The Care and Feeding of Lists

The Max tutors introduce lists without talking much about what they might be good for. The truth is, many Maxers use lists sparingly, if at all. However, most of us find lists simplify patch structures and streamline calculation. Use a list whenever you have a group of numbers that apply to a single entity or operation. For instance, figure 1 will play a note 60 at a velocity of 110 for 500 ms.[1]

Figure 1.

You can define the meaning of a list any way you want. Figure 2 shows two ways to use lists to represent chords.

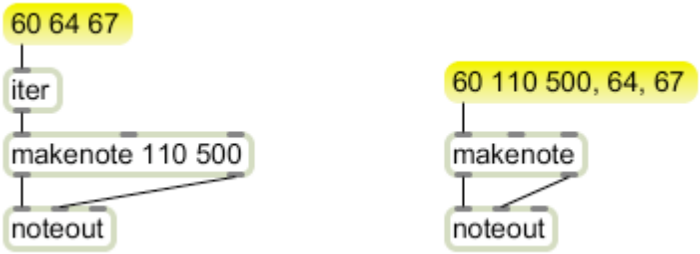


Figure 2.

The iter object converts a list into individual messages. They follow each other as fast as possible, but they are distinct. The left side of figure 2 plays the notes 60 64 67 simultaneously using the current velocity and duration of makenote. The right side version does the same with a twist. A comma in a message box breaks the items into separate messages. When the message box is triggered, all messages are sent as fast as possible. In figure 2, the first part of the message plays the note 60 and leaves the velocity and duration of makenote set for the following individual pitches.

I have often seen lists used to organize:
Notes
Chords
Rhythm patterns
Pitch rows
Scales
Sets
Graphic coordinates

[1] Many objects will treat a list as values applied to successive inlets, but not all. You have to check the reference in addition to the help file to discover this.

Colors
You will doubtless come up with your own systems. Whatever the content means, you will need to build, organize, manipulate and disassemble lists. I am going to offer techniques in two contexts-- basic Max objects, and lobjects.

First, let me review the definition of a list- a list is a Max message consisting of more than one number or symbol. The numbers may be ints or floats. Strictly speaking, the first member of a list should be a number--if it is a symbol, the list will be treated like a command, which may result in an error message. (This restriction is no longer strictly true-- quite a few objects can deal sensibly with lists of symbols.) The order of items in a list is maintained, and items may repeat. The position of an item may be indicated by its index, which is usually counted from 0. You occasionally find an object which counts indices from 1 (1-indexed), so you need to pay attention.

Lists have an upper limit on length, which has been growing over the years. As of Max 4.5, the limit was 256 items. In version 5, this has been relaxed and many objects have adjustable lengths. The default is still 256. Third party objects will have to be re-written to accommodate adjustable lengths, and that process will probably take years. Most processes that require very long lists can be done in Jitter.

List management with basic Max objects

Building Lists

A list can be defined by typing into a message box and sent along by clicking or banging the message. One item of this list can be made variable with a $1 token-- then a number box can make a series of similar lists.

Pack is the fundamental list constructor. The arguments to pack set up a default list, defining the length and initial contents. Each argument will have an associated inlet. Note well that the type of each argument defines the type of the item in the list, int or float. Leaving the dots out of packs that will handle floats is a common cause of problems. Pack will output its contents when it is banged or receives a message in the left inlet. **Pak** (pronounced "pock") is a similar object that puts out whenever any inlet receives a message. This makes it useful in user interfaces, but it's easy to generate a lot of partially modified lists. The test shown in figure 3C produces four messages for each button push.

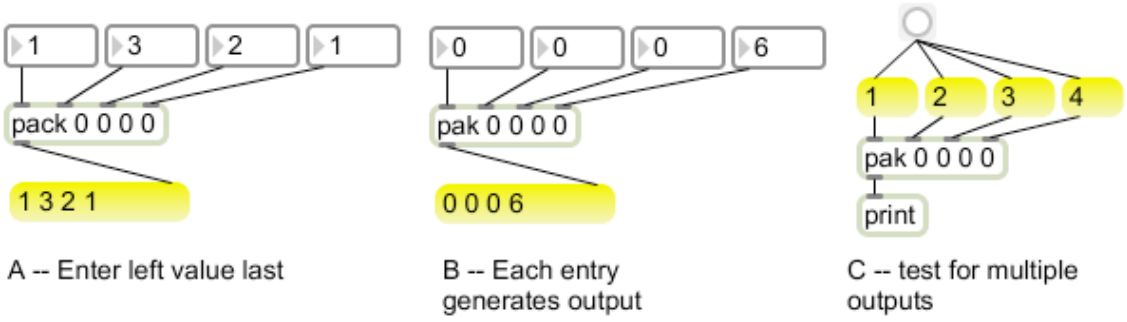


Figure 3.

If a list is applied to any inlet of pack or pak, it is treated as inputs to all following inlets. If nothing is connected to an inlet, the associated argument will always be the same in the list. We often use pak to generate commands by putting the name in the first argument.

Thresh creates lists out of individual inputs. As input arrives, the values are collected-- if a specified time elapses with no new input, the list is output.

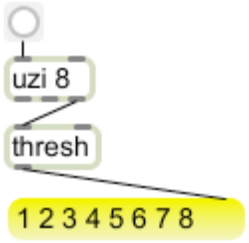


Figure 4.
The time interval can be set by an argument or input at the right. If the time is pretty long the list be triggered prematurely by a bang.

Manipulating Lists

When we want to cut up and rejoin lists, we turn to the **zl** object. In fact, we turn to the zl object for practically any list manipulation. The zl object is really a growing set of objects[2]. The function is determined by the first argument-- this will be a word, like 'slice' or 'join' that indicates what the object should do. There may be additional arguments to control the function. For instance, **zl slice** breaks a list into two lists. It requires an argument to specify where to slice.

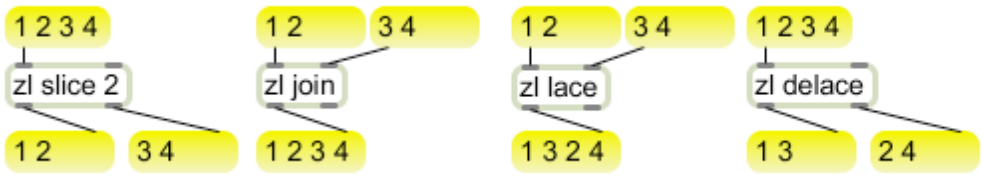


Figure 5.

[2] The zl objects originated in some third party objects authored by James McCartney, of supercollider fame. The first version of Max had no list operations to speak of, so James coded slice, join, sort and a few others and distributed them via the internet. These quickly became a fundamental part of every Maxer's object list. (This was 1992, when we were all more or less on a first name basis.) Later versions of Max were distributed with an assortment of third party objects and McCartney's were included. When Apple changed to the ppc architecture, all objects had to be re-written, but James was not available. After a period of uncertainty, David Zicarelli brought these functions into Max proper as the zl objects. Andrew Brown once quipped zl stands for a reverse engineering of Elsea's objects. That didn't seem so at first, but some recent additions look awfully familiar. [To get the joke, you have to know the proper pronunciation of my name, which is L-Z.]

Zl join will combine two lists into one. A list in the left inlet will be connected to a list already received in the right. Lace builds a list by alternating members of two input lists. Delace does the opposite, producing independent lists of odd and even members. **Zl ecils** is the same as zl slice, but the slice point is counted from the back. This a way to get the last member of a list, even if you don't know its length.

Zl union and **zl sect** combine two lists following rules for sets:

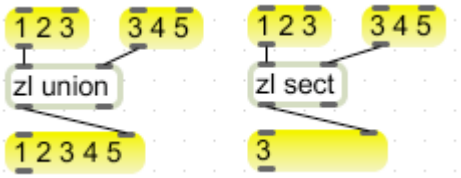


Figure 6.
The union contains all values that appear in either set, and the intersection contains only values that appear in both.

Zl group collects values until there are enough to fill a list:

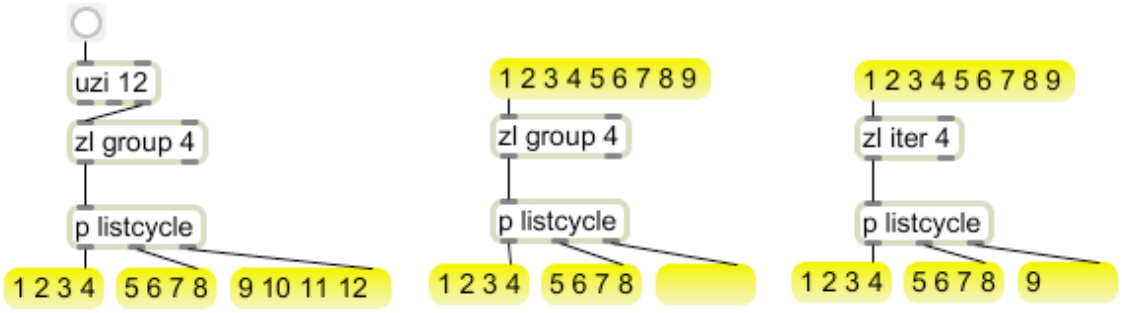


Figure 7.
If a list comes in, it is disassembled. The groups in figure 6 are set to make four item lists. In the center example, three more items are required before a list starting with 9 will be produced.

Zl iter (not to be confused with plain iter) will also break a list into small ones, but any extra values will be output as a short list. (The listcycle subpatch contains objects to display successive lists side by side.)

Zl stream is something like zl group, except each input produces a list that includes the previous n inputs.

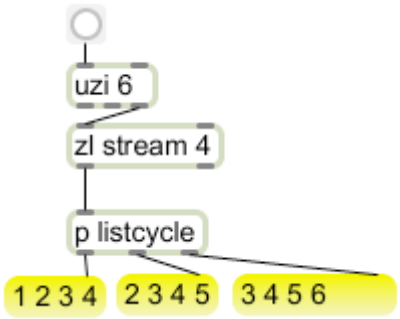


Figure 8.

Modifying List Contents

The are zl objects to modify the contents of lists also. **Zl sort**, **zl scramble** and **zl rev** do exactly what the names suggest, rearranging the contents.

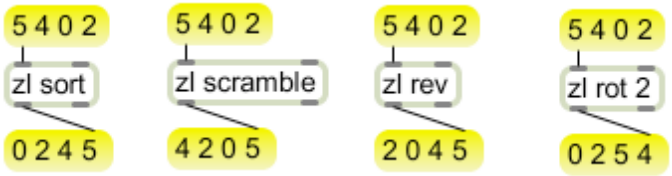


Figure 9.
ZL rot is short for rotate, which moves the last few elements of a list to the beginning.

Some zl objects delete specific values.

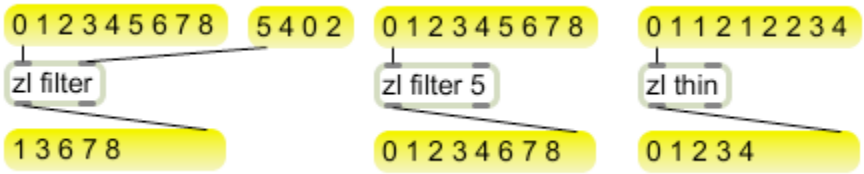


Figure 10.

Zl filter removes values that are contained in a right side list. The filtered values can also be set by arguments. The right outlet of filter reports the indices of the items that pass the test. **Zl thin** removes duplicate values.

Math with lists is done with vexpr, but some zl objects do math on the contents of a single list.

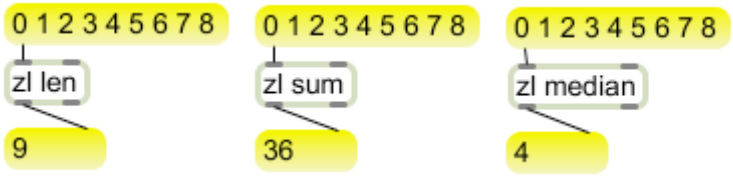


Figure 11.

Zl len shows the length of a list, **zl sum** adds up the members. **Zl median** sorts the list, then returns the center value. (If the length of the list is even, you get the mean of the two center values.)

Math on Lists

Math operations on the individual members of a list can be done with the **vexpr** object. Vexpr (vector expression) is much like expr-- it requires a line of C code to define its operation. Like expr, tokens like $i1 or $f4 represent values applied to inlets. Unlike expr, the inlets take lists, and the result is a list with the operations performed member by member. The length of the output list is determined by the shortest input list.

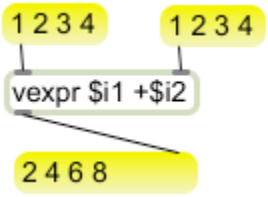


Figure 12.

The expression can be quite complex, so this single object can take care of almost all math operations between lists. The use of expr is covered in the *More Math* tutorial.

If you want to apply a single value to all members of a list, set the vexpr attribute @scalarmode to 1.

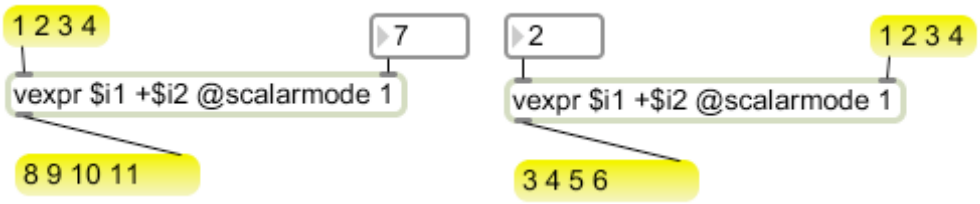


Figure 13.

Organizing Lists with coll

The best way to organize a bunch of lists is with **coll**. Coll is a collection of data, basically lists stored at selected addresses. Whenever I talk about coll, I remind students to check the Save Data with Patcher line in the inspector. You don't always need to save the data that's in a coll, but it never hurts to do so and this habit will save a lot of grief.

Figure 14.

You can open the coll editor with a double-click. The format of the coll editor may be familiar-- each line starts with an address, which may be an int or a symbol. A comma separates that from the data, which is a list. The data is terminated with a semicolon.

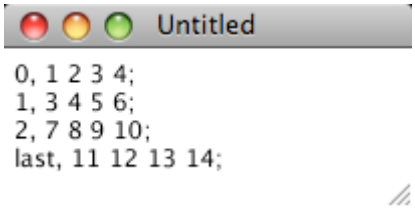


Figure 15.

To get a list out of a coll, apply the address to the left inlet. If there's no data at that address, nothing will happen. When the input matches a stored address, the associated list is output from the left outlet, and the address from the second left.

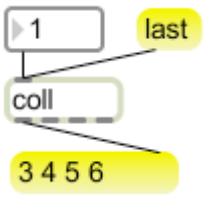


Figure 16.
In figure 16, the most recent input was the number 1, which retrieved the list at address one. The symbol last would retrieve the list 11 12 13 14, not because it is the last, but because the symbol 'last' is the address. You can set up an association between a symbolic address and a numerical one, then use either to retrieve the data.

You can also step along the data in order with the *next* or *prev* commands. The commands start, end or goto cue up the next pointer without putting anything out. This determines the address you will get with either a next or prev command. Surprisingly, the order is the order addresses show in the edit window, not numerical order. If you type address 4 data between address 1 and 2, the next command will give 4 after 1 and before 2.

You can sort the lists with the *sort* command. Sort takes two arguments-- the first is a direction flag, -1 for ascending 1 for descending results. The second argument is the item in the lists to sort on: -1 is the address, 0 the first item in the lists and so on. If sorting leaves the addresses out of order, there is a renumber command.

Coll has several commands that modify the data contents. You can experiment with these in the help file:

Insert n data places the data at address n. If there is already data at n, it is moved up.
Merge n data adds the data to the data already at n.
Clear empties the coll entirely.
Remove n deletes the data at address n, leaving an empty slot.

Delete n deletes the data at address n and moves data above n down.
Nth n m reports element m from the list at address n. This is one of these odd functions that counts elements beginning with 1. So nth 4 2 returns the second element of the list at address 4.
Nsub n m data places the data in element m at address n, replacing whatever is there. If the list is not long enough, nothing happens.
Sub does the same thing as nsub, then outputs the list at the address.

Colls can have names. If they do, Max looks for a file of that name to load in when the patcher is opened. You can edit these files with a text editor. The write and read commands let you save and retrieve the coll's contents. This is not the same as saving the file with the patcher. If you have a named coll set to save with patcher (embedded), you will be prompted to save when you close the patcher-- this does not update the named file, and the named file (with old contents) will replace any unsaved (via the file) changes.[3]

When a coll has name, two coll objects with the same name will share the same data. This can greatly simplify the layout of complex patches. If you want two colls to share data, but not refer to a file, add a second argument.

Other features of coll can be found in the reference. An example of colls in use will be discussed in the latter part of this paper.

Disassembling lists

Eventually we need to get the contents out of a list. **Unpack** is the basic object for this, as it shunts the items in the list to individual outlets.

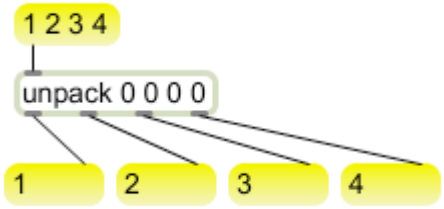


Figure 17.
Unpack will have an outlet for each argument. The arguments set the data type for the outlets, but are not otherwise used. A symbol like s in the arguments will set the data type to symbols. If a symbol is received in a numerical slot, a 0 is output.

Sometimes were only need one item from a list. The zl nth and zl mth objects provide this:

[3] If that isn't confusing enough, you always see a "save changes?" dialog when you close a coll's editing window. You want to save the changes, but this only transfers from the window to the coll-- it will still need to be saved when you close the patcher.

Figure 18.
Zl nth specifies the item by position, starting with 1.
Zl mth specifies the item by index, starting with 0.[4]

Sometimes we want to simply store a list temporarily, as we might store numbers in an an int or float object. Zl reg will do this, producing the list when it is banged.

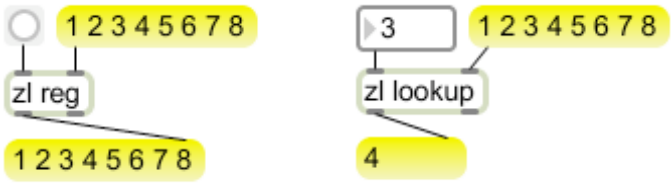


Figure 19.
Zl lookup will store a list and produce individual items by (0 based) index when requested. An int in the left inlet will produce a single value, a list of indices will get a list of just those locations. The indices need not be in order.

Zl queue and **zl stack** will collect data into a list, then produce one item at a time when they are banged.

Figure 20.
Zl queue outputs the oldest item first (First In First Out), and zl stack outputs the newest item first.

List Management with Lobjects

Most[5] of the Lobjects deal with lists in some way. I won't include example patches here, there are plenty in the help files.

[4] How did we get into this situation? The original version of Max was written in Pascal, which refers to the first item in an array as number 1. When Max went commercial, it was rewritten in C. Arrays in C are found by their address, and the first item is at this address + 0. Objects written by C programmers generally index starting with 0. All languages develop quirks over time, we just have to get used to them.

Llist is a simple list container and works exactly like the int or float objects. A list in the right is stored, a list in the left is stored an output. Bang outputs the list. So far, it is like zl reg. However, Llist has a *get* function-- the message get n with report the nth member of the stored list. There is also a *set* function. The message set n x will change the nth item to value x.

Building lists

To easily define a list, type it as arguments into an Llist object. The following objects create lists in a variety of ways.

Lcatch gathers data into lists of defined size, but if the data stops short of a complete list the remainder is output as a short list after a defined pause. If is like a combination of zl group and thresh.

Lchunk gathers data into lists of defined size. You can set a magic value that triggers output before the list is full-- thus a return character or sysex EOX can complete a message. Lchunk has a parameter called wrap[6]. When wrap is on, lists that are shorter than the defined size are passed through immediately. When wrap is off, the data in short lists is accumulated until the size is reached. Wrap defaults to on.

Link appends data to an initial list defined by arguments. If data is received in the right inlet, there is no output. Data in the left is appended to the list and the list is output. After output, the appending process begins again. The clear command removes any data attached to the argument list with no output. The set command with arguments sets the head of the list to the arguments.

Lmerge combines two lists by interleaving values. An argument of the type 2:3 sets up the pattern of alternation such as two from the left then three from the right (up to 9:9). A parameter named useall sets the length of the output. If useall is 0 (the default) output will end when the left list is used up. If it is true, any items remaining in the right lsit are tacked on the end.

Lifo gathers items into a list, placing them in front of previously collected data. Items in the right are collected, items in the left are collected, output is triggered, and the list is cleared. This last in first out behavior means items swept in by the usual right to left patch action will appear in the list just as they appear in the patch. If lists are colleted, their contents are not rearranged.
Bang outputs the list and clears it.
Peek outputs the list without clearing it.
Pop reports and removes the first item in the list.

[5] For descriptions of the ones that don't, see "More About Lobjects".
[6] Converting such parameters to attributes is on my to-do list.

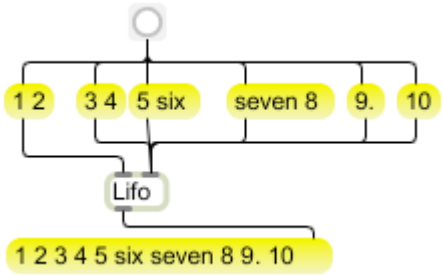


Figure 21

Lrun generates number series. With no arguments, a list with two values will be filled in, the list 2 6 will produce 2 3 4 5 6. An int in the left will generate the a series that starts with 0. This works in either direction. The list 6 2 will produce 6 5 4 3 2, and a negative int will produce a series that starts at 0 and runs down. Once a series has been started, bangs will continue the series. A argument or int in the middle inlet will set a different starting value when an int or bang is received in the left. A second argument or int in the right will set a step size, so you can count by 2's or whatever. Reset will make the inputs produced by bangs begin again.

Lrepeat generates lists of repeating patterns. An argument or input can set the number of repeats.

Lbag is similar to bag. The right inlet determines whether data in the left will be stored or erased, 1 for store, 0 for erase. Data may be in a list. There is no output from Lbag unless it is banged, then the contents are reported in a list. The list is sorted with no duplicates. If Lbag is empty when it is banged, the right outlet will bang. The contents of Lbag may be initialized with arguments.

Lreg is optimized to show the current status of a keyboard. Usually both inlets of Lreg are connected to pitch and velocity of notein. The number of a note on will be stored, and the number of a note off will be deleted. The keyboard state is output as a list each time data is received. If a note off empties Lreg, the right outlet will bang. If there is an argument of 1, Lreg keeps track of the number of times each key number is received. Key numbers will only appear in the output list once, but note offs must catch up with note ons before they are removed from the output list.

Lbuild is a high powered list builder. It has two inlets- data in either inlet is added to the internal list-- action at the left inlet triggers output of a copy of the list, but the list is retained. A bang also triggers output. If the list is empty, the center outlet bangs The internal list can be manipulated by the following commands:

Insert n data -- inserts the data at location n, moving the data already at n to the right. The location is 0 indexed, so insert 0 5 will stick a 5 at the beginning of the list.
Delete n m -- removes m items starting at index n. Delete with no arguments will remove the first member of the list. Items that are removed are output from the right outlet.
Drop n -- removes the last n items from the list. Items that are removed are output from the right outlet.

Get n -- outputs the value at index n from the left outlet. If there is no value at the index, the center outlet bangs.
Get n m -- outputs m items starting from index n from the left outlet. You always get m items. If the end of the list is reached, retrieval starts over at the beginning.
Next m gets the next m items following the last get command. This will cycle through the list repeatedly. Next with no argument gets one item.
Clear removes all items in the list without output.

Modifying List Contents

These objects change the order of lists or change values based on place.

Lswap performs arbitrary rearrangement of lists. The arguments form a template to determine the contents and order of the input. The arguments may be integers or symbols. Integers indicate the 0 based index to place at that spot. Negative integers are counted backwards from the end. A symbol is interpreted to mean "through" if it is between integers, "continue" if it is at the end. Typical templates:

1 2 3 will produce a list with the elements at indices 1 2 and 3, (the second to fourth items).
0 2 4 6 will include the first four even indices.
-3 -2 -1 will include the last three items.
*0 * -1* produces the entire list.
*-1 * 0* will produce the list reversed. (zl rev can do this.)
*0 * -1 * 0* will produce a palindrome of the list.
*0 2 **will produce every second item, starting at the beginning.

Items may be repeated, and templates may be combined.

Lror rotates the contents of a list to the right. This can work two ways. If a list is defined by arguments or a list received in the right inlet, an int in the left inlet will produce a list with the requested rotation. A negative number will rotate the list to the left. If there is a single int argument or an int received in the right, a list applied to the left inlet will be rotated and output. These two processes are independent of each other, although a patch that tried to use both features on the same object would appear rather confusing. An Lror object with one or no arguments contains a default list of 1 0 0 0 0 0 0 0 0 0 0 0.

Lshiftr shifts the contents of lists to the right. This can work two ways. If a list is defined by arguments or a list received in the right outlet, an int in the left will produce a list shifted right -- the first value of the list is repeated and the last values of the list dropped. Thus 0 1 0 0 shifted left by 1 would produce 0 0 1 0. A negative number would produce a left shift. If there is a single int argument or an int is received in the right, a list received in the left will be shifted as specified. Bangs repeat the last shift operation. The default list is 1 0 0 0 0 0 0 0 0 0 0 0.

Lshiftreg -- A list is set by arguments or input at the right. Data that comes in the left inlet is placed at the start of the list and the last item is lopped off. The bumped item is output from the right outlet and the shifted list is output from the right. Lists applied to the left input are itered. That means the data will be reversed in the list, but will be right way around when ejected from the right.

Lfilt removes undesired items from a list. These items are set by arguments or a list in the right inlet. Data in left is checked against the unwanted list and no match items are passed through. If all items are removed, there is no output.

Lsieve allows listed items to pass through. Approved items are set by arguments or a list in the right inlet. Data in the left is compared with the approved list and passed if there is a match.

Lscale modifies values in a list to fit a desired range. Lscale requires four arguments. The first two are the range of input, the third and forth set the range of output. Data input on the left is modified proportionally to these ranges. There is a parameter called limit. If it is 1, output data may not exceed the specified range. The ranges may be adjusted via the right inlets. Ether input or output range may have minimum higher than maximum.

Lscaler modifies values in a list to fit ranges tailored to each member. The input and output range minima are fixed at 0. The input maxima can be set by a list in the right inlet. (Default 127) The output maxima are set by arguments or a set message. Maxima may be negative for inverted action. If input or output offsets are required, they can be produced by additional Ladd objects.

Lcut sets all values that fall below a threshold to 0. This works two ways. If a list is stored by arguments or the right inlet, a number in the left inlet will output the list with values below the number set to 0. A single argument or value in the right inlet will set a threshold to process lists coming in the left.

Lpad adds values to a list to extend it to a desired length. Arguments or inlets can set the length of output lists, where the input values appear in the list and the padding value. The leading parameter will right justify the input list.

Lreplace replaces specified values in a list with new values or lists. Arguments or list in right set the beginning list. When lists are received in the left, the first item of it the target value-- all occurrences of that value are replaced with the remainder of the list. A bang outputs the modified list. There is a delete command to remove specified values. The restore command returns the list to the original form.

Lmask forces specified items in a list to desired values. This is defined by a mask list containing the desired values in place and symbols in the positions that should not be changed. The mask may be set by arguments or a list in the right inlet.

Lunique removes duplicated items from a list. An argument or number in the right inlet sets the number of repetitions to allow.

Math on Lists

There are several Lobjects that perform member by member operations on lists. They all have these common features:
Lobjects may be initialized to contain a starting list.
Lobjects automatically choose vector or scalar mode depending on the argument or right input. Ladd 1 simply adds 1 to each member of the list.
Lobjects automatically convert number types according to the type of input. Math between integers and floats results in floats.

That said, the math Lobjects are:
Ladd -- adds two lists, member by member.
Lsub -- subtracts the stored list from the input list.
Lcomp -- subtracts the input list from the stored list.
Lmult -- multiplies the lists.
Ldiv -- divides the input list by the stored list.
Lrem -- divides the input list by the stored list and outputs the remainders.
Linvert -- divides the stored list by the input list.
Labs -- provides the absolute value of the items in the list.
Lpow -- raises each member of the input list to the power in the stored list.
Lmax -- reports the higher value of the two.
Lmin -- reports the lower value of the two.
Lmean -- finds the mean value between the two.

Lround rounds off the contents of lists. It follows accounting rules: values ending in 5 are rounded to an even digit. The precision may be set by an argument. Thus youu can round off to 2 or 3 decimal places or even powers of 10.

Laccum keeps a running member by member total of input lists. If the list arrives in the right, there is no input. If in the left, the current totals are output. Numbers in either inlet are added to all members of the list. Arguments can set the initial values of the list.
Bang will produce output at any time.
Flush will produce output and set the internal list to 0.
Clear sets list to one element of 0.
Restore sets list to arguments.

Llimit works like Laccum, but the values of the accumulated list are clipped to lower and upper limits set by arguments. If the upper limit is 1.0, 05.+ 0.5+0.5 will result in a value of 1.0 and -0.1 will change that to 0.9
Addto n x adds x to the value at index n.

Lsum adds up the contents of a list. This function is now found in zl sum.

Lxor calculates the exclusive or of all items in a list. This is often used in checksums.

Lave returns the average value of the items in the list. If ther are no arguments, members equaling 0 are not included in this average. If there is an argument of 0, all members of the list are averaged.

Lexpr performs expr type operations on the members of a list. It does this by working along the list member by member and reports the results in a list. To do this, the variables in the expression are:
in, fn = current member of the list.
in-1, fn-1 = previous member, back to in-3.
iy-1,fy-1 = previous result, back to iy-4.
iC, fC = constants received in right inlet.
When typing these, leave no spaces within the variable, but always leave a space before parentheses. If you are using previous members or results, you can apply a seed list to set up the history. Otherwise, there will be no results until the process has moved far enough up the list. The seed message can set the history directly. Once a list has been calculated, bangs will produce further results based on extrapolated inputs-- this if you process a list of 1 2 3 4, a bang will continue the process based on 5. Figure 21 shows Lexpr calculating the logistic map.

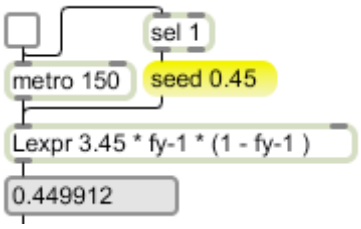


Figure 22. Lexpr generating chaos

Comparing Lists

Many Lobjects compare two lists and react in various ways.

Lsame reports 1 if all members of two lists are the same type and value, 0 if not.

L== reports how well one list matches another. If all are the same, the output is 1. Otherwise, the output is the number of matches divided by the length of the lists.

Lchange compares lists with previous input. If they are the same, the list is output from the right inlet. If different, output is from the left. There is a parameter named lock. If it is set to1, the most recent input will be retained as the basis for comparisons. Arguments define a comparison list and set lock to 1. When Lchange is locked, the comparison list can be changed by input at the right inlet.

Ltest compares lists using comparison operators specified in the arguments. The output list contains the results of the comparisons as 1 or 0. These lists are usually further processed by the objects described under logic with lists.

Like compares the beginning of a list with arguments or a list received in the right. If there is a match, the list is passed out the left outlet. If they do not match, the list is passed out the right. Symbols in the test list provide wild cards. Like treats equal values of different types as equal. The output list is not modified.

Lpos will search for a short list within a longer one and report the position of the first member. If the list occurs more than once, all matching positions are reported in a list. If there is no match, the list is output from the right. (Lpos replaces an older version called Lmatch.)

Lsearch will search for the occurrence of a value or list within a longer list. If it is found, the position of all occurrences is reported at the right and a 1 if sent out the left outlet. If it is not found, a 0 is sent from the left.

Lclose will search a stored list for values closest to the input. The values actually found are output from the right, their positions from the left. If the stored values are 15 10 12 7 1 4, an input of 5 will return a value of 4 with an index of 5. If the input is a list, the outputs will be lists.

Analyzing Lists

These object give information about lists:

Ltop reports the highest values in a list. This is output from the right outlet. The left outlet reports the index of the high positions. The number of values to report is set by an argument or a number in the right.

Lbot reports the lowest values in a list. This is output from the right outlet. The left outlet reports the index of the low positions. The number of values to report is set by an argument or a number in the right.

Lhigh converts all but the highest values in a list to 0. The number of values to leave in the list is set by an argument or number in the right. If the number is larger than the length of the list, all non zero values are converted to 1.

Ledge finds transitions in a list. The default behavior is to report transitions from 0 (space) to 1 (mark). The transition to report is specified as arguments; first is space, second is mark. If both arguments are the same the value defines the mark and transitions either way are reported, with the assumption that the values before and after the list are spaces.

Lpast will output a bang if the input list contains values that cross a target value. Once it has done this, it will not bang again until a value below a reset point has been received. If the target and reset are the same, the bang will occur when the data crosses the target in either direction. This was originally intended to process individual numbers, but it is useful for lists too.

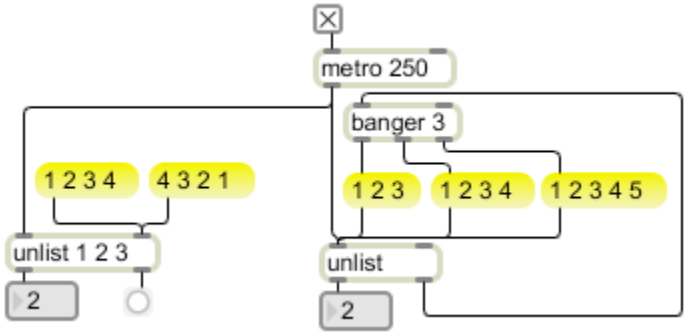
Lmost reports the most common item in a list with duplicate items.. The right outlet reports the number of occurrences.

Lsign reports the sign of all values in a list; -1 for negative, 1 for positive and 0 for 0.

Llength reports the length of a list. This function is now found in zl len.

Disassembling Lists

Unlist stores a list and reports its elements one at a time. A list in the right inlet is stored, a list in the left is stored and the first element is sent immediately. A bang input causes output of the next member. When the end of the list is reached, there are two options. If unlist has arguments, the cycle will repeat with no break. The right outlet will bang as the list starts over. If unlist has no arguments, there will be no output until a list arrives. When the list is used up, it is not restarted-- the right outlet bangs to request a new list. When that list is supplied, the first member is emitted immediately, so if the bang does fetch a list, there will be no perceptible pause in the output.



Figfure 23.

Lpair iters two lists in alternation. A value from the stored list is sent out the right, followed by a value from the input list, at the left then the next from the stored list out the right and so on. The primary intention was to enable arbitrary calculations between two lists-- the right and left outlets connected to expr and the results captured with thresh. Vexpr has made this procedure obsolete, but there are other uses, such as pairing coordinates for jit.lcd.

Logic Operations

Lobjects can perform Boolean logic on ordered lists member by member. Ordered lists are lists of fixed length containing 0s and 1s. I use them to represent pitch structures: {1 0 0 0 1 0 0 1 0 0 0 0} represents a C major chord. When harmonies are represented this way, many complex operations become simple. For instance, transposition is handled by the Lror object.

Ltoset will convert a list of numbers to a list of defined length with the input indices set to 1. Ltop will perform the opposite conversion if it has an argument bigger than the length of the input list.

Lbuildset will construct a set by accumulating indices. A bang outputs the set.

Ltrue will output a bang if a specified position in a list is non-zero. Other wise the input is passed from the right. This can work with a stored list and an input integer, or a stored integer and a list input.

L! or **Lnot** converts 0s to 1 and 1s to 0.

Union and intersection can be done with Lmax and Lmin.

Fuzzy Logic

Fuzzy logic works on lists of the type 0 0 0.1 0.2 0.3 where the values indicate membership in some set for each index. The following perform operations fundamental to fuzzy logic:

Lfind will search a list for a specific value and return an interpolated location. This can work two ways. If a list is entered as arguments or applied to the right inlet, a number in the left will be found. If the argument or right input is a single number, a list applied to the left inlet will be searched. The interpolated location is a fraction. If the search list has the value 7 at index 3, followed by 9, a search for 8 will return 3.5.

Linterp will report the interpolated value between two members of a list. If the search list has the value 7 at index 3, followed by 9, a asking for the value at 3.5 will return 8.

Linfer behaves like linterp, except locations with the value of 0 that are between other values will be assumed to have values interpolated from their positions. So a list with values 0 1 0 0 0 5 will be treated like a list of 0 1 2 3 4 5. It's just a short cut.

Lcent finds the centroid or "center of gravity" of a list.

Lhigh, Ltop, and **Lbot** are essential fuzzy operations, as are Lmin and Lmax.

Fuzzy logic is covered in detail in several of my tutorials.