no idea what nouns, verbs, articles or even letters are. The point is once you know the basic components of any spoken language it is possible to learn any other language by "reading the tutorials." Programming languages are no exception. For this reason, this book assumes that the reader has no idea what the basic components of a programming language are. This book is intended to approach Max/MSP as your first programming language, for it will lay a foundation and build upon it step by step, never using any vocabulary without introducing it first.

This is not to say that the resources available within and around Max are totally useless. Many of the tutorials are written very clearly, and at times, this book will also reference help patches and other resources that are already included in the Max environment. The problem is that these resources are of varying type and purpose and are found in so many different places that it is hard to for someone just starting out to understand how they fit together. So while this resource does not aim to entirely replace all of this information, it offers the beginner an insight into how all of the available resources can be used in concert so that the road to creative freedom with Max/MSP can be travelled with confidence.

Why Use Max/MSP?

I have heard is said many times that, "Max can do anything you want it to," when it comes to manipulating and handling sound. While this is undeniably true, it is also true that many of the things which you could do in Max are much more easily and efficiently achieved with other more polished pieces of software. So what are the unique abilities that Max has that other programs do not offer? While there are multiple answers to this question, for my purposes the answer is resoundingly clear: It is Max's ability to handle

input data. This ability, to handle live incoming data and have it affect output, gives you a platform from which to create performance patches that are adjustable in real time, interactive performance situations for musicians, interactive installations, cross continental web based performance patches, and the list goes on and on. If it is unclear what this all means or how it all fits together, then this book is what you are looking for. After completing all of the lessons, you will be able to use Max/MSP in a creative and flexible way that will open the door for you to all of its myriad possibilities.

Purchasing

Max is currently in its 5th version and is available from cycling'74. There are student prices as well as an option for a 9 month trial.

Cycling'74 makes a version of the program called Max/MSP Runtime which is available for free download. This version of the program only allows you to run patches that have already been written. It is useful when you want to operate a patch on another machine which does not have the full version of Max installed.

In addition to these options, there is an alternate freeware version of the program called PureData, or PD for short. It is developed and maintained by the creators of Max with the intent of being a leaner and faster version of the prepackaged program. The differences between the two are almost entirely aesthetic with the exception of a few identical objects that have different names. These discrepancies are pointed out within each lesson as they arise making it possible to use this book in conjunction with either Max or PD.

The World of Computer Programming

Max's Place on the Map

Max, at first glance, may not appear to be a proper "programming language." When most of us hear this term, our minds immediately turn to scripted pages of computer vomit with more punctuation than text. While these types of programming languages do exist, they are really not as scary as they may seem, and, moreover, they are not the only type of programming languages around. For an analogy, it might be helpful to return to the notion of spoken languages. Think of low level procedural languages as being equivalent to Latin and Ancient Greek: they form the basis of most modern languages even though very few people actually still speak them. Max is based on these low level languages, but exists in a more highly evolved hybrid state like the English language. Technically, Max is an object based programming language with a graphical user interface (GUI). This means that the lines of code which define each object do actually exist, but that you as the programmer never have to see them. Instead, the objects serve as a graphical representation of these lines of code making them much more intuitive to handle. In this way, Max blurs the line between what a programming language is and what an application is. In truth, it truly is a mix of the two. Some might put it in the class of a programming environment, but it might be best described as an application for programming.

Programming Culture

The fact that hybrid application/languages even exist brings up an important point about how these languages have evolved. Again, the ties to the development of spoken languages seems apt. English is a Creole language, which means that it is an amalgamation of many different languages all rolled into one. It has been so well established that its roots no longer seem of any importance. For example, it would seem utterly absurd for someone to learn English by beginning with Latin and tracing the evolution of the entire language from the bottom up. This same type of hierarchy exists around computer languages. Nobody has written a piece of code using 1's and 0's for decades, except to prove their geekhood to the two people who care. Max is a very high level language that has evolved from many other languages.

The larger issue here is that outright stealing is an accepted part of programming culture. Pieces of code are borrowed, share and ripped from other people, and this is what you are expected to do. The best programmers working right now never start with an entirely blank screen. The problem is that people who are new to writing computer code sometimes feel a tinge of morality running up their spine when they "borrow" a piece of code for the first time. But imagine the comparison: nobody feels like they are stealing anything from the people who spoke Latin when they are teaching their kids English.

Admittedly, there is a different issue at work here with first time programmers, and that is the desire to know about the levels of abstraction that lie beneath the programming language they are currently dealing with. These are right and good things to be curious

about, for no matter what level of technology you are operating at you should always be aware of the levels of abstraction that surround it. However, when you stumble upon the first piece of code that somebody else has written that you find to be absolute genius take it and sleep well at night, for you are paying that person homage.

Pseudocode

It is often helpful to begin writing a computer program by describing what you want to have happen in plain English. This is absolutely the best advice I can think of to offer someone who is writing a computer program for the first time. For example, simply state the function of the program like, "I want to record this sound and play it back at this point in time and at this speed." This is referred to as a pseudocode and is a good habit to get into early and will speed up your learning curve ten-fold. Writing pseudocode allows you to begin writing your program by tackling the portions that you know how to achieve first and then filling in the gaps. Going from something to something else is always easier than starting with nothing. . . or something like that.

How Computer's Think

The nice thing about computers is also the thing that is most frustrating about them: they do exactly what you tell them to do. I've never felt the difference between man and machine more intensely than when I think I'm telling my computer exactly what I want it to do but its doing something completely different. The difference is that I care and start spitting fire at the screen, but the computer just sits there still doing the thing I don't want it to do.

The point here, if I may beat an already long dead horse, is that writing a piece of computer code is similar to the process of writing anything else: it is never going to come out perfectly the first time. In the programming world, this process of revision is known as debugging and it is an essential yet tedious part of writing any piece of computer code. Nobody likes it, but everybody has to do it. My advice is to remember that the computer looks at the world through a soda straw: it sees one thing at a time and in the order in which you tell it to do so. So just remember, when your computer isn't doing what you want it to, it's because you told it to do it that way, and when you feel yourself getting angry, walk away before you do more harm than good.

Max Pre-Basics

Let's begin with a quick vocabulary lesson:

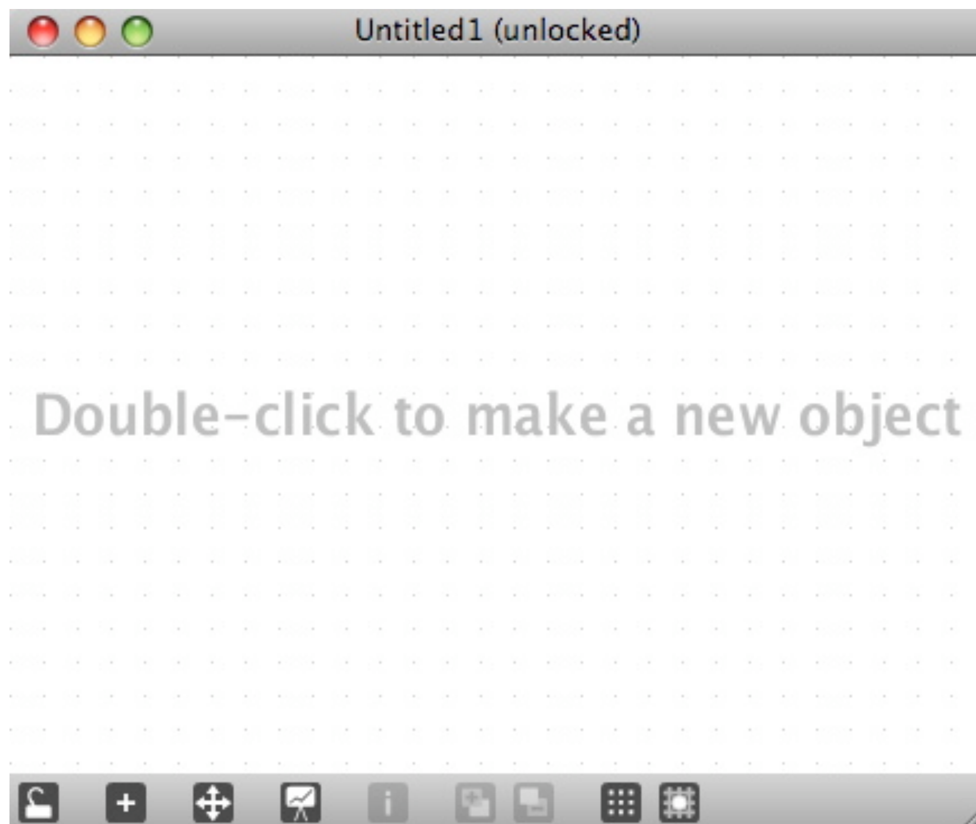
Object: objects are the primary building block of the Max/MSP programming language (more detail on these in a minute).

Patcher: when people refer to a patcher, they are usually talking about the most basic type of window that the application uses. Its like saying, "Open a blank document," when talking about a word processing program.

Patch: the code that you will write using Max is commonly referred to as your "patch." We will get to the root of this terminology later, right now its important to understand the difference between the patcher window and your patch.

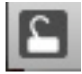
The **patcher** window is the primary Max window and is the platform upon which all of your code will be written.

Here is a **patcher**:



From left to right the buttons across the bottom serve the following functions:

The patcher window has two modes: locked and unlocked. Currently, as you can see the patcher is unlocked. When the patcher is unlocked, changes can be made to the patch, i.e. objects can be added and removed. When the patcher is locked, the patch will run, i.e. buttons can be pushed, numerical values can be entered and sound can be heard.



This button allows you to add objects and everything else available to your patch. The same menu can be viewed by double clicking within the patcher window as the background so grandly implies. The very first item in the top left of the pop up window is for a generic, yet to be defined, object.

When your patches get bigger, "zoom" offers a way to easily navigate to different regions of your patch.

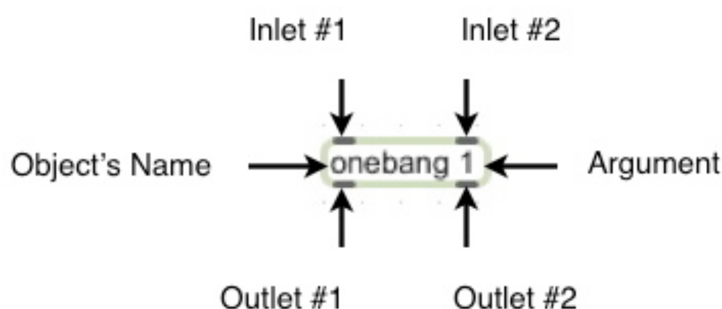
This button allows you to switch the patch in and out of "Presentation Mode": a new feature of Max5, see Chapter 11 - Cleaning Up Patches for more on this.

Skip the next three which are not bolded.

These last two give you a choice about whether or not you want new objects to conform to a grid and whether or not you want to see said grid as you are programming. The choice is yours.

The Object

As I already mentioned, objects are the building block of any Max/MSP program, and they look like this:



Take time to note how we will refer to the various aspects of objects.

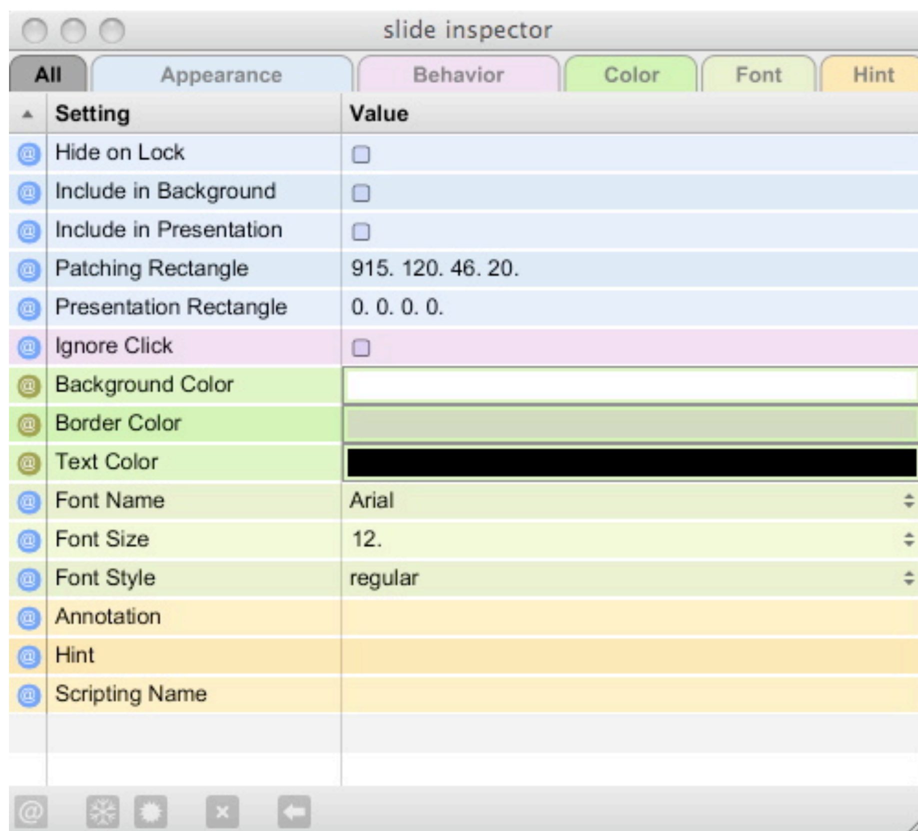
DSP Window/Max Window

The DSP (digital signal processing) window is found under the Options / DSP status drop down menu at the top of the screen. The DSP window tells you whether Max is ready to make sound or not and how that sound will be made. The most important fields are the ones at the top that show a list of all the available input and output drivers your computer has at its facility. If your patch is not making any sound when you think it should, look here first. Another thing to note here is the CPU% box which tells you how much of your computers processor is being used by your patch - with time you'll be surprised at how CPU intensive your patches can become. This is also where you set up MIDI devices.

The Max window is found in the Window drop down menu. The Max window is where you will find any errors in your patches that Max is trained to catch. You can also print things to the Max window, but more on this later, just know where it is and how to find it for now.

Inspector

Also new to Max5 is the Inspector window. Each object has its own Inspector window that you can access by highlighting that object and clicking on the Inspector icon at the bottom of the patcher window or by clicking on the "i" icon that appears when you scroll over the left side of any object. As you can see, different aspects of an objects function and appearance can be manipulated from within that objects Inspector.



Referencing Objects

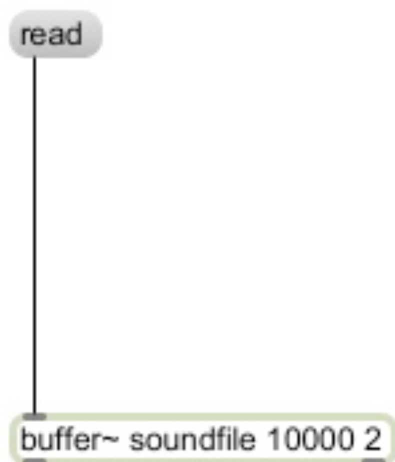
As you delve into the Lessons, you will quickly notice that not all objects have the same boxy little object-like appearance for some have rather elaborate interfaces. I will reference all objects by their names, and this is all you will need to be able to find them. No matter how fancy an object may appear it can still be implemented by typing its name into an object box. Try **nslider** for a quick example.

Order of Operations: Right to Left

Max functions from right to left. This means that if you are trying to get two things to happen simultaneously the thing on the right will happen first. In most circumstances, this is not an issue in large part due to the increased speed of central processors. However, as you become more advanced, this issue may rear its ugly head, so for now just store this fact away in the back of your brain for later use.

Lesson 1: Playing a Sound File

In this lesson, you will learn how to load a sound file from anywhere on your computer into Max and you will be familiarized with one of the many options Max offers for playback of that file.



This is a "message box". Messages are different than objects. Messages are sent to objects and instruct the receiving object to perform a task when clicked upon. Different objects take different messages. In this instance, the "read" message is telling the **buffer~** to look for a sound file to store.

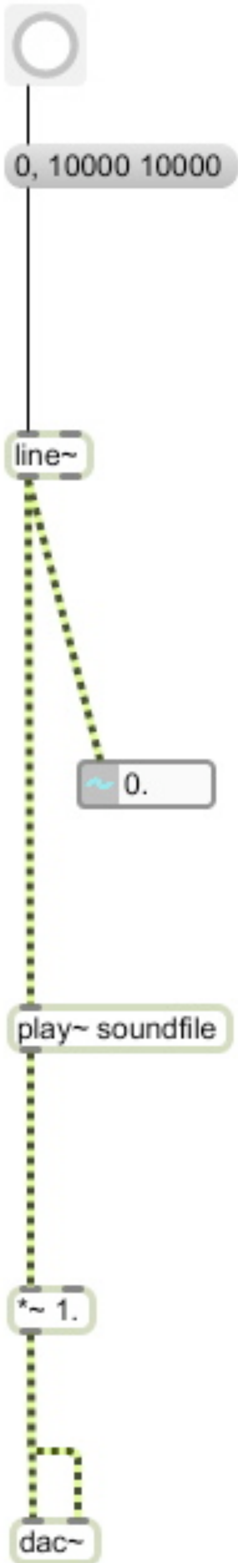
To create a message, you can type **message** into an object box, or they are also easy to find in the pop-up menu which lets you add new items to your patches.

The black line is referred to as a "patch cord" as a throw back to the days of analog audio. Patch cords are used to make all connections in the Max environment. Hence the terms patcher and patch.

buffer~ is an object which stores audio data so that it can be called upon at anytime. Most objects take a series of arguments: additional information that further defines its parameters. In the case of buffer~, the arguments give the buffer a name, set its length in milliseconds and sets the number of channels. Once you have read a file onto the buffer, double click on it to make sure that it's there. This **buffer~** is called “soundfile” and is set to store 10 seconds of audio on 2 channels.

Tip #1: While most objects do not require that all of their arguments be included, buffer~ seems to be happiest when all of its arguments are in place. Generally speaking this is a good/safe mode of operation and will save some inevitable debugging time.

comments are a ubiquitous part of programming culture. Essentially, a comment is exactly what its name implies: a little blurb of text that describes what is happening with a piece of computer code - beyond this they serve no real function. Like writing pseudocode, commenting your patches is a good habit to get into and I strongly encourage you to comment your patches as you begin to work your way through these early lessons. Comments can be added to your patches using either of the techniques which were just outlined for adding message boxes.



This is a **button**. Any time it is clicked it sends out a "bang" which is used to tell other objects to “start”, “go” or “make it happen”.

This is a message for the **line~** object. Most messages that an object understands will be demonstrated in the object help patch which you can view by option+clicking on that object in edit mode. However, a complete list of messages for an object can be found by clicking on the link in the header of the help patch which says "open object reference".

line~ takes a message with a specific structure. The message gives line a starting and ending number and then tells it how much time to take to count between the two. So the message in this example could be parsed as follows: Go from 0 to 10000 milliseconds in 10000 milliseconds. Take the time to experiment with other combinations. You could go from 12 to 9999 in 4876 ms: this is the beauty of Max - ultimate flexibility.

This is a **number~** box and will show the flow of information coming from the **line~** object.

Tip #2: Objects that include a ~ in their name handle audio data, other objects simply handle raw numerical data. Some objects like "number" come in both flavors.

play~ plays back audio that is stored on a **buffer~**. It requires an argument that specifies the name of a buffer to read from: in this case "soundfile".

Patch cords that are checkered indicate that these objects are passing an audio stream of data.

*~ is an amplifier. It takes an audio signal (~) and multiplies it (*). Values you give it should range from 0.0 to 1.0 as anything over 1.0 will cause your audio to clip.

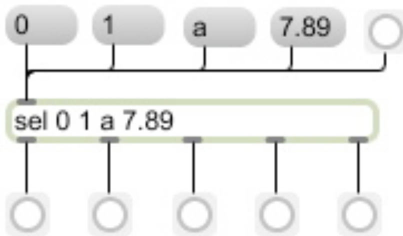
Tip #3: To clean up a patch cord highlight it and type Apple + y.

dac~ stands for digital to audio converter and does exactly that. It converts digital audio to analog audio so that it can be heard over your computer speakers. Its default is for 2 channels of audio, hence the 2 inlets, however it can be set for more channels than there are speakers in a Best Buy show room.

* Double click on **dac~** to see the DSP window.

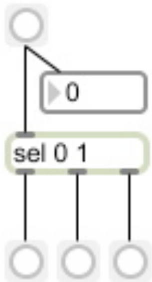
Lesson 2: Decision Making 101

In this lesson, you will be introduced to the most basic tools Max has for making decisions about streams of data.



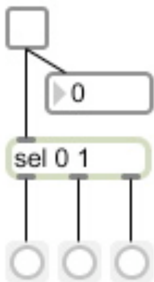
select, or "**sel**" as it's referred to by those in the know, is an extremely useful object. It takes any type of input and routes it to a specific outlet. The outlet furthest to the right outputs incoming data that does not match any of **select**s' arguments.

Tip #1: Note that buttons are also useful for indicating when something has happened.

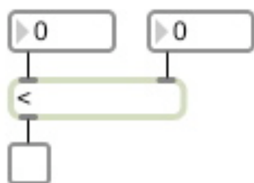


Jargon Lesson: Computer programs, Max included, differentiate between whole numbers, or integers (referred to as "ints") and numbers with decimal points known as floating point numbers (referred to as "floats").

A bang, from a **button**, does not have any numerical value and is therefore rejected by select.

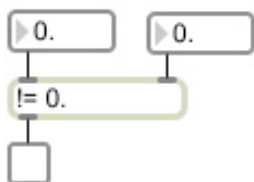


The **toggle**, when compared to the **button**, is the other equally important type of trigger in Max. The biggest difference is that a **toggle** <u>does</u> output a numerical value: 1 when it is on or true and 0 when it is off or false. *Computer languages still reference good ole 1's and 0's all of the time. It is extremely important to correlate the concept of 1 being on or true and 0 being off or false. This will come up time and time again.



<, less than, compares two numbers to see if something is true, and, therefore, provides a great example of how the toggle functions.

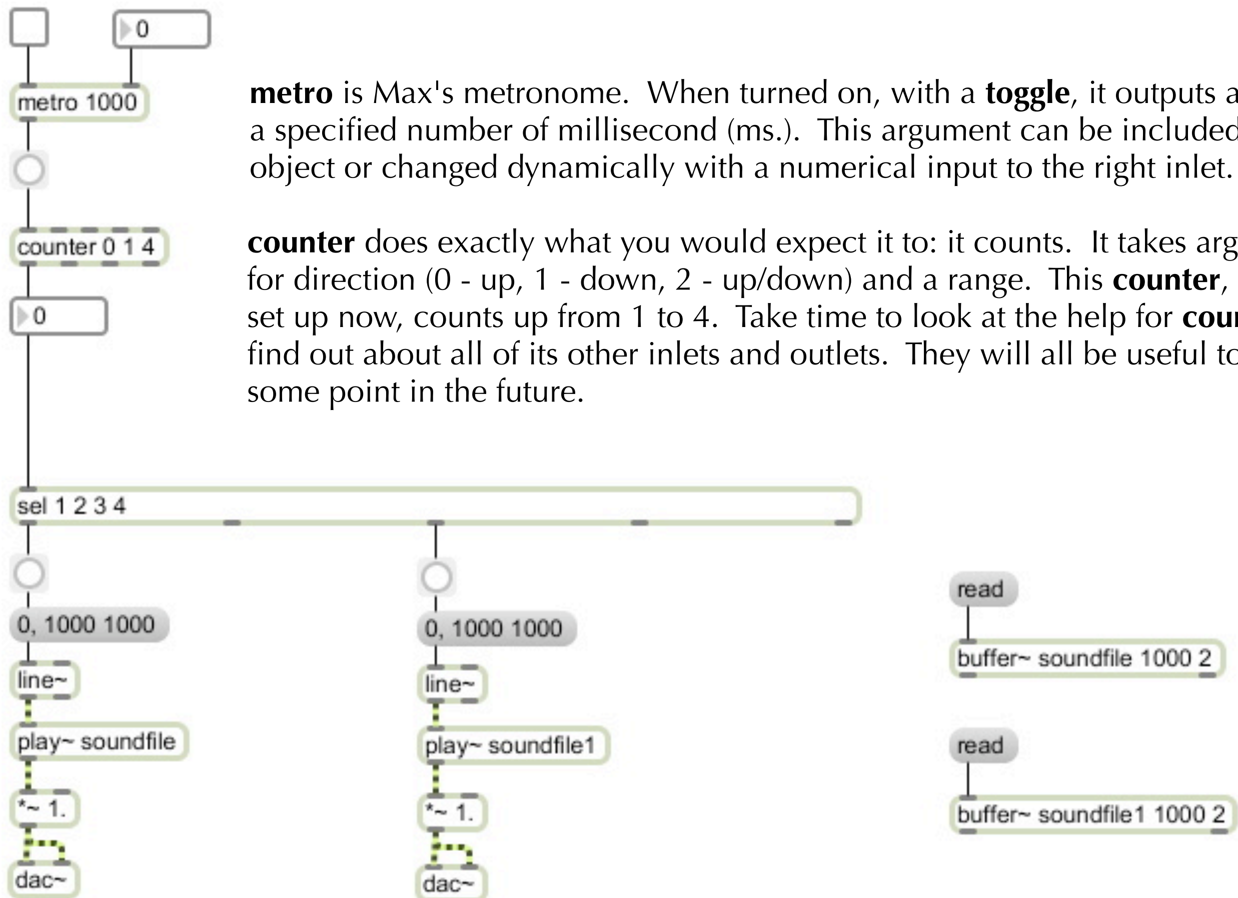
Tip #2: The left inlet of most objects must be "banged" before the object will produce anything. Place a button between the right number box and the left inlet of < and see what happens.



All objects in Max which handle numerical data can take the argument "0." which allows them to function with floats including **==**, equal to and **!=**, not equal to . Notice that the floating point number box is also required. Be sure you can follow the logic going on here. Can you predict which position the **toggle** will be in when the number ares the same.

Lesson 3: Simple Sequencing

In this lesson, you will be introduced to **metro** and **counter**. Using these, objects in combination with others that we have already covered, will allow you to create a patch which coordinates the playback of sound files over time.



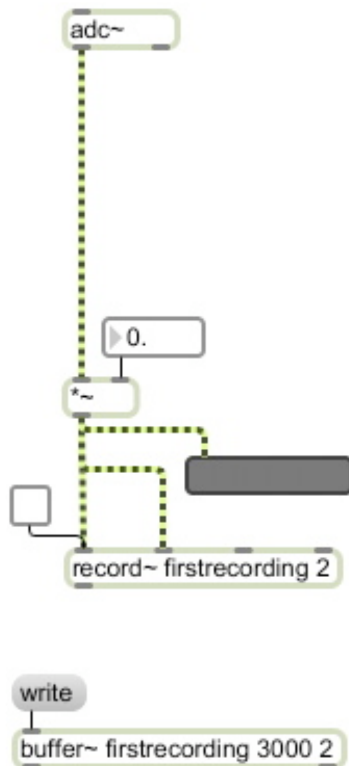
metro is Max's metronome. When turned on, with a **toggle**, it outputs a bang at a specified number of millisecond (ms.). This argument can be included in the object or changed dynamically with a numerical input to the right inlet.

counter does exactly what you would expect it to: it counts. It takes arguments for direction (0 - up, 1 - down, 2 - up/down) and a range. This **counter**, as it's set up now, counts up from 1 to 4. Take time to look at the help for **counter** to find out about all of its other inlets and outlets. They will all be useful to you at some point in the future.

By attaching a **sel** to your **metro**/**counter** combination, you will have achieved your first successful sequencing tool with Max/MSP. Everything that has been added on should look familiar from previous lessons. Don't forget to "**buffer~**" a couple of sound files. Have some fun adjusting the parameters of either **metro** or the messages that **line~** is receiving and see if you can get the sounds to overlap.

Lesson 4: Live Sound

So far all the sound that we have dealt with has been prerecorded. In this lesson, we will examine the basics of bringing live sound into your Max patches.



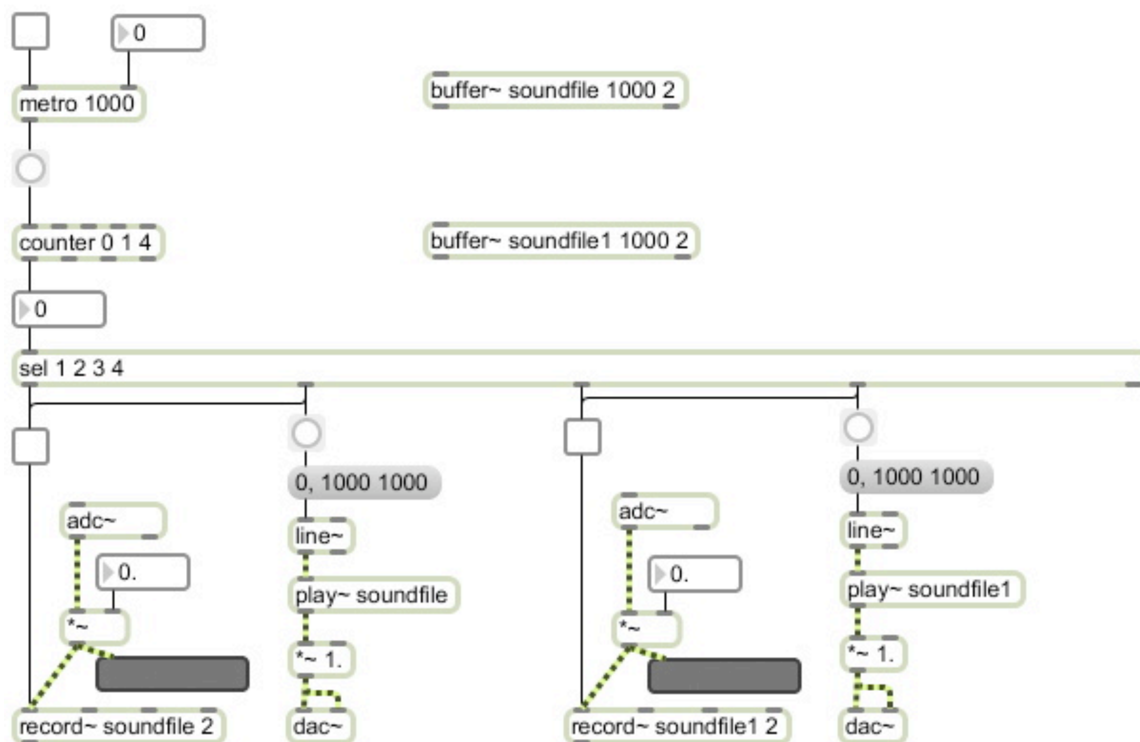
adc~ is an analog to digital converter: which means it is essentially a microphone. It takes real world sound and digitizes it so that your computer can understand it. It will listen to the input source you have selected in the DSP status menu. Like **dac~** it takes an argument for the number of channels you want and defaults to 2 channels, however, unless you are using a stereo mic, you will only need the first outlet.

***~** is functioning as a preamplifier. Inserting this helps to insure that you're recording at a good level and without clipping.

Adding a **meter**, the gray box, helps you to visualize how much sound is coming in.

record~ is in the same family of objects as **play~** because it requires a **buffer~** to record onto which must share the same name. It also takes an argument for the number of channels. Recording is started and stopped with a **toggle** to the leftmost inlet.

A write message to **buffer~** allows you to save an audio file to your hard drive.

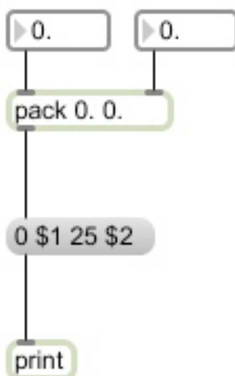


Here's the sequencer we built in the last lesson incorporating live sound:

Recording is started on beats 1 and 3 and turned off on beats 2 and 4. On beats 2 and 4, the recordings that were made on beats 1 and 3, respectively, are heard. Clap or whistle on beats 1 and 3 and you will quickly understand what I'm talking about.

Lesson 5: A Serving of Amplification Issues with a Smattering of Variable Handling on the Side

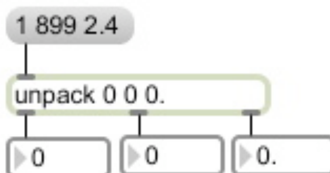
In the sequencer patch that we just built, you might have noticed some issues with unwanted pops and clicks in the audio. These occur whenever you try to start digital audio at full amplitude. Solving this problem in Max is a major headache for experienced people who do this stuff all the time. In this lesson, I am going to present one option for solving this problem. We are going to make a ramp that goes from 0.0 to 1.0 and back to 0.0 in exactly the amount of time it takes to play any sound. This ramp will ultimately be plugged into an amplifier (*~), removing those pops and clicks from the edges of your sounds. To achieve this we first need to tackle one major issue: variables and how to handle them.



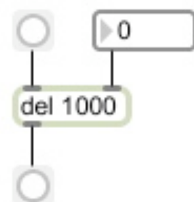
pack takes data from separate sources and makes it into a list. It creates a new inlet for every number you want to have included in the list. Right now it is set to receive 2 floats.

Variables in Max are indicated with a "$". We call them variables because they have no defined value. This message has 2 variables: $1 and $2. "Packing" the variables into one list allows us to place the variables wherever we want to within a single message.

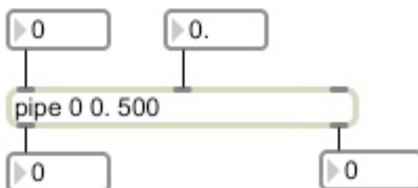
print allows data to be displayed on the Max window which you can find in the Window drop down menu or by clicking Apple+m.



As you might have suspected, lists of numbers can also be "**unpack**ed".

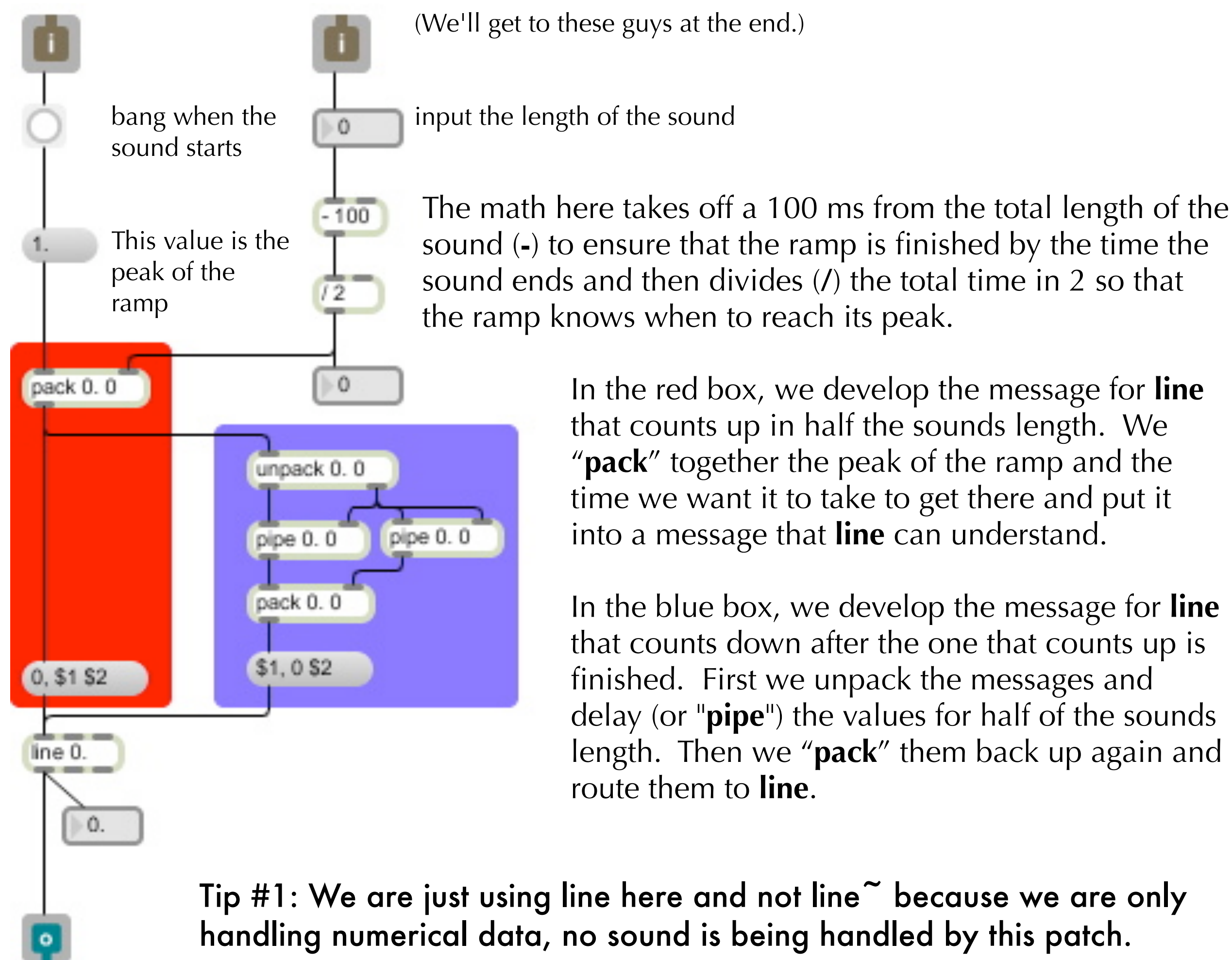


Now is as good a time as any to introduce one of the most useful objects in the entire language. **del** (short for **delay**, similar nomenclature to **sel**) delays a bang for a set number of milliseconds - which can be changed with a new number in the right inlet similarly to **metro**. See the help patches for **tapin~** and **tapout~** for the audio signal version of **del**.



pipe is a very special kind of delay, for it delays a number, instead of just a bang, for a set period of time. This one is set to delay an int and a float for 500 ms. (the rightmost inlet can alter the delay time).

Now that we've covered all the objects we'll need, let's look at how to actually put something together that will eliminate all of those pops and clicks. As I already said, these occur when you start a sound at full amplitude. So to eliminate the problem we'll construct a ramp that can be applied to the value of an amplifier so that a sound never starts or stops at full amplitude.



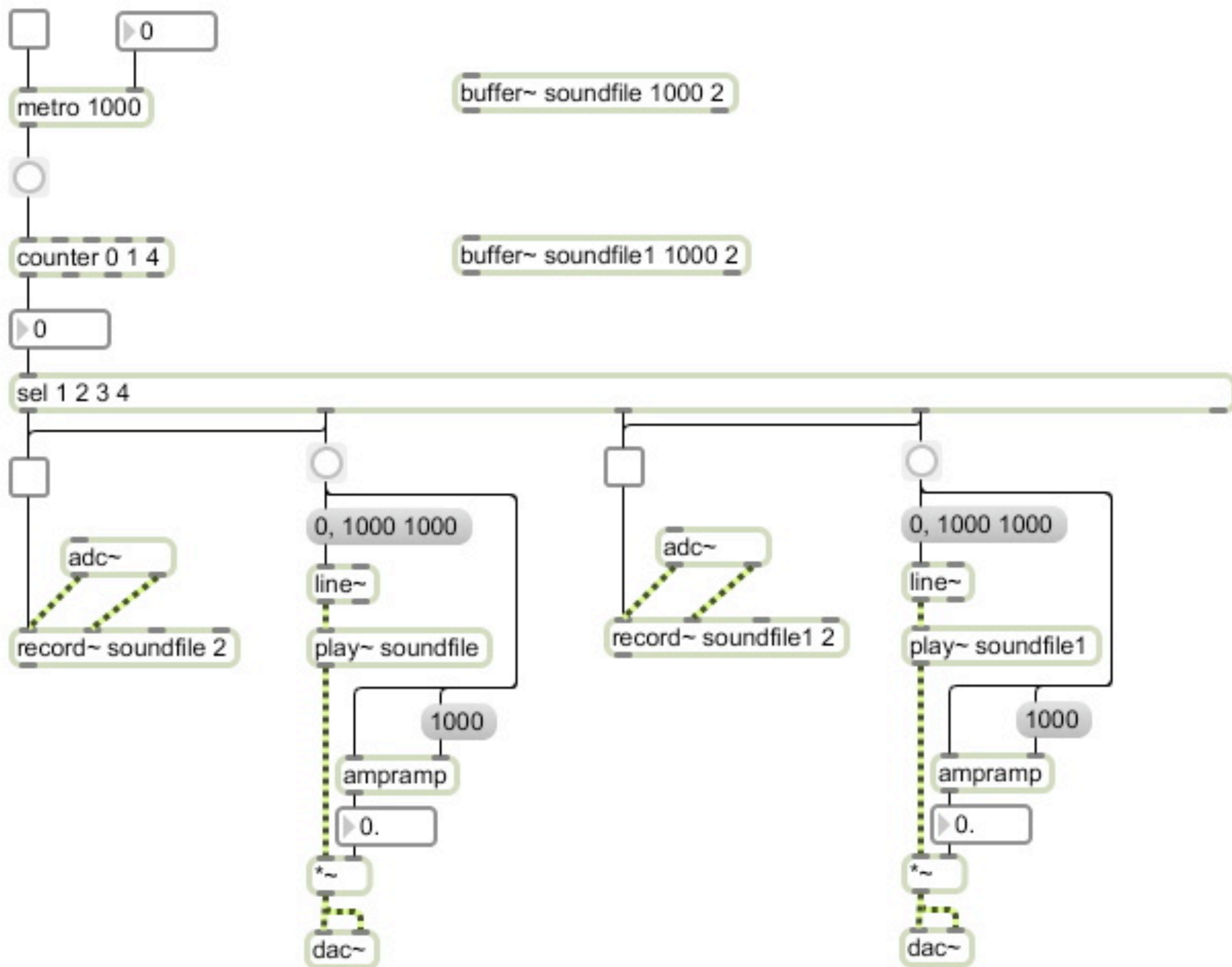
Take the time to understand every patch cords function in this patch. Notice which arguments are floats and which are ints. This will help you follow the data throughout the patch. Make sure you are able to describe the flow of this patch using plain English. Use number boxes in your construction is you're not sure what value is getting to the different inlets - this is a key part of troubleshooting any patch and a good lesson to learn early. If you can get this one, you're well on your way. A lot of key concepts are at work here.

If you're feeling like you have a really good handle on this patch, try to simplify it so that the ramp reachs 1. in 100 ms. and goes down 100 ms. before the sound ends. This can be done with fewer objects!

Understandably, you might still be confused about how to actually implement this thing. The explanation has in part to do with the little green and blue boxes that say **i** and **o**. These are inlets and outlets, respectively, just like you'd find on any other object. By adding these, we can save this patch in Max's file structure so that it can be called upon just like any other preexisting object. Here's how to create your first, very own, Max object:

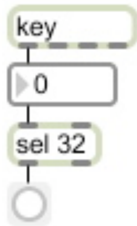
1. Save the patch with a catchy object-like name that describes its function, like "ampramp."
2. Go into Applications/Max5/Cycling '74 and create a folder for your objects, like "My Objects."
3. Save the object into the newly created folder. Restart Max, and now when you go to add a new object, "ampramp" should now be in the list.

Now you can create the following click free sequencer:

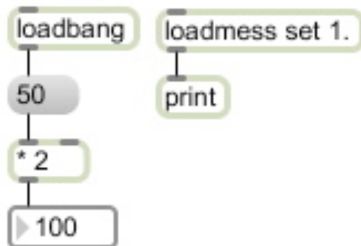


Lesson 6: Introduction to User Control

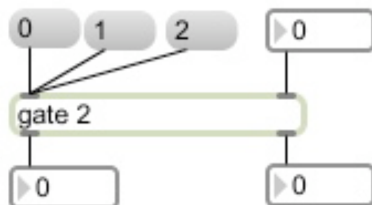
Let start by going over some new objects:



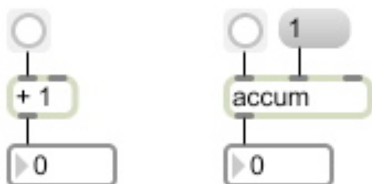
key is the most basic type of user control after pointing and clicking with the mouse. It allows you to utilize the keyboard in controlling your patches. Right now it's set up to work with the space bar. The numbers it uses to reference the keys are known as ASCII - they are one of the original standards still around in the computer world. Search online for an ASCII table of numbers to discover how to reference the other keys, or simply watch what comes out in the number box when different keys are pressed. For other simple forms of user control, check out **hi**.



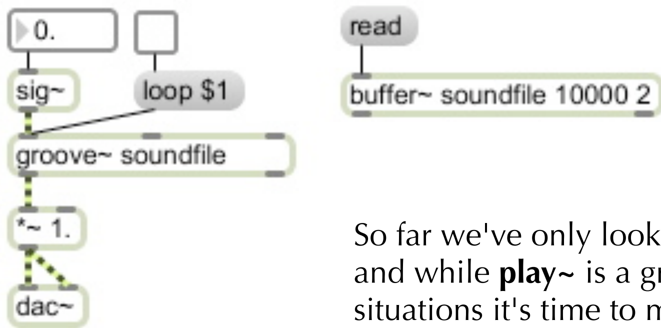
loadbang and **loadmess** are very closely related objects. They allow you to set values and messages automatically whenever a patch is opened so that you don't have to load them by hand every time you open the patch. **loadbang** just sends out a bang, as you might have expected, and **loadmess** sends out whatever message you have typed in as an argument. They also work when you double click on them.



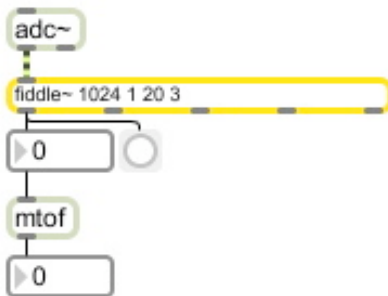
gate is an extremely useful little object which helps to control the flow of information throughout a patch. **gate** takes data in its right inlet and sends it out via which ever outlet is specified at its left inlet. In this case, there are two outlets (you can add more if you need them) and 0 always closes the "gate". gates with only one outlet are often controlled with a toggle. In PD, a **gate** is know as a **spigot**.



When you repeatedly bang a math object, it doesn't continue to add to the value. Have you noticed this problem already? The answer to this problem is **accum** - short for accumulator. It does just what a simple **+** object won't: it continues to add to the total. Notice that it doesn't output anything until you bang the leftmost inlet. The right most inlet is for multiplication.



So far we've only looked at **play~** as a way to produce sound with Max, and while **play~** is a great object to know and extremely useful in many situations it's time to move on to bigger and better things. **groove~**, also uses a **buffer~**, but allows for looped playback. Looping is turned on and off via a message. Make sure you understand how the variable is being handled here (remember 1 is on and 0 is off). Don't concern yourself too much with understanding the **sig~** right now, just know that in tandem with **groove~** it allows you to control the speed of playback. 1 is normal speed, .5 is half speed, 2 is twice as fast, etc.



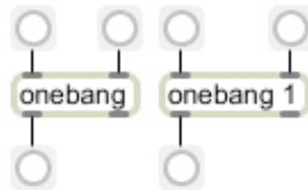
fiddle~ is not an object which comes with your original installation of Max. This is an important part of the Max culture: anybody can create new objects and share them with the community online. Some are similar to the "ampramp" and are built out of existing objects, others are written in different languages entirely. The best resource for external objects, as they're known, is www.maxobjects.com. To use **fiddle~**, you'll have to get online and download it and put it into Max's file path just like we did last lesson with the "ampramp". There will also be a help patch included which you can put into one of the help folders.

Tip #1: Some people have entire libraries of external objects which you can download. A few that I suggest are "ftm", "artificial tango" and "tap.tools" (this last one costs money but is worth every penny).

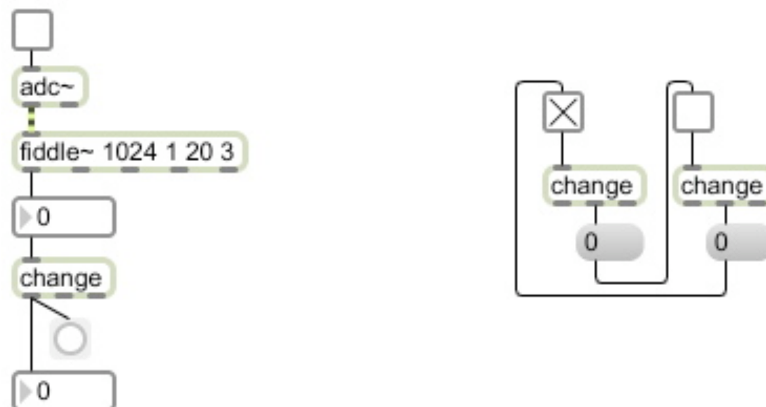
fiddle~ is a pitch tracker. It takes in audio data and tells you what midi value the pitch is. It requires four arguments which are all explained in the **fiddle~** help patch, but if you're like me, you'll never need to fully understand them and can just copy and paste them each time you call on **fiddle~**. Turn on your DSP and see what happens. Using **fiddle~** in performance can be done, but with the understanding that it is not 100% acturate. Precautions must be taken with the way you set up your performance so that, for example, microphones are only picking up the instruments you want to track and are not getting sound coming from loudspeakers as well. It's probably most reliable with electric instruments that can produce a signal which can be directly routed into your computer. Precautions must also be taken with the way you handle the data coming from **fiddle~** to ensure that what you want to happen happens despite **fiddle~**'s inaccuracies.

mtof converts midi values to actual frequency.

fiddle~ produces what we will refer to as an irregular or continuous data stream. Sing one pitch and fluctuate the volume or wiggle the pitch around slightly and notice that the number doesn't change but that multiple bangs are produced. This means that the number is also being sent out that many times even though you don't see it changing. This happens because **fiddle~** is listening to incoming sounds at a very fast rate which is a good thing because it lessens the chances of it missing something. However, this is a problem when we want just one thing to happen when a performer plays a middle C, for example. **onebang** is an object which helps to handle these types of data streams. It does just what it says: it only lets one bang go through the left inlet until a bang to reset it is received in the right inlet. Giving it the argument 1 sets it to allow through the first bang it gets.

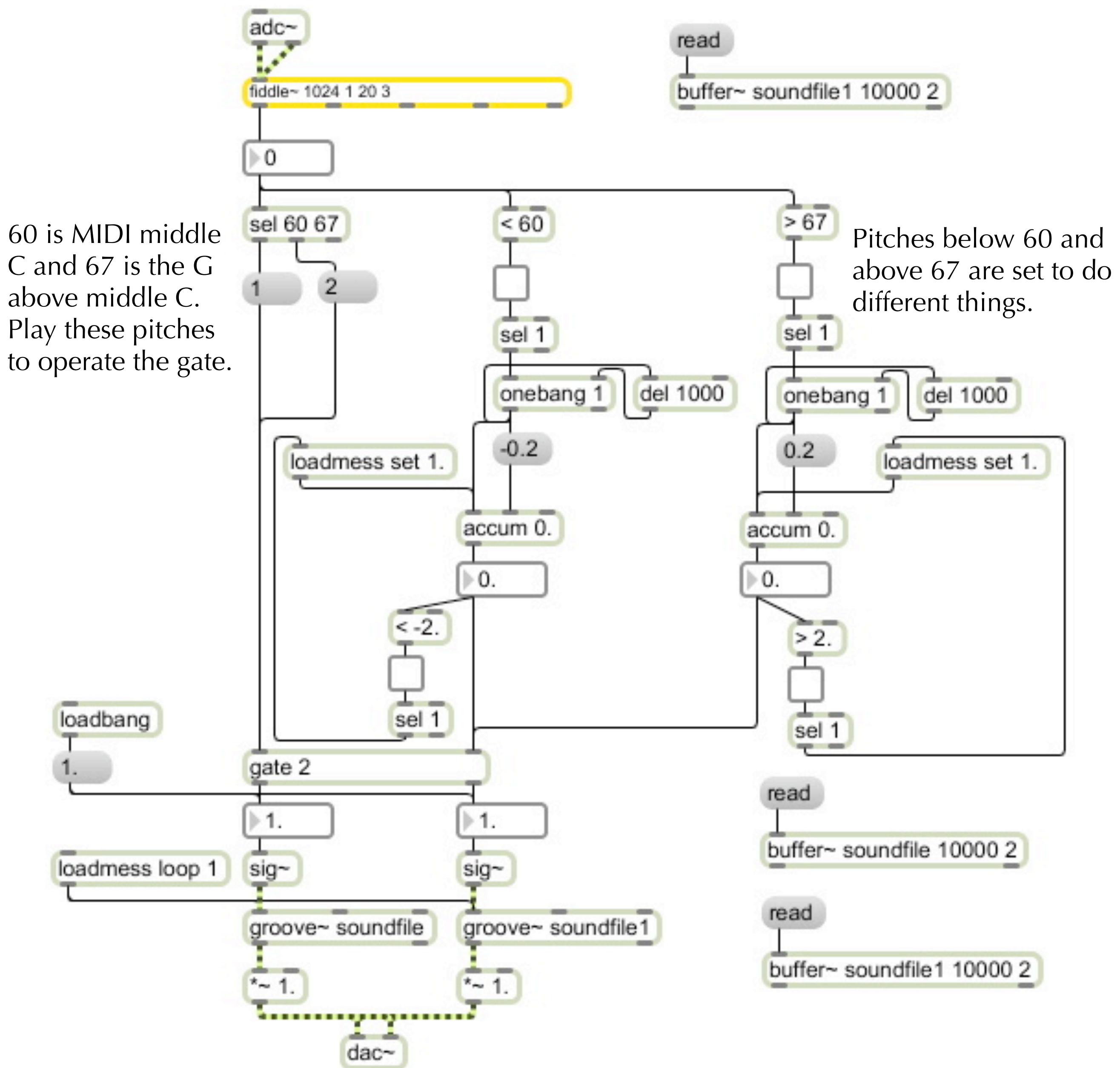


Another extremely useful object which aides in controlling irregular data streams is **change**. **change** only outputs a number when the number it receives is different than the last one it received. In other words, it waits for a "**change**" before it does anything. When used with **fiddle~**, change functions like an intelligent version of **onebang**. Notice how many bangs are registered when the patch on the left is functioning. The other two outlets of **change** report changes from zero to non-zero numbers and vice versa, respectively. These outlets can be helpful when you want one **toggle**, for example, to switch off when another one is activated. Take the time to understand the logical flow of the patch on the right, it is a brainteaser.



Tip #2: For away to track rhythm explore the possibilities of using **timer** in combination with **fiddle~**. Look for the "clever" hint in **timer**'s help patch and discover what a "cooked" pitch is coming from **fiddle~**.

Here it is finally all together in one patch. Ultimately, this patch is kind of silly, but it demonstrates a lot of the thought processes that are involved with handling a data stream and getting that data stream to perform a task. Take your time to understand everything by trying to describe what's going on with plain English and remember that the computer sees the world through a soda straw. If you have a keyboard close by, it's probably your best bet for testing this patch. Be sure to **buffer~** a couple sound files before you try to listen to anything.



By using **del** in tandem with **onebang**, you can set a sample time for looking at the data that is much slower than the one **fiddle~** uses.

Lesson 7: Effects

In terms of producing sound, so far we've only looked at ways to play back sound files that you processed and edited in another program or that you recorded live with Max and played back unaltered. Most effects that you find in a commercial sequencer can be achieved in Max. I'm going to demonstrate just a few of them, but more importantly, give you an idea of the variety of places where you will find Max effects.

read

buffer~ soundfile 10000 2

This is an object called **radiogroup**. Get into the inspector to change how many buttons there are.

sig~ loadmess loop 1

groove~ soundfile1

0

gate~ 4

gain center fq. Q

0. 0 0.

reson~

*~ 1.

dac~

reson~ is a pretty generic filter, but it does the job. It's an object that comes with Max. It takes arguments for gain, center frequency and Q (the tightness of the filter around the center frequency). Try attaching **line** objects to any or all of the parameters for sweeping effects.

This is an object called **slider**. Its default range is 0-127, but this can be altered in the inspector.

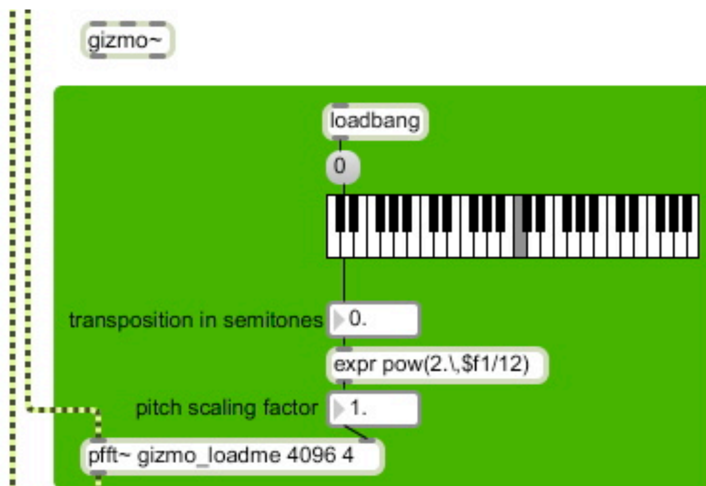
0

pan2

*~ 1. *~ 1.

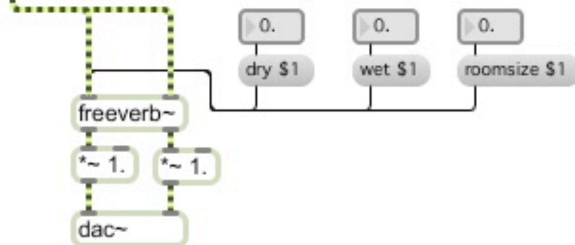
dac~

pan2 is a basic panner. Its second inlet takes values from -1 to 1; its third inlet takes values from 0-127 (which conveniently works with the default range of **slider**); and its last inlet allows you to control the fade time of the pan. This "object" and a lot of other cool effects can be found by taking the following file path: Max5/ Cycling '74/examples/spatialization/. There are lots of other really interesting patches in the examples folder that I highly recommend stealing from.



Another nice effect, included in Max, is **gizmo~**: a pitch shifter. In order to function properly, **gizmo~** needs to be in between a couple of other objects. To use **gizmo~** what I have always done is to open the help patch and copy and paste the following portion - which is in green. This is a perfect example of a time when it's the right thing to do to steal something. Use the keyboard (**kslider**) to change the pitch of your clip by a certain number of half steps.

No need to know exactly what's going on here, but here's the jist of it: The "0" message to the **kslider** essentially “zeroes” it on middle C, normally **kslider** would output MIDI values. **expr** allows you to write your own mathematical functions in Max. I know that's what it does, but I've never had an instance where I've needed to do math that's so complicated that normal math objects can't get the job done, but you might, so check out **expr** if you need it.



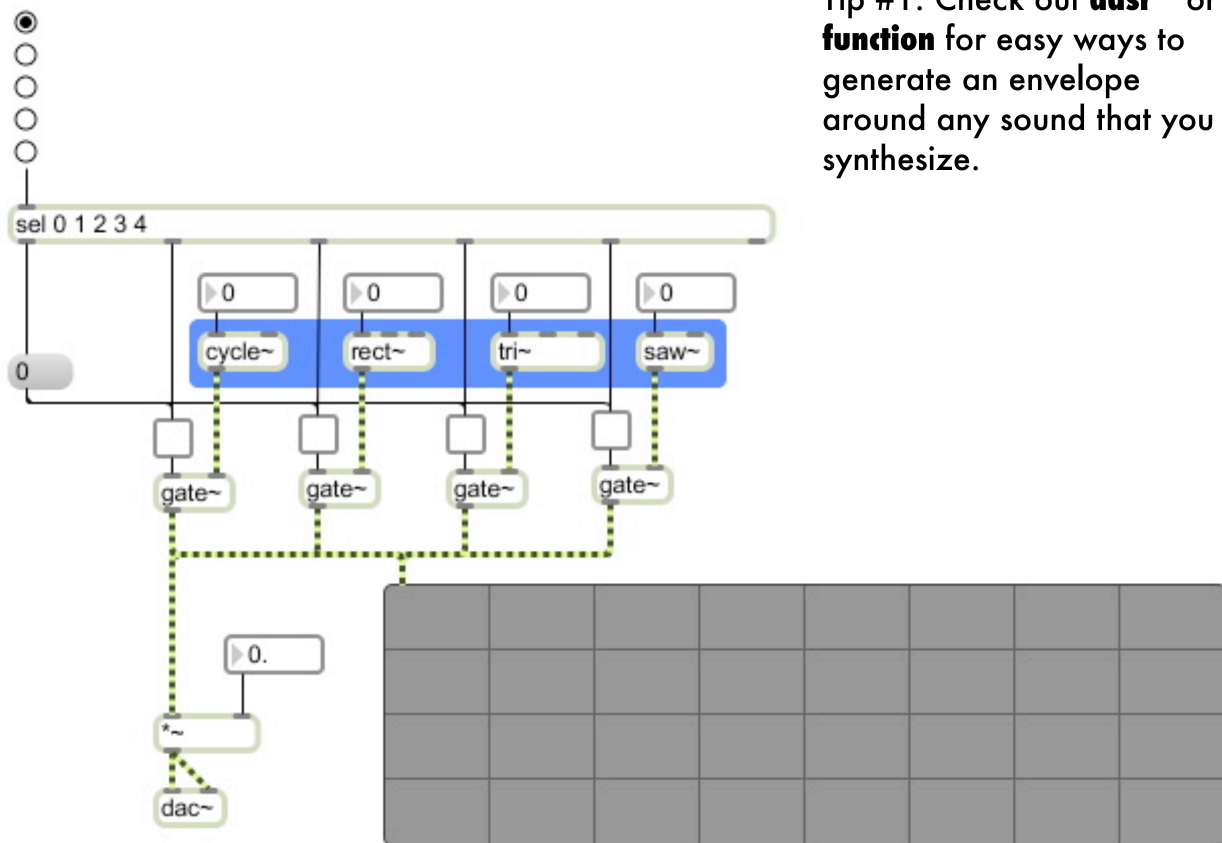
freeverb~ is a really basic, but good sounding reverb. It takes stereo input and produces stereo output. All of it's parameters are controlled with messages to its left inlet. I've included the three that I find the most useful, but you can find a full list in the **freeverb~** help patch. **freeverb~**, unlike the other three effects we've looked at, is not included anywhere within the application. It has to be downloaded from the Internet and installed just like **fiddle~**.

Lesson 8: Synthesis

If the MSP tutorials have one strong point, it's that they do a pretty good job of explaining how to use the program to achieve a couple different kinds of synthesis. The tutorial on additive synthesis is about the best one they have, so check it out before you work through this lesson. Having said that they completely ignore subtractive synthesis, so we'll be sure to take a look at that. Max can be an incredibly powerful tool for synthesis because it is so malleable, however, something new that we haven't really touched on is that Max can also be a powerful teaching tool. When it comes to explaining different wave forms, harmonics and other sonic phenomena, Max can help you visualize and understand what's actually going on within these otherwise abstract concepts in a very real way and more importantly it can save a lot of academic mumbo jumbo.

Waveform Generators / Teaching Tool

Tip #1: Check out **adsr~** or **function** for easy ways to generate an envelope around any sound that you synthesize.



The only thing new here are the waveform generators. **cycle~** produces a sine wave, cosine waves are also available with **cos~**. **rect~**, **tri~**, and **saw~** are all fairly self explanatory. If you're not familiar with this terminology, check out the **scope~** (basically an old school oscilloscope but not quite as heavy) and you'll be able to make a pretty good guess about what they do.

Subtractive Synthesis

This is an area we technically already touched on when we discussed filtering in the lesson on effects. Filtering and subtractive synthesis are really synonymous concepts. However, in the more tried and true tradition of subtractive synths, Max offers not one but two different noise generators which are great for filtering since they contain waveforms at all frequencies. **noise~** produces traditional white noise and **pink~** produces pink noise - non-technically speaking it's just what you might have guessed: a softer, more cuddly version of white noise. As soon as you hook these objects up, they are active. In other words you don't have to tell them to start working, so you're going to want to have plenty of ways to control the amplitude of their signal because they can get out of hand in a hurry.

We covered **mtof** when we first looked at **fiddle~**. **ftom** also exists as you might have suspected.

Remember the arguments for **reson~** are gain, center fq. and Q (do as the help patch suggests and try a range from 0-100 for Q).

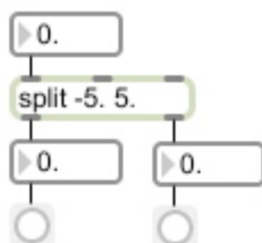
Be sure you're able to explain in plain English what's happening here. Can you predict what the sound will be like before you hear it?

The FM synthesis tutorial, which you might conceptualize as multiplication synthesis if you're just starting out, is also one that is definitely worth checking out.

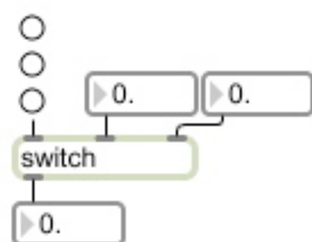
Also, for a really slick way to visually control your synthesis patches look at **techno~**.

Lesson 9: More on Flow Control

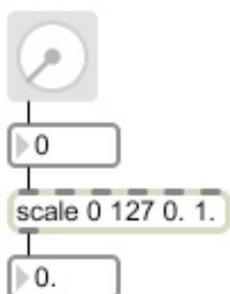
The flow control of your patch refers to the way in which data is handled and passed around to various objects. We began looking at the issues involved with flow control when we routed the pitches coming in from **fiddle~** onto various parameters of **groove~** back in Lesson 6. As I've mentioned previously, the ability to have incoming data effect and control how your audio functions is the primary ability that using Max offers you over all other audio software. In this lesson, we're simply going to look at a lot more objects that allow you to manipulate and handle data streams.



split is a Godsend when it comes to handling data streams. It takes arguments for a range of numbers (notice they can be negative and floats). Any input that falls within the range is sent out the left outlet and any input that does not fall within the range is sent out the right. Note also that any numerical output can easily be simplified to a bang if you don't actually need the number.



switch can be thought of as the opposite of a **gate**. While a **gate** takes in only one input and routes it to multiple outputs a **switch** takes multiple inputs and routes them to just one output.



dial functions similarly to **rslider** except it looks like a dial.

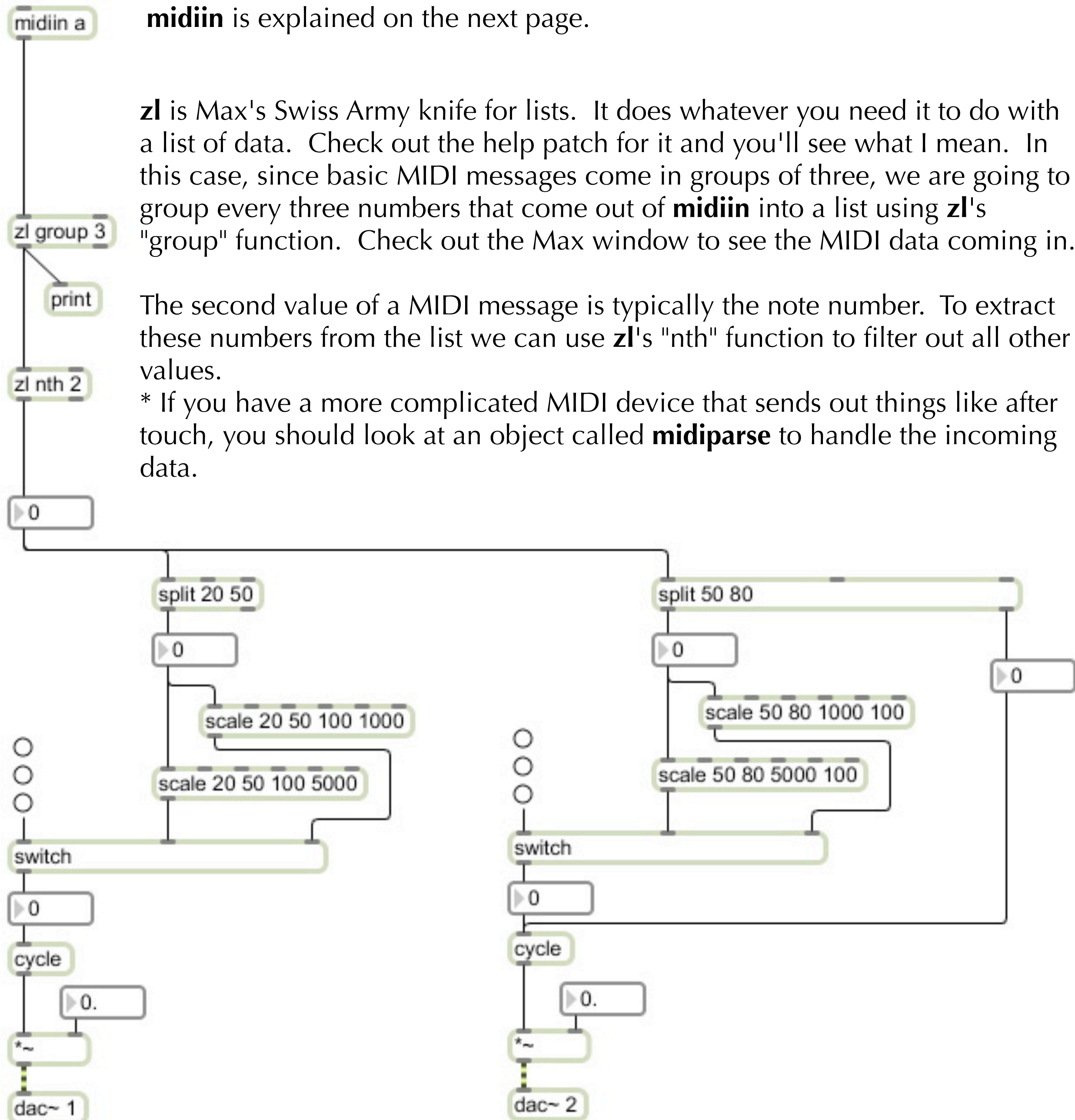
scale is an essential object when it comes to manipulating data streams. It takes two ranges of numbers for arguments. The first range is the input range and the second is the output range. In plain language you could say, “the object ‘scales’ the input range to the output range.”

bucket route router spray funnel bondo buddy

These are all objects that help to manage data and flow control. All of them are fairly similar but serve slightly different purposes, and more importantly they all have very helpful help patches that will be there to help you when their time comes to serve a role in one of your patches.

For now, let's look at a couple more commonly used objects and see if we can put some of the ones we just looked at together to achieve something interesting with a MIDI data flow.

* This example will work best with a MIDI keyboard. To use it with other MIDI devices some of the ranges may need to be adjusted.



midiin is explained on the next page.

zl is Max's Swiss Army knife for lists. It does whatever you need it to do with a list of data. Check out the help patch for it and you'll see what I mean. In this case, since basic MIDI messages come in groups of three, we are going to group every three numbers that come out of **midiin** into a list using **zl**'s "group" function. Check out the Max window to see the MIDI data coming in.

The second value of a MIDI message is typically the note number. To extract these numbers from the list we can use **zl**'s "nth" function to filter out all other values.

* If you have a more complicated MIDI device that sends out things like after touch, you should look at an object called **midiparse** to handle the incoming data.

Back in the days when Max was just Max and not Max/MSP, everything Max could do was in some way related to MIDI and all of those old Max objects are still around. Adding a MIDI controller to your setup is a popular way to enhance user control of your patches beyond the keyboard and mouse. Adjusting sliders and knobs is just more intuitive when it comes to handling sound. Usually hooked up to your computer via USB these days, **midiin** is the simplest way to bring raw midi data into your patches. **midiin** takes an argument for a port specification: this will usually be "a". Many people who own USB MIDI devices find it useful to make external objects that are digital mirror images of their devices physical layout so that they can drop their "device" into any of their patches and easily assign knobs and sliders to different functions.

Let's try to track the flow of the signal and describe the usefulness of each object as the data moves around this patch.

split is used here to section portions of the keyboards range to serve different purposes. Pitches from 20-50 effect the pitch of the sine tone which is heard on the left side of the stereo field and pitches from 50-80 effect the pitch of the sine tone which is heard on the right side. Pitches above 80 are matched exactly by the sine tone on the right.

scale takes the ranges produced by **split** and expands them. Notice that **scale** can invert ranges. i.e. The low value of the input range can be mapped to the high value of the output range as you can see and hear on the right side. Some especially astute readers right now might be questioning the usefulness of using two different instances of split. Why not just have one split with a range from 20-80 and send it to all of the different **scale**s and let **scale** do the work? The answer is that **scale** functions in a less than ideal way, for it will interpret output values for inputs that are not in the range it is given. This sounds confusing but give any of the **scale**s a value outside of their input range and you'll see what I mean. The lesson is to only allow a **scale** to receive numbers that are in its given input range.

The **switch**es let you determine which **scale** is affecting the frequency of **cycle**.

Could you have described this sequence of events on your own? As I've said once and will say again, knowing what you want to achieve and being able to express it in plain language is the first step to writing any piece of computer code.

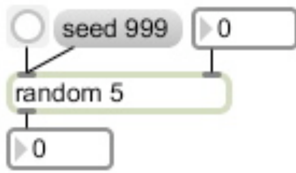
Lesson 10: Randomness

To some randomness has a trite and trivial connotation, but to others it is a deeply profound philosophical matter. There are actually multiple conferences held annually that address the issue of randomness attended by both mathematicians and actual philosophers. People get together and debate things like whether something can even be truly random. When you stop to think about this, it is a rather fascinating question because everything has to come from somewhere, but if you know the source of something, is it really purely random? Anyways, its interesting food for thought.

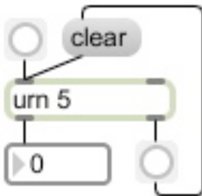
For our purposes, we will set these questions aside and look at others that are more pertinent: the first of which is interactivity. Interactive art has become extremely popular in the past couple of years. The word itself has been applied to so many different scenarios that is has become about as meaningless as the term postmodern. We have already looked at a couple ways in which it is possible for you to "interact" with your patches, but I will make the claim for the rest of this book that for something to be truly interactive there must be a reciprocal relationship between user and machine. Let's first look to human interactions for some correlations to what I'm talking about. Telling your computer what to do with the click of a mouse is equivalent to a drill sergeant commanding a cadet to give him 50 push-ups. Is this an interaction? Sure, but each side does not inform the other equally. Take, as another example, a conversation with a colleague about a current project where ideas are exchanged and where each person leaves the conversation with a new understanding of something, whatever it may be. Now this is a true and real interaction.

The first scenario I described when applied to artistic contexts produces art that is interaction as spectacle. It is art that says look at what I can do. The second scenario, when applied to artistic contexts, produces true interactions in which both parties involved (in our specific case the collaboration between you and your computer) benefit and are informed and changed through the process of collaboration. One of the techniques that can be employed to generate this type of interactive relationship is the use of randomness in your patches, for using randomness allows the computer to make a decision on its own which you can design to be within a specific range of outputs but ultimately will be left to the computer to determine. Then in response to the computers decision, your reaction can in turn be informed and thus the conversation begins.

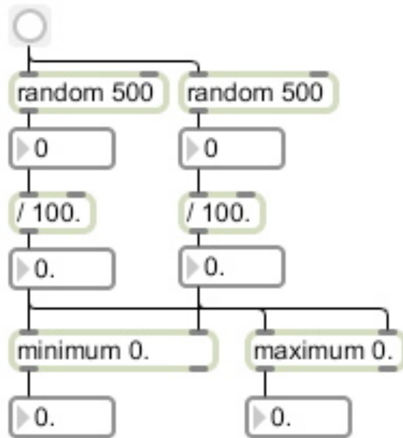
In an attempt to simplify this concept, you could conceive all of the patches we have built up until this point as being patches that are designed for performance and what we are attempting now is to build patches that perform. Let's begin by looking at the objects Max offers to generate random values and how Max gets these values.



The simplest object for producing random values is believe it or not **random**. It takes an argument for the range of numbers which you want to produce. Remember that computers start counting with 0, so an argument of 5 will give you values from 0-4. To overcome this it is often useful to put a "**+** 1" right after **random**. **random** generates values by looking at your computers internal clock. The seed message tells **random** to begin with a specific clock value. Allowing you to produce the same sequence of random numbers every time - which kind of defeats the purpose but can be useful if you happen to find a string of numbers you really like.

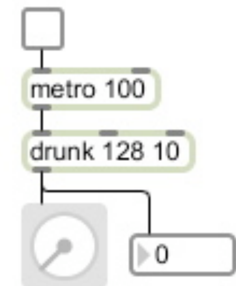


urn is a little more controlled version of **random**. It stands for "un-repeating random number", and does just that: it will produce every number within the range you specify without repetitions. After it has gone through all the numbers, it produces a bang out its right outlet which if you loop around to a clear message, will restart **urn**. Without this, **urn** will simply shut down.

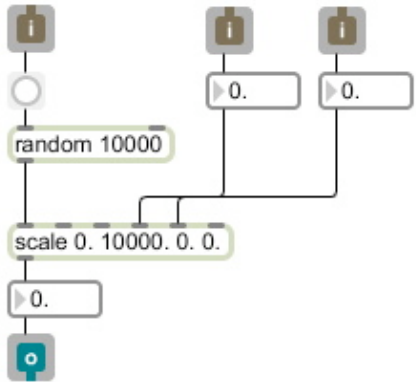


random only works with integers, but notice that it only takes a little math to use it to produce floats.

minimum and **maximum** function similarly. They take in two values, compare them and make a decision about which one to output. When used in conjunction with **random**, they allow you to control the probability of receiving different values. For example, when using **maximum** a higher value will be more likely than a lower one. Check out two other objects that are very similar to **maximum** and **minimum** but are even more specialized: **peak** and **trough**.



drunk is the oddball of objects which produce random values. It takes argument for the maximum value of the range you want it to produce and a step size. Once you build this and turn it on, it's easy to see how it works and also why they call it **drunk**.

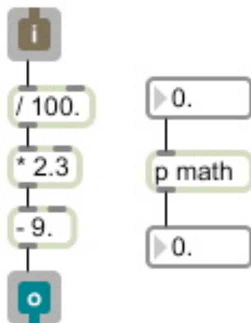


Using **scale** in conjunction with a sizable **random** allows you to produce random numbers in any range. Negatives, floats and ranges that don't begin with zero are all fair game here. Simply specify the range you want as the output range of scale. Mathematically speaking, this is also a truer form of randomness than the one that **random** produces on its own. I've included the inlets and outlets here because I guarantee you this will be one that you'll want to save to your library for repeated use (its referred to as **rrange** in the patch below).

Let's take a look at the patch we built in the last lesson and see how we can sup' it up with some random objects. As always, make sure you can describe the function of each object and its purpose. In addition, brainstorm the various ways you could have a performer respond to the decisions the patch is making. For starters, if the pitch comes out higher than the one you played, do this. . . if a pitch you played is matched at a lower volume, do this. . . if higher, do this. . . if you hear nothing back, do this. . .

Lesson 11: Cleaning Up Your Patches

It is always a good idea to leave your patches looking as clean and readable as possible. While it can be a drag to do so, it is much less of a drag when you do it immediately after finishing a patch. Trust me when I say coming back to a messy one years later and having to essentially rewrite the whole thing just to figure out what's going on will make you hate your life. In addition to looking at ways to clean up your patches, we'll also take a look at some ways to coordinate multiple patches, for as you get over the hump and start to feel creatively free to do what you want, your patches will undoubtedly grow in size.



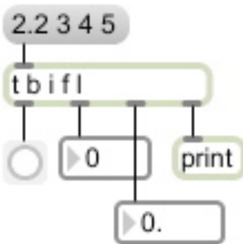
Let's start with some basic cleaning advice. So far we have looked at using **inlets** and **outlets** as a way to construct new objects, however, they can also be used to create what is known as a subpatcher. This is exactly what is sounds like: a patch within a patch. To create one copy all of the objects you want to include and go to the Edit menu and choose Encapsulate. You'll get an object box with a **p**, for subpatcher, which houses all of the code you just "encapsulated." You can still view the code by double clicking on the subpatch when your big patch is locked. You can name the subpatch whatever you want. Subpatchs are especially useful when you have a large bit of code or a highly specialized piece of code that you're going to use multiple times within the same patch. Once you copy and paste a subpatcher, you can give it a new name and change any of the details inside of it without effecting earlier iterations.

Another simple cleaning technique is to hide objects and patch cords when your patch is locked. To do this simply select the object or a group of objects and go to the Object menu and choose Hide on Lock.

Back in the good old days hiding objects when the patch is locked was the only way to get finished patches to look really professional. Max5 has made this technique almost obsolete with the new Presentation Mode. To get to Presentation Mode go to the bottom of your patch and click on the icon that looks like a easel with a line chart on it. To include objects in Presentation Mode highlight them and control+click on them and choose Add to Presentation. In Presentation Mode, no patch cords will appear and you won't be able to do any patching, but you will be able to arrange things so that they look really good and more importantly user friendly. Anything that you move around in Presentation Mode doesn't move around in Patching Mode and vice versa. A bit of advise: don't start messing with Presentation Mode until your patch is all the way done, I mean absolutely ready to roll.

s and **r** stand for **send** and **receive**, and they are extremely useful for communicating between patches. **s** will send its input to any **r** that has a matching name. When I say any **r**, I mean any **r**. It doesn't matter if it's in a subpatcher or a completely separate patch on your computers second monitor, it will get there. When you have to get a message between patches or send a message to multiple places **s** and **r** will save you from long patch cords and tedious subpatching. For an excellent tutorial on networking, i.e. communicating between computers with Max see:
http://www.cycling74.com/story/2006/10/23/104657/91

A **preset** is useful for patches that have many variables which you can choose to control. When you find just the right combination of those variables, a **preset** will allow you to store them so that they can be called upon again. In its default mode it works globally, but you can connect its left most outlet to specific objects for local presetting. Simply shift+click on a button to store a preset. **preset** has a nice help patch which explains all of its many uses.

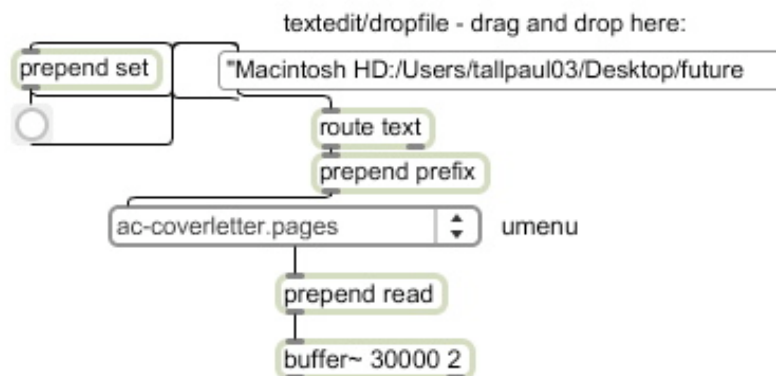


t stands for trigger, and anything that it can do, can be done with others objects so I've included it here because it really just helps you to simplify things. It allows you to send the same message in multiple forms. The arguments are b for bang, i for integer, f for float, and l for list.

These are all objects that you know but with a new look. The top two are **ezadc~** and **ezdac~**. The are identical to **adc~** and **dac~** except in appearance and the fact that you can't double click on them to view the DSP Status menu, which is why I prefer the ugly versions, but the choice is yours.

These two are more visual versions of **gate** and **switch** known as **gswitch** and **gswitch2**. I find these to be useful when writing especially tricky bits of code when having the visual aid helps me to see how the program is flowing. A bit of caution though: these objects start in whatever position they were last saved in, which can be annoying when you reopen a patch and things aren't working and you can't figure out why. To avoid this give them a **loadmess** 0/1 that starts them in the position you want them.

Here's a bit of code that's been extremely useful to me. I admittedly stole it outright from a help patch. I could understand the details of it if I wanted to, but I don't and I know how it works so it's ok. What I do know is that it overlaps a **textedit** and a **dropfile** and that somehow this allows you to populate a **umenu** with the contents of whatever file you dragged and drop onto it from anywhere on your hard drive. This is useful when you want to be able to access a lot of different clips or files - especially when you want to have access to more clips than your computer allows you to keep in active RAM. This becomes even more of an issue when working with video files as we'll see in the next lesson.



The last little bit I will take the time to explain in more detail. The second outlet of **umenu** outputs the menu item as text. **prepend** allows you to put a piece of text before any message that it receives (**append** is also available if you need it). So putting the message "read" before any file that's in your **umenu** allows you to easily load that file onto a **buffer**~, because as you know, "read blank file" is a message that **buffer**~ understands.

Adding Color

panels are one of the easiest ways to delineate different sections of your patches that serve different purposes. They are just big boxes of color and really help to dress up your patches. Once you have created one, get into its Inspector and move it to the background so that it doesn't cover up other objects and choose a nice interior color for it.

The border of any object can also be set to a different color by accessing that objects Inspector.

Lesson 12: Alternate Controllers

This is a broad a wide ranging subject, but one that really expands the possibilities of Max ten fold. While there are many books and systems out there that offer ways to link their equipment to Max, few sources of information start from Max and tell you how to get out - so that's the approach we are going to take. First let's rewind and make sure everybody is on the same page. When we talk about alternate controllers, an example that we've already looked at is a MIDI controller: a device that offers an alternate way to control your patches. But remember back at the beginning when we talked about where Max is positioned in the hierarchy of computer programs, well we can draw similar analogies here. A MIDI controller would be like a highly polished piece of commercial software in the world of alternate controllers, so as you might expect, there is room to explore below this. That room is the land of the microcontroller. A microcontroller is an interface between your computer and the real world. It simply converts information about the real world into a signal that your computer can understand. While you might not be aware of it or have maybe even never heard the word before, they are literally all around you. There are tons of them in your car, for example, and they're even in your TV remote. The one that I'm going to recommend that you begin with is the Arduino for a couple of reasons. 1. It's open source 2. It's reasonably priced and 3. It's positioned in the heirarchy of microcontrollers on a similar plane to Max: it's not completely raw, but it's still extremely flexible. Spend some time on the Arduino website just so you get an idea of what a microcontroller even looks like. If you do decide to buy a board, I recommend the latest version of their USB board.

What can you do with a microcontroller?

As I was saying, microcontrollers help to get information about the real world into your computer. They do this by converting information from sensors into a signal your computer can understand. This brings us to our next big topic: what is a sensor? A sensor can be as simple as a button, switch or knob and as complicated as an infrared motion sensor. Here's a short list of the all the physical phenomenon that have corresponding sensors: light, motion, heat, force, bend, tilt. The ability to specifically tailor a system of sensors to your needs opens up Max to a myriad of possibilities from interactive art installation to sculptural scenarios to unique ways to interface with musicians and dancers to instrument building. The possibilities become even more endless than they already are, and there's still another dimension to this whole realm that I haven't even mentioned. Microcontrollers can also output information that can be used to control everything from an LED to a stepper motor. In other words, think of the land of microcontrollers as an open invitation to feel empowered about thinking of ways to sense and control the physical world.

Getting Started

Once your board arrives, if you've decided to purchase one, spend some time on the Ardunio website under the "getting started" and "learning" tabs. The Arduino website

has become extremely friendly for beginners over the years, which is another reason why I'm endorsing this system. Microcontrollers are perfectly capable of functioning on their own, even without a computer, understanding how they function is the first step to understanding how they interact with Max. In addition, spending just a little bit of time trying to tackle some of the straight Arduino code will dramatically improve your understanding of how Max code functions. So put this book down and get online.

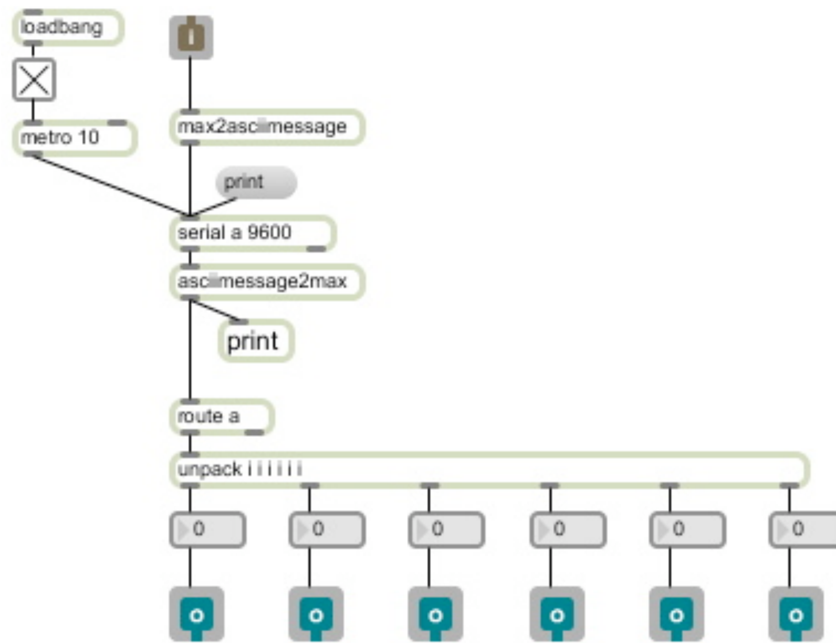
As you should have picked up from the Arduino website, most microcontrollers still communicate with computers using serial ports, which are now simulated with USB connections. Serial communication is based on a very old computer protocol known as RS-232 which harkens back to the days of printer paper with all those little holes on the side of it and computer monitors with flashing white boxes and no mouses. Anyways, all you really need to know is that Max is capable of communicating over serial ports as well with an object called **serial**. It forms the basis for communication between Max and any other program, including microcontrollers. Its first argument specifies a port and the second argument specifies the baud rate, basically the speed at which data is being passed back and forth (9600 is a good starting point). Sending **serial** the "print" command will print a list of all available ports to your Max window, if you have your Arduino hooked up you should see its port listed first. You can set the Arduino's port from within its software (when two options are given choose the ".cu.")

At this point, you should get back online and download the Simple Message System from this website:

http://www.arduino.cc/playground/Code/SimpleMessageSystem

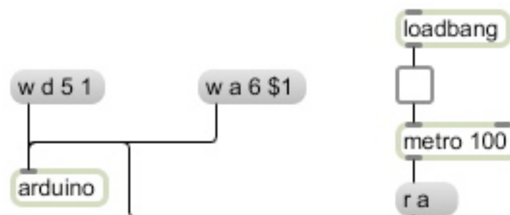
This is a system which facilitates the communication between your Arduino and your Max patches. In the folder, you'll find instructions about how to install the needed files into Arduino's library. You'll want to take all of these steps and upload the SimpleMessageSystem program onto your Arduino board. Once you do this, any programming you want to do for the Arduino can be achieved from within Max.

What follows is admittedly a quick and dirty introduction, however, I hope that your curiosity has been peaked about the possibilities of exploring microcontrollers further. There'll be some other things you'll need to figure out on your journey down this path which fall beyond the scope of this book and include soldering, breadboarding and an understanding of some basic circuitry, but do not be afraid - remember how scary Max used to be!



I've set this patch up to be an external because if you get into doing this kind of work it's helpful to have an "Arduino" object which you can easily call on. **max2asciimessage** and **asciimessage2max** are objects which come with the SimpleMessageSystem. If you open them up, there are a few objects which we haven't covered which help to translate information to and from the ASCII protocol which is how they have to be broadcast over the serial port. **spell**, **fromsymbol** and **itoa** all have adequate help patches. Remember that we first looked at the ASCII numbers when dealing with **key**.

You can now write to the Arduino and pole it with the following types of messages:



The first one, "w d 5 1" is translated via the SimpleMessageSystem objects to mean "write digital pin 5 to HIGH" to the Arduino. The Arduino is also capable of simulating analog out on a select number of pins which is illustrated in the second example. "w a 6 $1" is translated by the Arduino as "write analog pin 6 to the value of the number box (0-255)". The third example illustrates how you would pole all of the Arduinos analog ins at a sampling rate of 10 times per second. Sending this message to your "Arduino" object will cause the Arduino to return values to Max which can be viewed coming out of the six outlets at the bottom of the patch.

Lesson 13: Jitter

If you want to get yourself really confused, really quickly, start reading the Jitter tutorials. If you think that the Max tutorials make leaps too quickly, wait till you come upon the jargon that bogs down anybody who is trying to get a handle on Jitter. As has been our approach so far, we're going to come at Jitter with the hopes of breaking it down. Admittedly, we won't cover all of the objects, but I'll do my best to give you the basics and some general ideas about how to approach Jitter.

First of all, if you haven't figured it out all ready, Jitter is the set of objects that expands Max into the visual realm (Gem is the PD equivalent). There are two freestanding programs which Jitter stands on: Quicktime and OpenGL. These two programs mark the two different functions of Jitter which I am going to propose.

Let's start by looking at what Jitter can do via Quicktime.

Quicktime

Any video format that can be opened and played using Quicktime, will work best with Jitter. Other formats can work, but stick to those formats which QT likes best, like .mov, it'll save you a lot of hassle. Before you can walk you have to crawl, so let's dumb down movie playback to its essential elements.

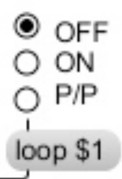
(jit.pwindow)

jit.qt.movie is kind of like a combination of **buffer~** and **play~** for a movie. A movie file has to be read into it as you can see, but it's also the object that plays back the movie. To playback the movie a **qmetro** has to be on and attached to **jit.qt.movie**. **qmetro** is a special kind of metronome, the details of which you should not concern yourself, but its function is to tell **jit.qt.movie** how often to output an image from its internal "buffer". 30 ms is the standard setting, and while this is not exactly equivalent to frame rate it can be thought of similarly. The object **jit.pwindow** provides an in patch screen which allows you to see the movie. Notice that Jitter patch cords which actually handle an image have their own unique look just like audio patch cords.

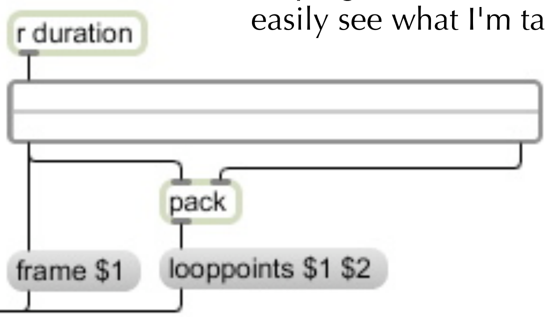
Jitter objects pass around what is known as a matrix. A Jitter matrix is essentially a grid of values that describes the color of each individual pixel which comprises a single image. Open your Max window and run this patch to get a visual idea of what a matrix actually consists of. **jit.print** allows you to print the numerical values of a Jitter matrix into the Max window.

Now that we're starting to crawl around a little bit, let's see if we can take our first baby step by looking at some of the ways we can affect the playback of a movie.

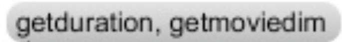
One of the simplest aspect of playback to control is speed which is communicated with the rate message. 1=normal playback, 2=twice as fast, .5=half speed, etc. Experiment with **line** and see what you can do. Negatives are also fair game here and can be used to interesting effect.

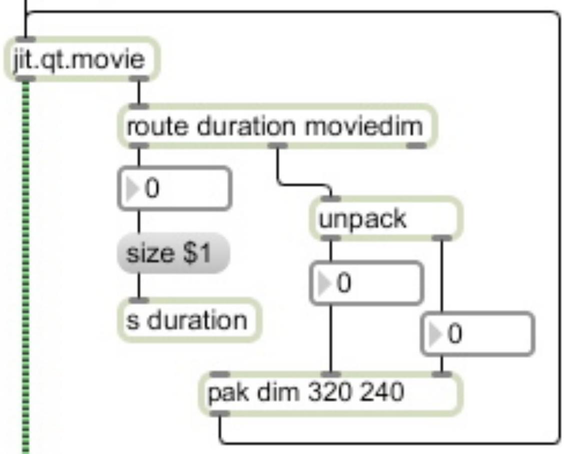


Looping is another obvious parameter to control. With video there's an option that we don't typically encounter with audio that is ping-pong. Ping-pong looping plays something through start to finish and then loops by playing from the end of the clip back to the beginning instead of looping from the end straight back to the beginning. Try it out and you'll easily see what I'm talking about.



An **rslider** is a special type of slider that allows you to select a range of numbers: which works well for trying to set individual loop points. Attaching “frame” to the **rslider** allows you to visualize the loop points your setting on the **jit.pwindow**. It is useful to use this in conjunction with a **preset** so that you can easily recall the loops that you like.



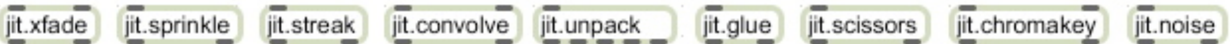


There are a plethora of messages that allow you to query **jit.qt.movie** for information about the video that you read into it. “getduration” and “getdim”(ension) are two of the most useful. When asked for, these will come out of the right outlet of **jit.qt.movie**. Putting the duration into a size message for **rslider** will ensure that your slider covers the exact range of your clip. Sending the dimensions immediately back into **jit.qt.movie** sets the objects buffer to conform to the original dimensions of the movie ensuring the best quality of playback and that no buffer space is under or over utilized.

pak is a special version of **pack** which outputs a list whenever any of the values change. Remember that **pack** only outputs its list when the left most inlet receives a new value.

You are not limited to playing videos only inside of your patches. To create a screen which exists outside of your patch, call upon **jit.window**, give it a name, and it seems to like to know that you want it to be rectangular. It's also a good idea to pass it the original dimensions of your movie. The subpatcher, "**p** esc-fullscreen" can be stolen from **jit.window**'s help patch. Inside, is a **key**/**sel** combination which allows the esc key to toggle a "fullscreen" message for **jit.window**. The window that you create can easily be dragged onto your second screen for projection so that you can still operate a patch from your main screen.

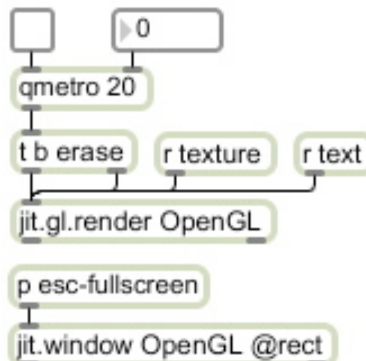
Now that we've hit some of the major ways that you can control playback, here is a laundry list of objects which actually affect the image.

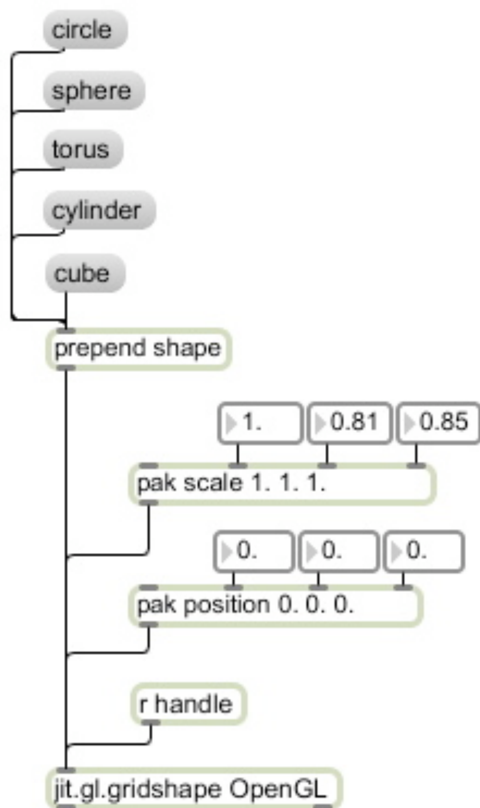


All of these objects offer different kinds of video effects, and all have clearly written help patches which I highly recommend stealing from. Check them all out. They are all worth it.

OpenGL

The other half of what Jitter can do is based on a program called OpenGL. OpenGL is a rather simple program that 3D designers and animators rely on to test their models after they have sculpted them in programs like Blender and Maya. OpenGL is kind of like Preview for 3D object files. The basis of OpenGL in Max is **jit.gl.render**. **jit.gl.render** takes an argument for a window name and can be either a "pwindow" or an "outside of your patch window". Once you have set **jit.gl.render** to a window, only OpenGL objects (jit.gl.___) will be able to display things on that window. This is why I say that there are really two halves to Jitter: don't think that you can operate OpenGL on top of a Quicktime movie playing in the background cause it ain't gonna happen. **jit.gl.render** needs a **qmetro**, just like **jit.qt.movie** does, with one exception: every time you want **jit.gl.render** to display a new image you have to erase the old image. This is why we have the **t**rigger set up with a bang and the message to "erase" (remember that Max processes data from right to left).





While it is possible to display 3D objects that you have rendered in other programs with Max, Jitter also offers a variety of ways to sculpt 3D objects from within the environment. All digital 3D models usually begin with a "primitive" geometric figure. **jit.gl.gridshape** is the most basic object which creates these shapes (there are more than I have listed here). Notice that it also has to have an argument for a window, and, furthermore, it has to be a window with a **jit.gl.render** operating on it or you are not going to see anything. In addition to rendering these shapes, **jit.gl.gridshape** also understands messages which affect the "scale" and "position" of the objects it creates.

There is one way, and only one way, in which you can meld the two halves of Jitter. **jit.gl.render** can create a "texture" on an object out of a still or moving image. Notice that you have to send an additional message, "usetexture", before the texture appears on the object. This makes it easy to switch rather rapidly between textures.

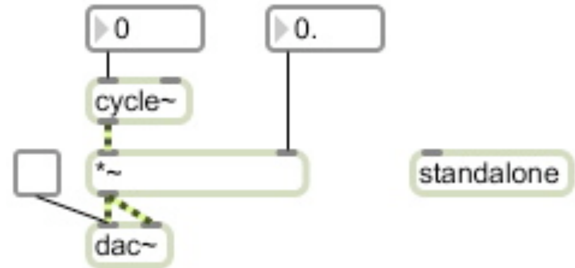
jit.gl.handle allows you to use your mouse to rotate objects that are being rendered. Sending the message "visible" makes all of the handles which allow you to rotate an object on different axis visible. This is a useful object when it comes to creating interactive installations.

In addition to shapes, OpenGL can also render text in either 2 or 3 dimensions. Guess which one **jit.gl.text3d** does? The "text" message will render whatever text you insert after it. **jit.gl.text3d** understands the "scale" and "position" messages that we used with **jit.gl.gridshape**. Text can also be “textured” and used in conjunction with **jit.gl.handle**.

Lesson 14: Building Applications

At the end of the day, when you have crafted the next piece of genius audio/visual software, something you might want to do is make it into a real life application that you can share with your friends. Max makes this incredibly easy for you to do.

Take your patch, however simple or complex it might be and include a **standalone** object. We'll use the following patch as an example:



The inspector for **standalone** lets you set the parameters you want for your application. Most of the defaults are what you're looking for, but it's good to know where this all comes from. Checking audio and midi support will ensure that all of Max's audio/MIDI drivers are included in your application.

Tip #1: It's important to have a way to turn DSP on and off from inside of your patch when getting ready to build an application.

Now follow these steps:

1. Go to File / Build Collective / Application

This screen allows you to write a script that serves as the start up script for you application. Basically, it tells the application everything it needs to look for and load upon start up. The command "open thispatcher" is, rather obviously, referring to the patch from which you started the application builder. Any subpatches that are within your application will automatically be included, but if you have sound or text files that your patch needs to run, choose whether they are in a file or folder and include them in your script. If your application is going to be made up of more than one patch, you'll need to include all of the patches in the script that the application will need.

2. Once everything is in your script tell it to "build". The order of items in your script is not important.

3. When you get to the Save screen, there will be a drop down menu at the bottom which will give you an option about the format for your application.

Here's the difference between the two formats: A Max collective requires Max Runtime in order to function properly. An application does not; an application can stand on its very own on any machine of the same platform which you used to build it. (This is an important point, an application built on an Apple computer will not function on a PC and vice versa.) Collections are really only useful when a program is dependent on a lot

of different patches and external files, for as you know, a normal everyday patch can be opened and run from Runtime without being saved as a "collective". I guess the argument would be that saving it as a collection doesn't allow the recipient to mess with your code. Anyways, building Applications is a far more useful avenue in my mind, and who knows, potentially lucrative as well.

Object Index by Lesson

Object	Lesson
!=	2
*~	1
+	6
-	5
/	5
<	2
accum	6
adc~	4
adsr~	8
append	11
bondo	9
bucket	9
buddy	9
buffer~	1
button	1
change	6
comment	1
counter	3
cycle~	8
dac~	1
del	5
dial	9
dropfile	11
drunk	10
expr	7
ezadc~	11
ezdac~	11
fiddle~	6
freeverb~	7
fromsymbol	12
ftom	8
function	8
funnel	9
gate	6
gizmo~	7
groove~	6
gswitch2	11
gswitch	11
i	5
itoa	12
jit.gl.gridshape	13
jit.gl.handle	13
jit.gl.render	13
jit.gl.text3d	13
jit.print	13
jit.pwindow	13
jit.qt.movie	13
jit.window	13
key	6
kslider	7
line	5
line~	1
loadbang	6
loadmess	6
maximum	10
message	1
meter	4
metro	3
midiin	9
midiparse	9
minimum	10
mtof	6
number(~)	1
o	5
onebang	6
pack	5
pak	13
pan2~	7
panel	11
peak	10
pipe	5
play~	1
prepend	11
preset	11
print	5
qmetro	13
r	11
radiogroup	7
random	10
record~	4
rect~	8
reson~	7
route	9
router	9
s	11
saw~	8
scale	9
scope~	8
sel	2
sig~	6
slider	7
spell	12
spigot	6
split	9
spray	9
standalone	14
switch	9
t	11
tapin~	5
tapout~	5
techno~	8
textedit	11
toggle	2
tri~	8
trough	10
umenu	11
unpack	5
urn	10
zl	9