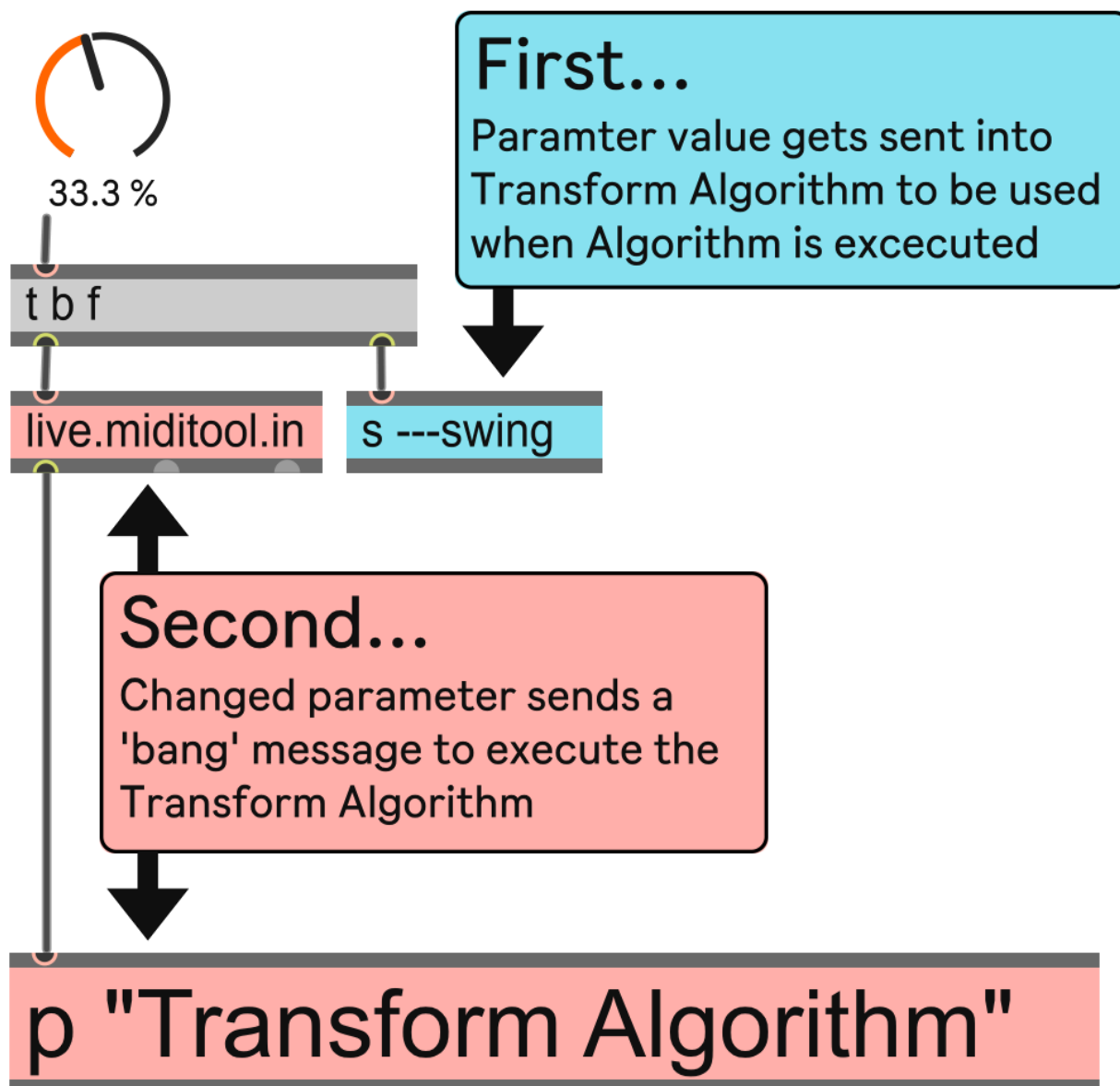


# Building a Midi Transformer

## STEP 1

Interface parameters get sent to prepare the transform algorithm



# STEP 2

## Interface parameters trigger the retrieval of note data from Live

The transform algorithm gets executed by sending a “bang” message into the [live.miditool.in](#) object, which then outputs an **array** of note data from live.

An **array** is a collection of **elements**, each identified by an **index**. In this case, each element in the array contains information about a MIDI note inside of the clip. If you were to select 8 notes to transform, the [live.miditool.in](#) object would output an array with 8 elements... 1 for each note.

<b>Element</b>	note 1 data	note 2 data	note 3 data	note 4 data	note 5 data	note 6 data	note 7 data	note 8 data
<b>Index</b>	0	1	2	3	4	5	6	7

Each element of the array contains the following information...

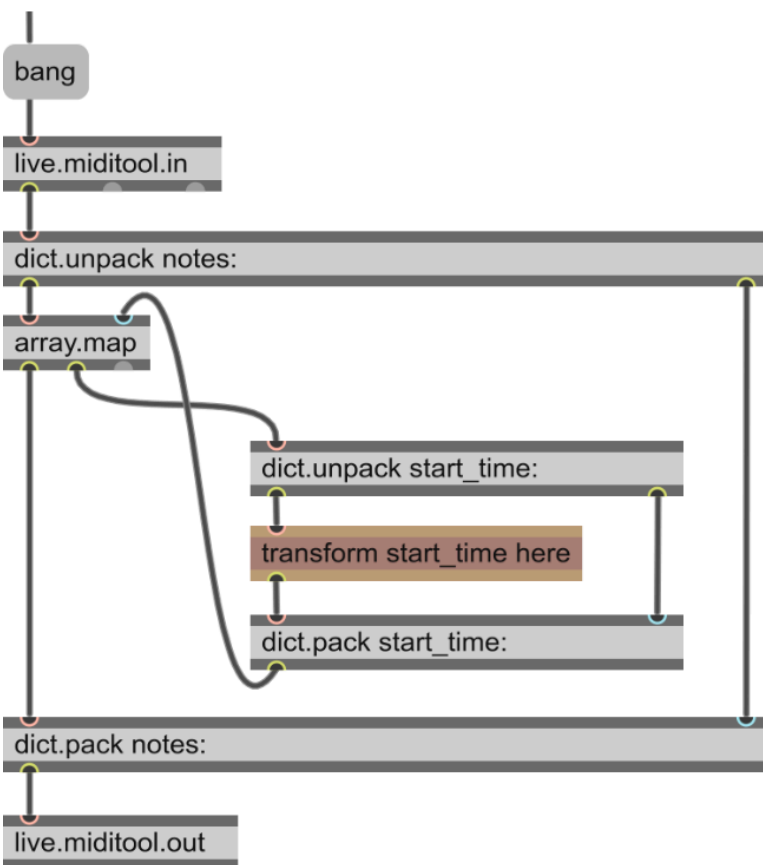
note_id:	The first note in the clip = 1. The second = 2... This is related to the clip, not the selected notes
pitch:	Pitch of the note (0-127)
start_time:	Where the note is positioned in the clip. The end of the first 1/4 note = 1. The Second = 2...
duration:	The length of the note. So end-time = start_time + duration...
velocity:	The velocity of the note (0-127)
mute:	If the note is disabled in the editor 1, otherwise 0
probability:	The note's probability parameter value
velocity_deviation:	The note's velocity deviation parameter value
release_velocity:	The release velocity of the note (0-127)

# STEP 3

## Retrieved note data gets unpacked and transformed

To access and modify this information, we use a series of `dict.unpack` and array objects. The image below shows the basic blocks required to extract and modify note data from Live.

### Basic Blocks



### Dealing with Dictionaries

#### STEP 1

#### Extract note array from dictionary

The note data is stored in an array within a data structure called a dictionary. To access this array, we use a `[dict.unpack]` object with the 'notes:' argument.

#### STEP 2

#### Extract note specific values from each element in the note array

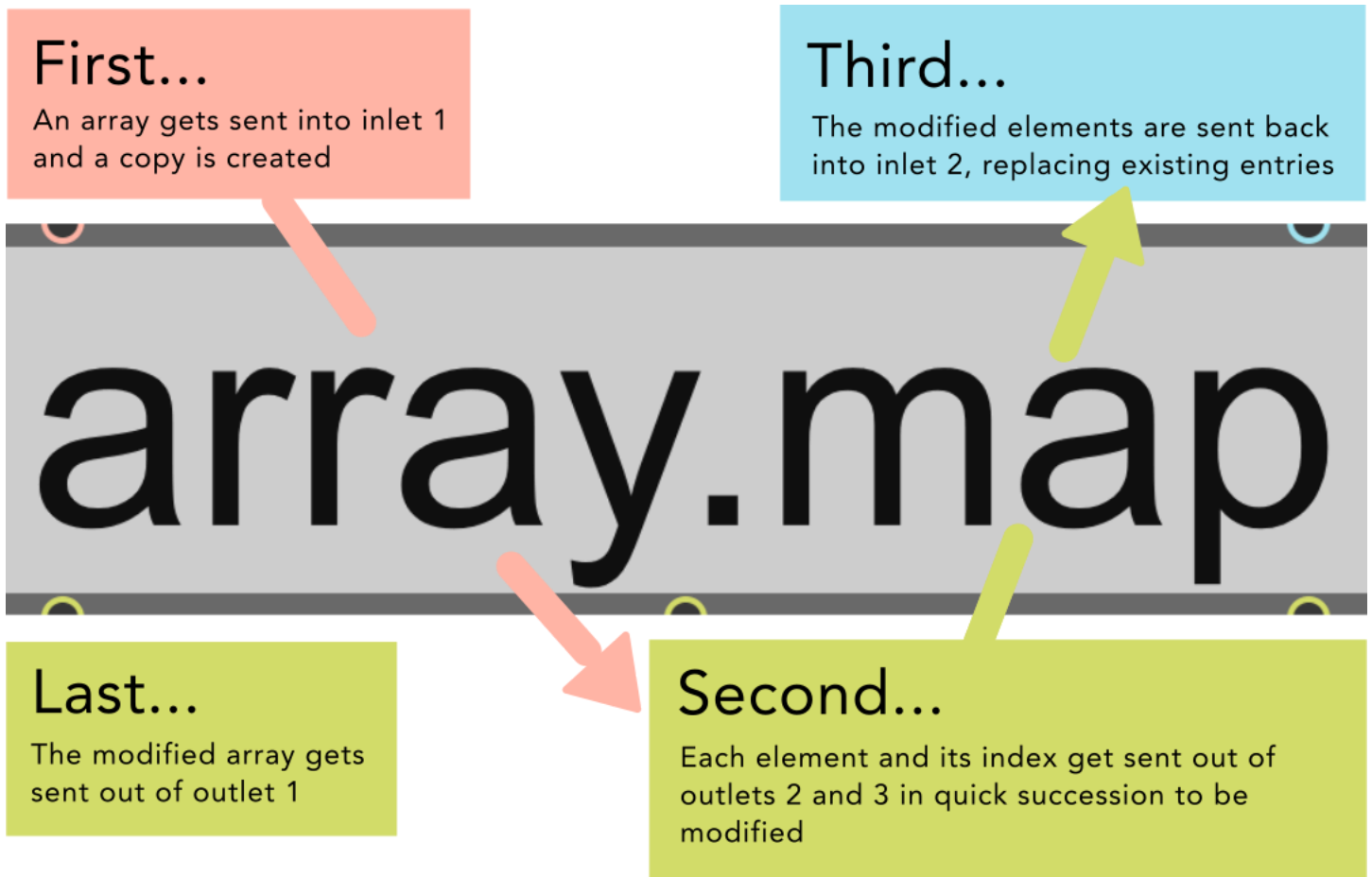
In this example, we extract and modify the `start_times` for each element in the note array. This gives us a floating point value that we alter.

#### STEP 3

#### When finished, repack everything

The data needs to be packed back into a dictionary before being sent out to Live. Notice how the second outlet of each `[dict.unpack]` object is connected to the corresponding inlet of the `[dict.pack]` object. This ensures that any data not included in the specific entries we're editing gets passed through unchanged.

The `array.map` object lets us iterate through and modify elements of an array. The image below shows what happens when we send an array into an `array.map` object.



# EXAMPLE PATCH

Here's the Swing Quantization Algorithm from our free Midi Transformer, Swing

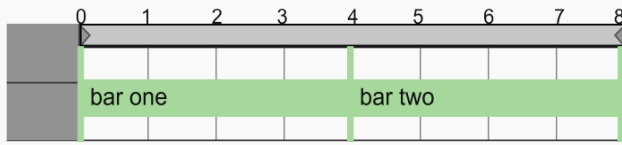
# [Transform Note Array] patcher from Swing.amxd

This code segment extracts note 'start\_times' from arrays of MIDI Note data.

To understand this process, it's helpful to know how these 'start\_times' are represented within the arrays.

Each notes "start\_time" is portrayed as a number relative to the the MIDI Clip inwhich the note resides.

The image below depicts one bar in 4/4 time. Each of the numbers along the top represent quarter note periods of time.



## BASIC NOTE LENGTHS

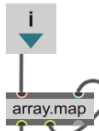
1/4 note = 1.0  
1/8 note = 0.5  
1/16 note = 0.25

So... if a MIDI note has a start\_time of 5, we know that MIDI note starts on the end of the 5th quarter note period

### STEP 1 Normalizing Quarter Notes Periods

We treat each quarter note region as ranging from 0 to 1, regardless of its actual position in the MIDI clip.

note array in from live



### STEP 4 Denormalizing Quarter Notes Periods

After applying quantization we we add the remainder back on.



transformed note array out

### STEP 2 Generate a Swung Grid

### STEP 3A Quantizing MIDI Notes

We compare each MIDI note's starting position to our adjusted grid, and find the closest grid line for each note.

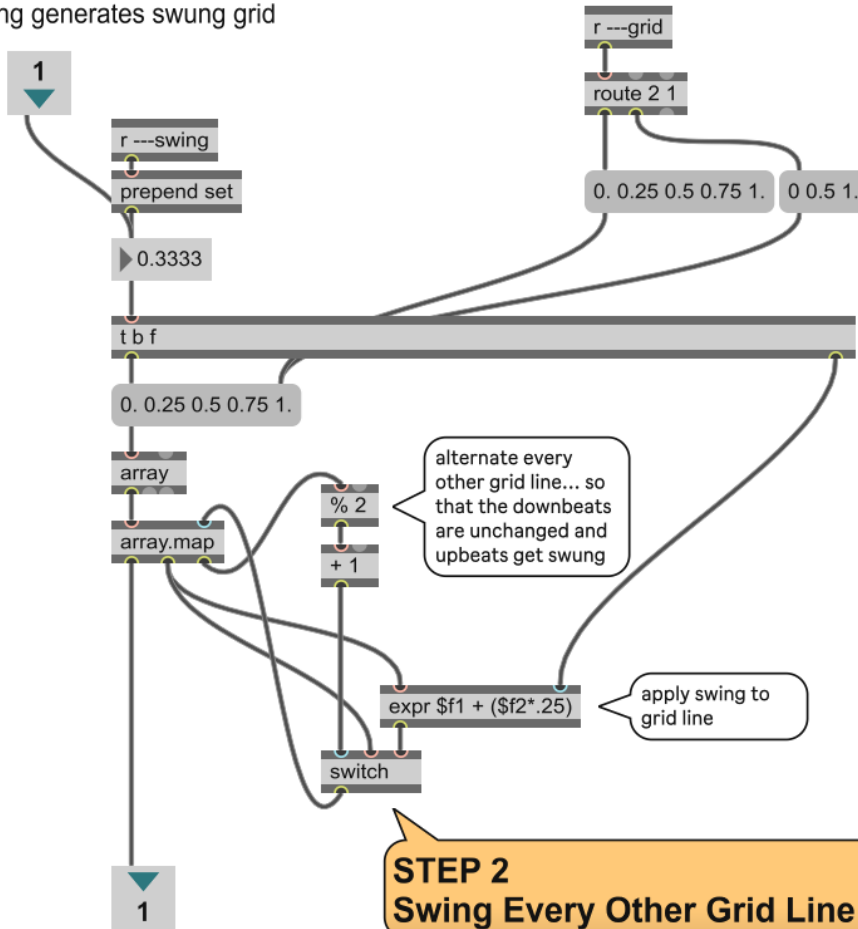
### STEP 3B Quantizing MIDI Notes

Then we move the note towards the nearest grid line based on the Quantize Strength parameter

Abstractions on next page

# [Generate Swing Note Grid]

bang generates swung grid



## STEP 1 Create a Straight Note Grid

First, we divide a quarter note into either sixteenth or eighth notes.

0, 0.25, 0.5, 0.75 <- sixteenth notes  
0, 0.5, <- eighth notes

Then, we add a '1' to end of each list so that notes close to the end of the quarter period can be quantized to the start of the next period

0, 0.25, 0.5, 0.75, 1 <- sixteenth notes  
0, 0.5, 1. <- eighth notes

## TIP

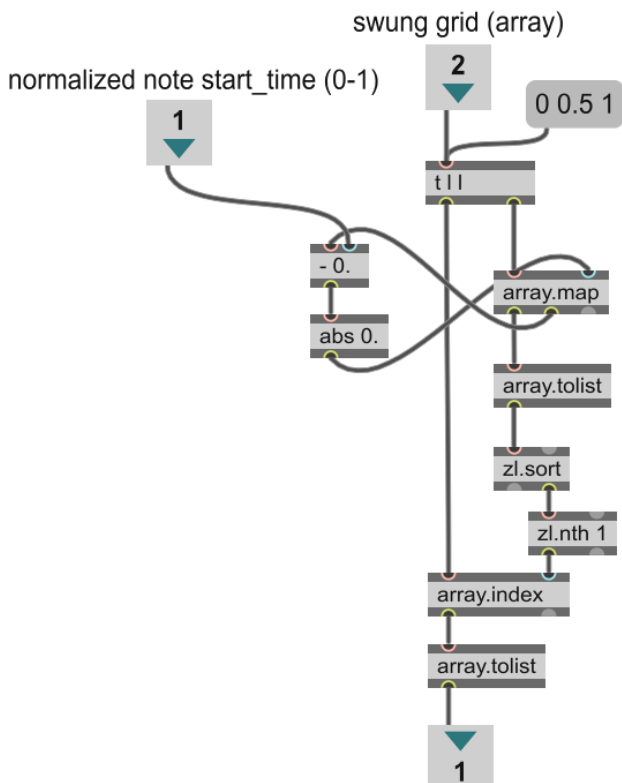
To observe the contents of an array, try connecting the outlet of a [array.to-list] object to a message box!

array.to-list

## STEP 2 Swing Every Other Grid Line

output swung grid (array)

# [Compare start\_position to adjusted grid and find the closest gridline]



## STEP 1 Create A Distance Array

We compare each element of the array containing our swung grid to each note's start\_time.

Using that information, we calculate the distance between the note\_start position and each element of the swung grid array, replacing said element with the calculated distance value

## STEP 2 Find the Shortest Distance

Next we determine which index of the array contains the shortest distance. We use that information to refer back to the original array containing the unmodified swung grid, outputting the corresponding grid line position

nearest grid line (normalized 0 -1)

# Want to learn more?

Dig into Swing... Our free Midi Transformer used in this lesson → [windmakeswaves.com/swing](https://windmakeswaves.com/swing)

Cycling74 also has a great lesson on Midi tools here...

[https://docs.cycling74.com/max8/vignettes/live\\_miditools?q=array](https://docs.cycling74.com/max8/vignettes/live_miditools?q=array)

