Documentation of the Sonification Desktop

R. Michael Winters
Input Devices and Music Interaction Laboratory (IDMIL)

December 19, 2011

Abstract

The purpose of this paper is to provide documentation to the Sonification Desktop, a software tool for sonification of large databases of gesture data in music performance. A paper presenting the tool is currently under review [1], and a literature review of the field of sonification of gesture is presented in [2]. The paper will present the tool as it is meant to work as opposed to the current implementation. This is because as of December 19, 2011 the tool is not fully operational, mostly due to its complexity and changes made to Max/MSP since 2009.

1 Background

The "Sonification Desktop" was created by Alexandre Savard for his Master's thesis at McGill University in 2009 [3]. The primary contribution of the project was an interface to make the analysis of large databases of gesture an easier task for researchers in the field. A secondary contribution was the use of Principle-Component Analysis (PCA) for analysis of the gesture features. The use of PCA was challenged by [4] who showed that subjects could more easily identify gesture features using a velocity based mapping.[1]

The motivation for creating a tool for sonification of large databases of gesture in music performance is presented in [1]. But briefly,

1. Sonification offers the possibility of exploring large databases efficiently, potentially uncovering data relations that might be obscured through visual-only methods.
2. Presenting data in sound allows a method of data analysis in the same medium as the performance audio.
3. Sonification can be used to communicate aspects of visual performance to the blind (or those that cannot see).

[1] It should be noted that velocity-based mapping to amplitude is a feature of our tool. The PCA is challenging because it sometimes does not have an obvious visual meaning

Savard left his thesis project shortly after completion and no documentation was provided outside of what was published in his thesis. Understanding how this tool works will be crucial to the development of a better tool in the future. Beyond gesture in music performance, the ability to analyze music related-information in the same medium as the underlying audio may be lead to more meaningful sonifications.

A short overview of the system is provided in section 3 of [1].

2 Overview

The primary user interface is presented in Fig. 1. Starting from the upper left, 'Load Subject' allows you to choose your subject, condition, and reference subject. The condition refers to the way they were asked to play in (i.e. immobile, expressive, etc.) during performance. The reference (Refer.) allows you to chose a subject upon which to base all time-warping. The choice for this reference subject will stay fixed until you change it. Without setting this, no time-warping can be done.

The 'Synthesis Mapping/Data Preprocessing' section allows you to chose the synthesis mapping and preprocessing steps that you wish to use. For the mapping, you have the choice of {Audification, Cycle, FMSynth, BeatingSine, RissetSynth, Subtractive}. For the preprocessing, you have the choice of {Raw Data, Euler Distance, Euler Angle, Circular Movement, Body Curvature}, and for channels 6-10, you have the additional options of {PCAhead, PCALBody, PCARBody, PCALLeg, PCA RLeg} respectively. When you choose a preprocessing or a synthesis, you can access the corresponding sub-interface by clicking the *button* located to the right of the *umenu* object. Clicking on the 'Ch#' button applies your choices, and must be clicked before choosing input data.

Having clicked the 'Ch#' corresponding to your preprocessing and synthesis choices, you can now move on to pick your input data. The input data is the data that will be preprocessed according to your preprocessing and synthesis choices for that channel.

The user has to choose signals according to the allowed inputs for a each preprocessing option. For example, {Raw Data} can only accept one input, while {Euler Angle} requires nine (three dimensions for three markers). They must also be chosen from the right place. To list: {Raw data} requires one input signal (first column, first row), {Euler Distance} requires two markers with three coordinates X, Y, Z (first two rows, all three columns), {Euler Angle} requires three markers (all three rows, all three columns), {Body Curvature} requires 3 markers with two orientations (all three rows, first and second columns), and {Circular Movement} requires two inputs (two coordinates for a single marker). There is also a preprocessing step for weight transfer, but it isn't being implemented currently.

The 'Play/Record/Calibration' section sends the trigger to start the sonification, start video, and calibrate the system according to the reference condition as chosen in the 'Load Subject' section. The timer will start when the sonifica-

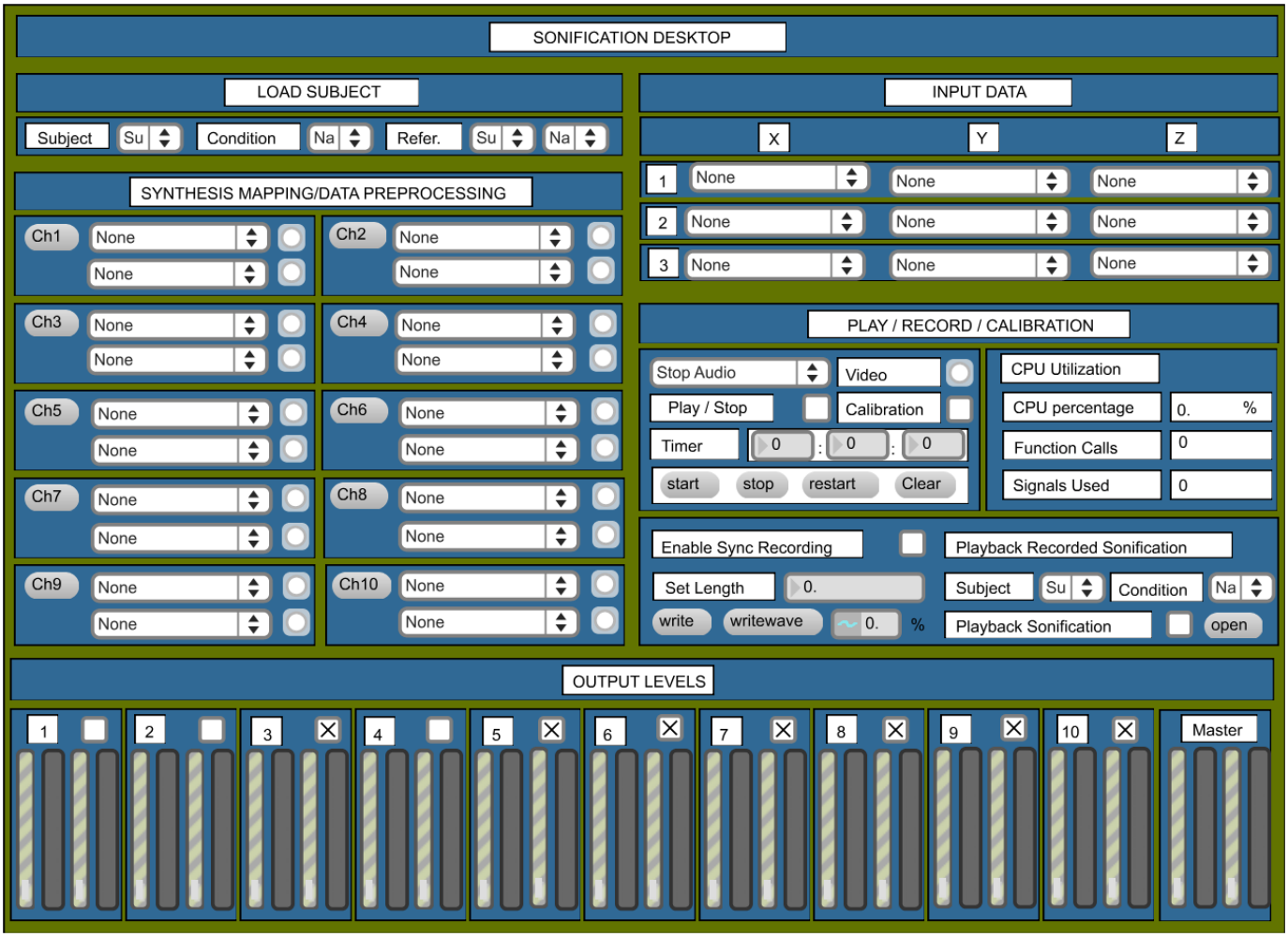


Figure 1 – A screenshot of the Sonification Desktop. Load Subject allows you to choose your subject for sonification, the condition they played in, and do time-warping synchronize with a reference subject. The Synthesis Mapping/Data Preprocessing section is where you choose what preprocessing step and synthesis mapping for each of ten channels. Once the preprocessing and mapping is chosen, click the Ch# (e.g. Ch1) to register your selection. From there, you pick the input data you will be using. The number of input parameters varies from 1-9 as discussed in section 2. Once you have made your selection, you can turn on the audio and press play. The output levels allow global control of loudness like a mixing board.

tion starts, but can be chosen to restart at any desired location. The recording section will be detailed later.

3 Structure

The purpose of this section is to display the inner structure of the desktop. The structure can be broken into three large categories, data selection, preprocessing, and synthesis. The PCA and the time-warping are key features of the system that will be presented in their own subsections see section 5.1 and 5.2.

Figure 2 displays the sonification desktop Fig. 1 unlocked. The SonificationCore~ Fig. 3 represents the finalized signals sent to be played through 10 different channels, each with volume control. The signals from the SonificationCore come from the ChannelPoly Fig. 6, which contain subpatches for loading data, preprocessing and synthesis. SendMarkers and SendInterface are both selector patches, patches whose purpose is to allow the user to select input data, preprocessing, and synthesis, but do not do that within themselves. Explanations in this section will now flow within the figure captions.

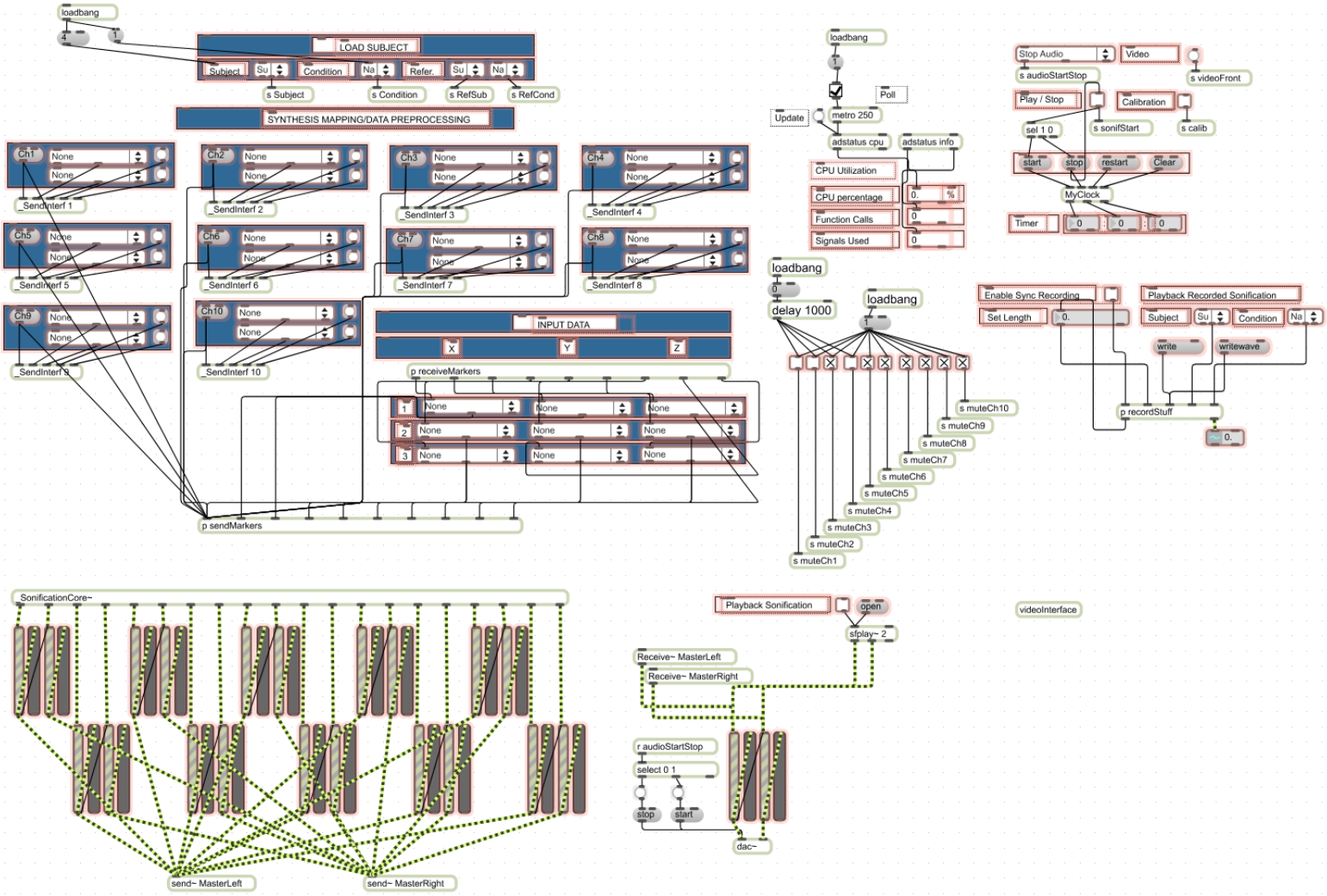


Figure 2 – The desktop Fig. 1 unlocked. Load subject strings (i.e. Su1) are sent out as messages (s Subject). In the Synthesis Mapping/Data Preprocessing, the options are sent to the sendInterface Fig. 4 for each channel and clicking the channel number (e.g. Ch1) sends a message to the sendMarkers subpatch Fig. 5. The output of SonificationCore∼ are sent to the 10 different channels for volume control and mute options. SonificationCore∼ represents the outputs after choosing data, preprocessing, and synthesis has been applied. 'SonificationCore∼' is displayed in Fig. 3.

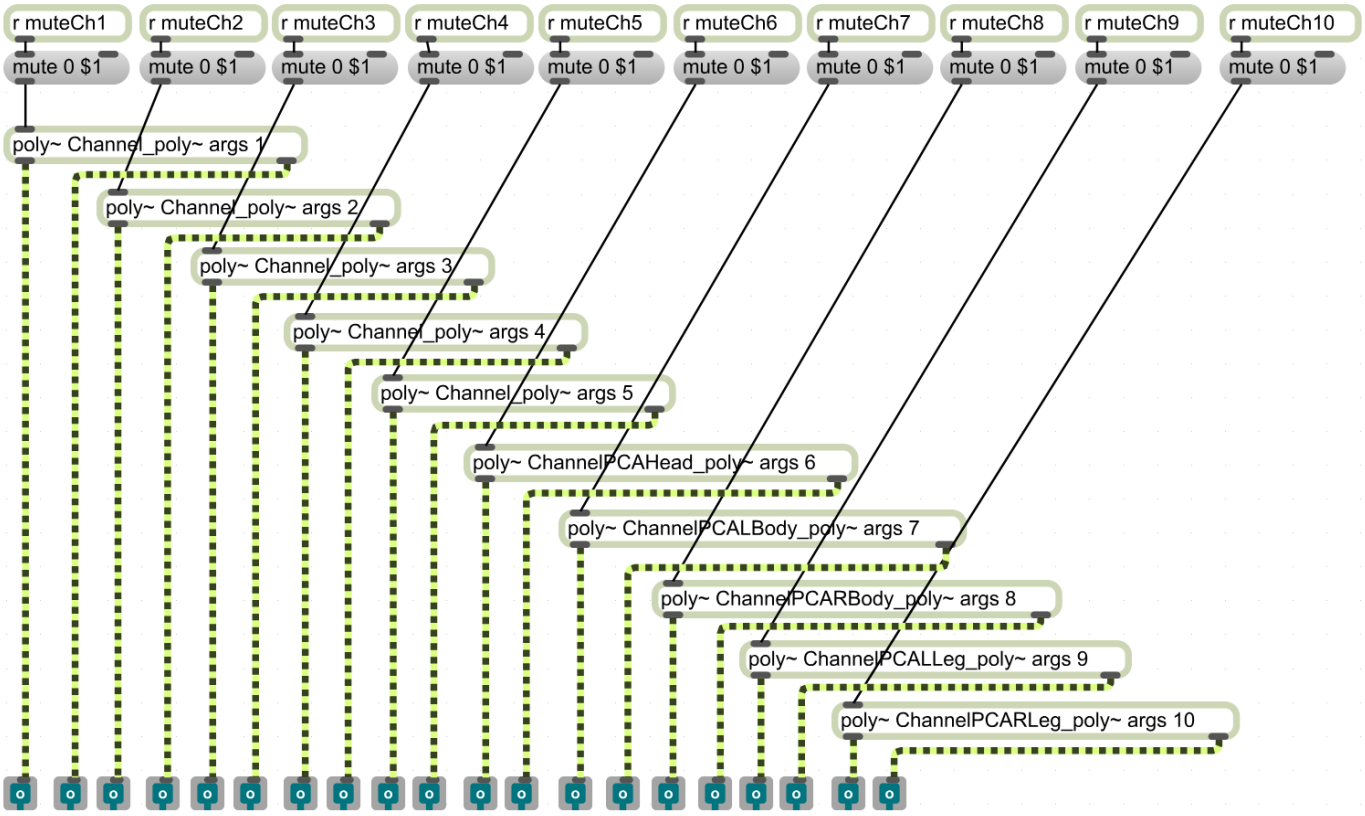


Figure 3 – A screenshot of the 'SonificationCore' subpatch. There are many instances of ChannelPoly Fig. 6 that allow for the standard preprocessing procedures of {Raw Data, Euler Distance, Euler Angle, Circular Movement, Body Curvature}. Channels 6-10 allow an additional preprocessing feature which is the PCA of individual body regions (i.e. The head, left body, right body, left leg, and right leg). The PCA channels (see Fig. 20) are different from the regular Channel_poly patches (Fig. 6). The outputs of this patch are the the left and right inputs from the 'Output Levels' as shown in the bottom left of Fig. 2 and the bottom of the primary user interface Fig. 1. The r muteCh# come from Fig. 2 and mute the channel so that no CPU power is wasted.

Figure 4 – A screenshot of the 'SendInterf' subpatch for Channel 1. The structure is repeated for all channels. The exact connections with the channel are shown in Fig. 2, and the outputs are sent to the ChannelPoly as displayed in Fig. 6. In order, getInfo is actually the channel number sent as a message of the form 'Ch#', SonifType and PrepType are the synthesis mapping and data preprocessing the user has chosen, and OpenInterS and OpenInterP are messages to open the synthesis mapping and preprocessing sub-interfaces.

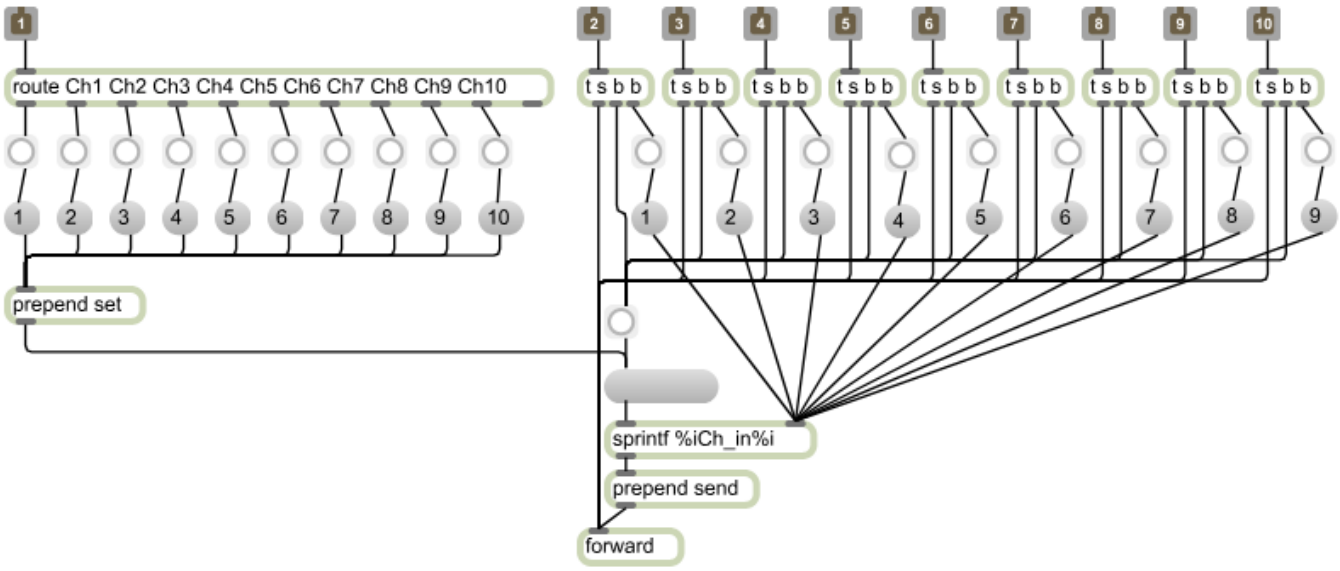


Figure 5 – A screenshot of the SendMarkers subpatch, a patch which sends selected channel numbers and choices for parameter inputs. When the user chooses a channel # (e.g. Ch1), a 'set' is prepended to the channel's # and sent to sprintf where that number is prepended to 'Ch_in'. The values from inlets 2-10 come from the data parameters chosen in the 'Input Data' section. Whenever a a new value from the right inlet comes in, sprintf sends it out as a string. 'send' is prepended to this string (e.g. 1Ch_inLFHD_X), and the string is received by the the ChPlayerPoly patch Fig. 7. Note: the 'prepend set' object may not be necessary.

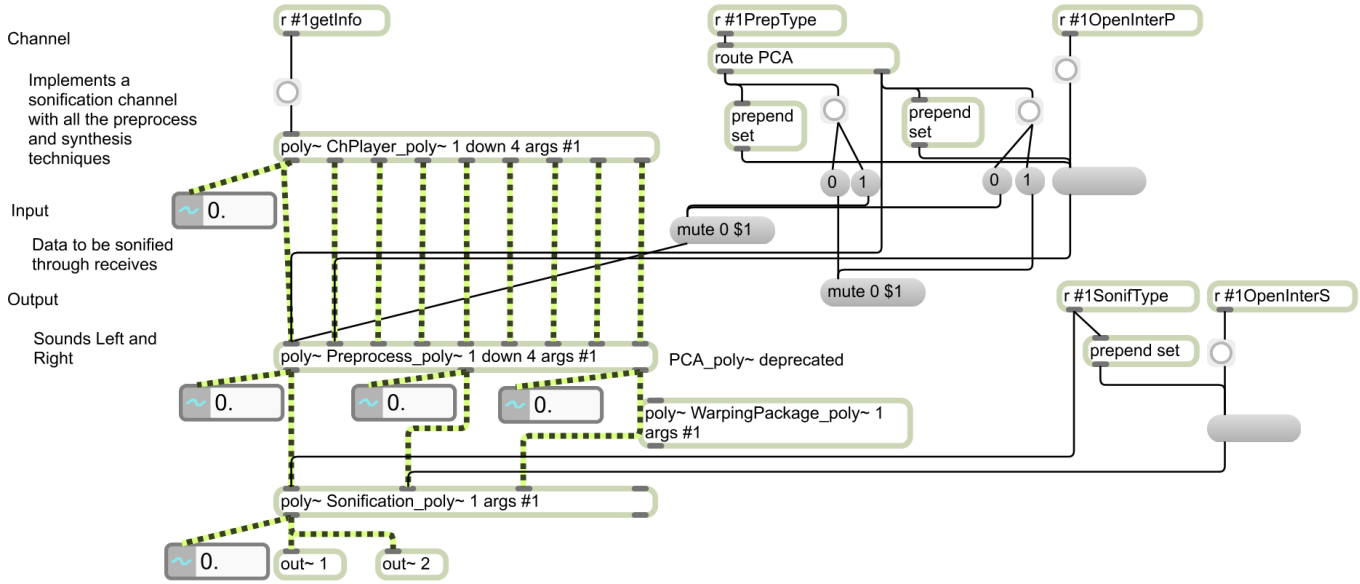


Figure 6 – A screenshot of the ChannelPoly subpatch. It contains the ChPlayerPoly Fig. 7, the PreprocessPoly Fig. 8 and the SonificationPoly Fig. 9. When the PCA and time-warping are included, this patch is a high-level representation of the underlying system. The five inputs of the SendInterf subpatch Fig. 4 are sent here. ChPlayerPoly loads the data from the nine possible channels and sends it when the sonification is running. PreprocessPoly molds these parameters into what is fed to the SonificationPoly . The two outputs of this system are sent to the SonificationCore subpatch (Fig. 3) where they are routed to the correct amplifier.

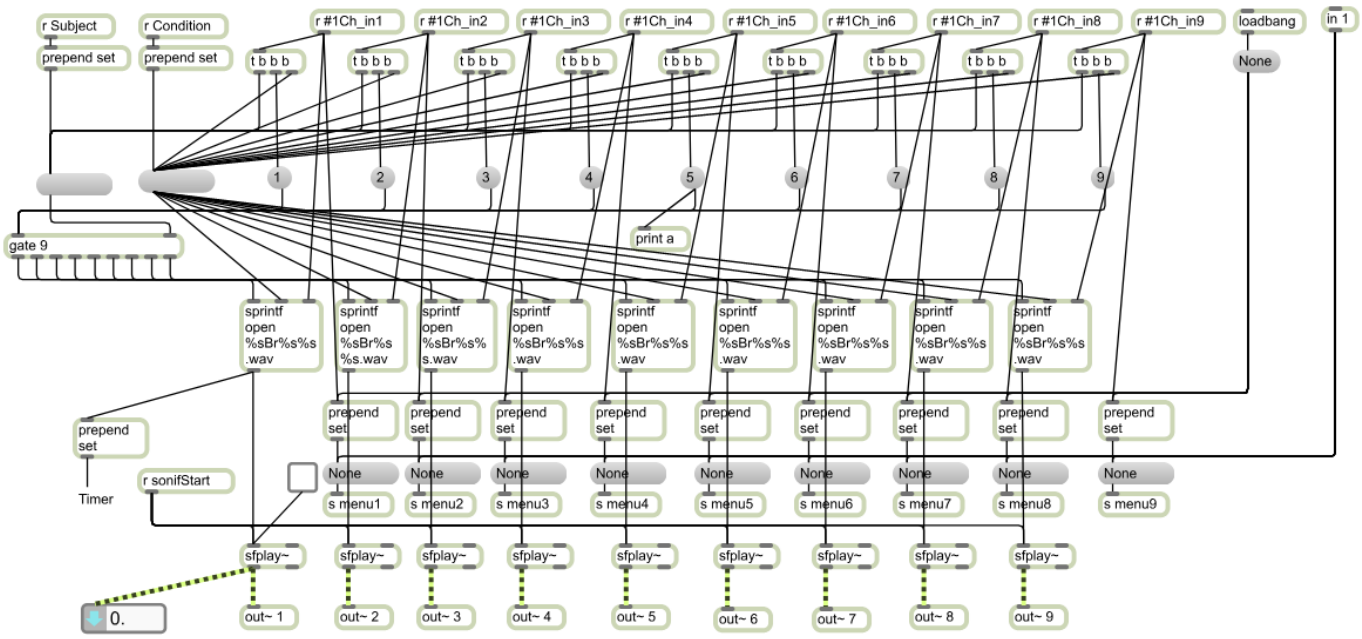


Figure 7 – A screenshot of the ChPlayerPoly subpatch. The purpose of the ChPlayerPoly patch is to read the correct data file. This topic will be provided in section 4. The files it is looking for are in the .wav format with the indices that have been assigned to our data. As presented so far, these indices refer to our subject {Su1, Su2, Su3,...}, our condition {Na1, Na2,...}, and our marker data e.g. {LFHD_X, C7_Y,...}. The outputs of this object are our .wav files. Ordinarily, you will have to modify the Max search path so it can find your files: (Options→ File Preferences→{add new path}.) When you have selected input data, these are opened, but they are not played until you receive the sonifStart trigger which comes from the 'Play' trigger in the GUI (Fig. 1). The menu#' items declare the input of the patch as variables. Previously, they were just messages.

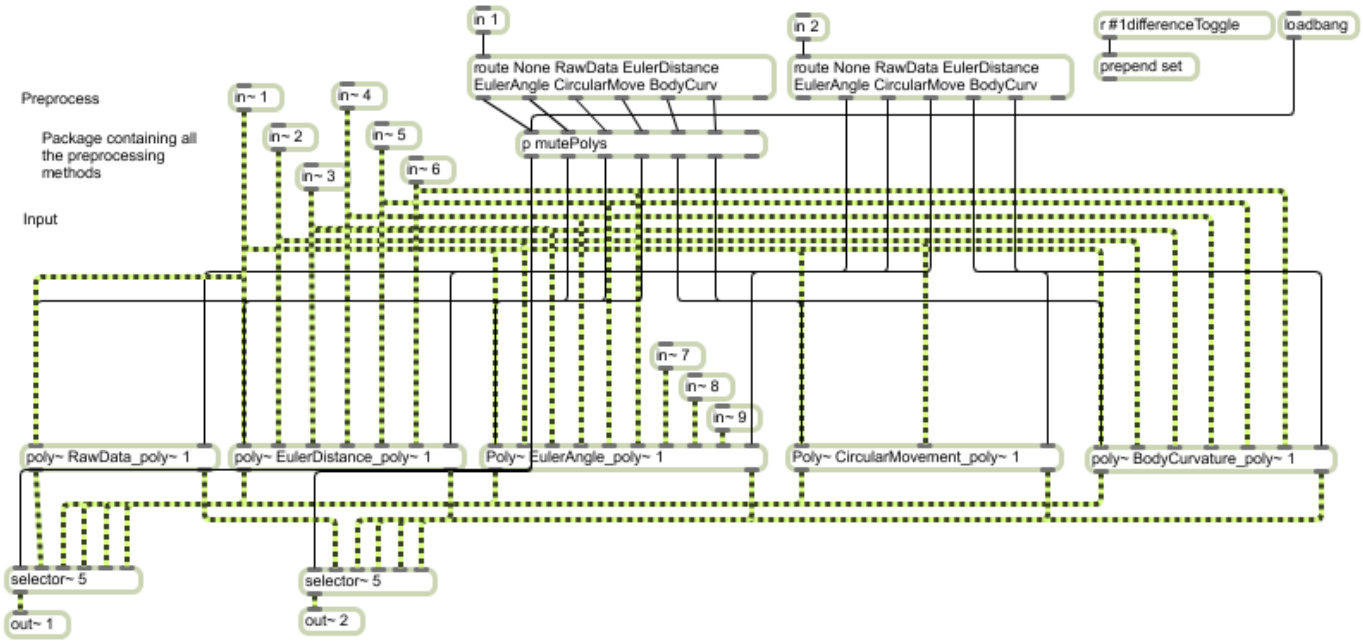


Figure 8 – A screenshot of the PreprocessPoly subpatch. The nine inputs come from the ChPlayerPoly Fig. 7 and are the input data as represented in a .wav file. There are five preprocessing options displayed; the details of each will be presented in section 5. The outputs of each are are routed via the selector object. The two non-signal inputs are essentially just messages that came originally from the SendInterface Fig. 4, but were routed in the ChannelPoly Fig. 6.

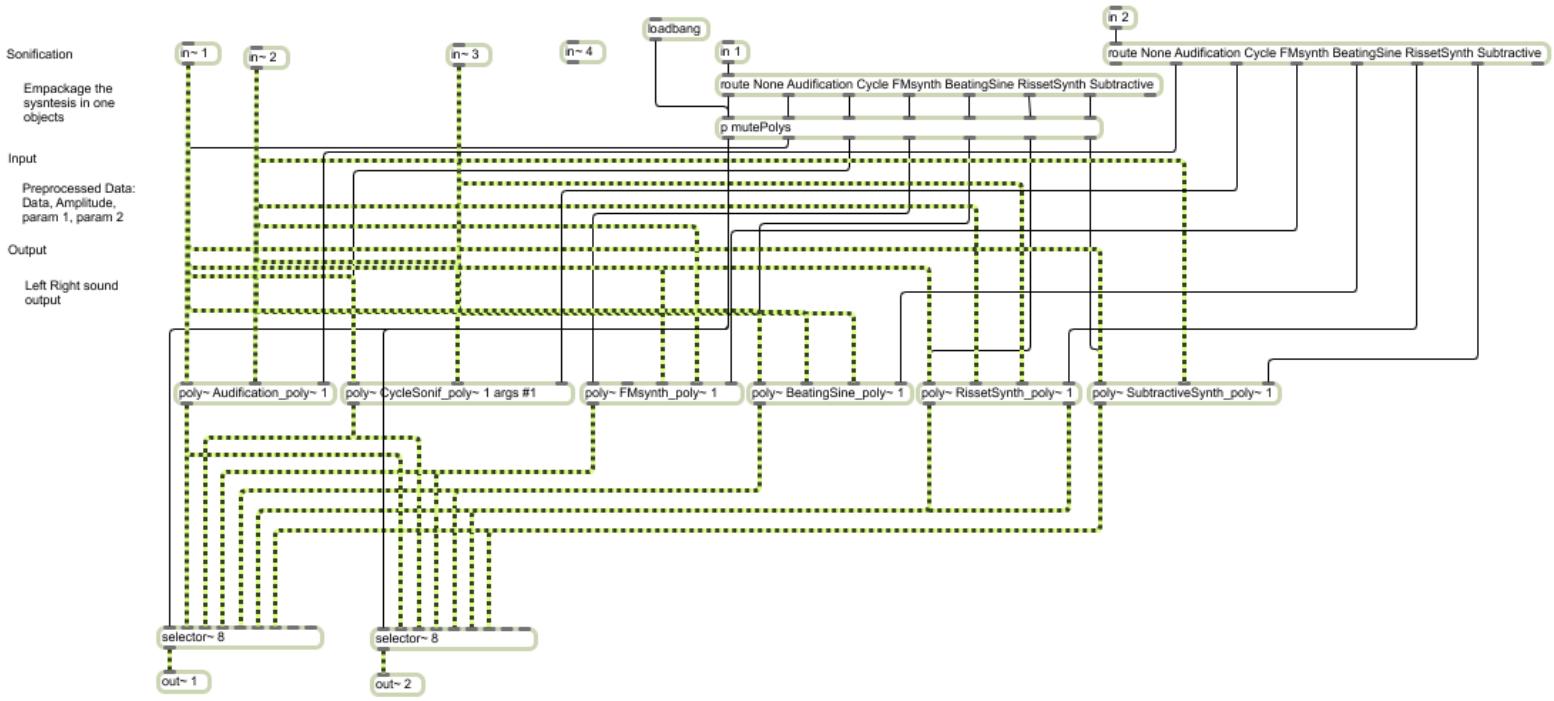


Figure 9 – A screenshot of the SonificationPoly subpatch. Each of the preprocessing options has two outputs that are the instantaneous value normalized between 0 and 1 and the time-derivative of this value expressed as an amplitude. Much like the PreprocessPoly, the SonificationPoly has two non-signal inputs that select the type of synthesis to be done. The implementation of these synthesis steps will be presented in section 6. The output of this patch is the outputs for the left and right speaker of the particular channel that has been chosen by the user.

4 Loading Data

In the subject *umenu* object, there are prefixes of the form {Su1, Su2, Su3, etc..}. These are prefixes used to call data into Max/MSP through the ChPlayerPoly patch displayed in Fig. 7. Currently, you must set the subject and condition before selecting the Channel you want to use for sonification. The goal of the ChPlayerPoly patch is to open the correct file in your directory and prepare to feed it to the chosen preprocessing step.

Currently, the system requires that your files be stored in a .wav file. Often anomalies in gesture data are best dealt with in an external program such as MatLab. As mentioned in [1, 3], a MatLab routine has been created for turning gesture data files into the .wav format. It however does more than this conversion including the generation of "metadata" like max and min values of the .wav files, an important part of the system for time-warping as shown in the .txt files of WarpingPackagePoly Fig. 25.

5 Data Preprocessing

Once you have a gesture feature in mind for synthesis mapping, the user is directed to one of the ten sonification channels displayed in the Data Preprocessing/Synthesis Mapping section of Fig. 1. These channels provide the user with preprocessing features and synthesis mappings. The preprocessing features require between one and nine input data signals depending upon the preprocessing chosen. The exact nature of this variation was discussed briefly in section 2.

The purpose of this section is to present the implementation of the preprocessing features that have been created for the tool. These include {Raw Data, Euler Distance, Euler Angle, Body Curvature, Circular Movement}, and Weight Transfer. Weight transfer is a new option in development. The PCA,and time-warping proceeedures will be presented in section 5.1 and 5.2. As in section 3, the explanations will flow within the captions. The preprocessing steps are held within the PreprocessPoly Fig. 8, which is part of the ChannelPoly Fig. 6.

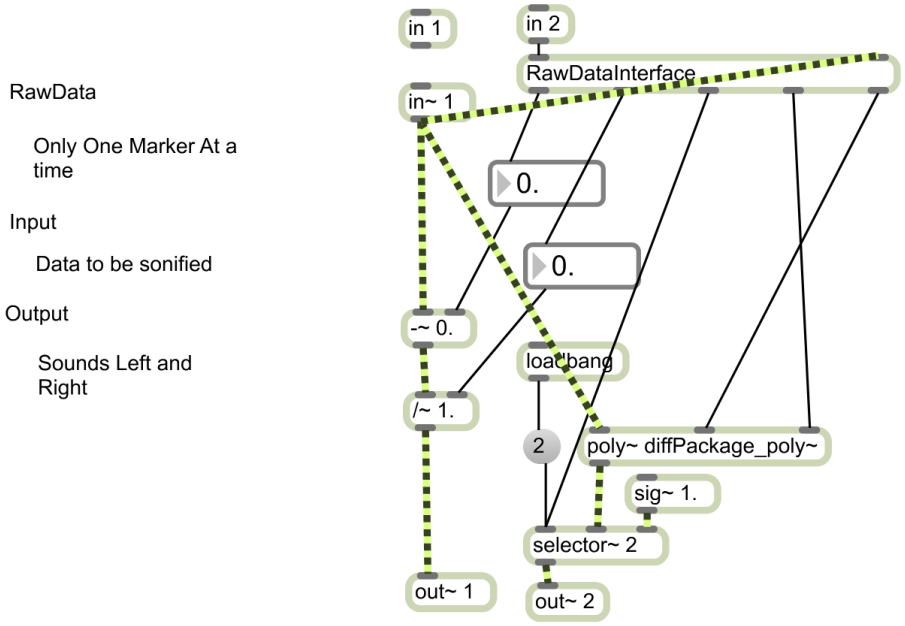


Figure 10 – A screenshot of the RawDataPoly subpatch. It accepts one data stream corresponding to one marker along one axis. It receives messages from the RawDataInterface Fig. 39(a) which is accessed by choosing RawData in the Data Preprocessing section of Fig. 1, and then clicking the button to the right of it. The first two inputs from the RawDataInterface are user defined min and max values. **Note: This should be done automatically in the new implementation.** The remaining three inputs of the RawDataInterface are for velocity mapping. The first selects the velocity mapping, and the next two correspond to the amplitude scaling and the number of samples to take the difference between. The first output is therefore the normalized position between 0 and 1 and the second is the amplitude.

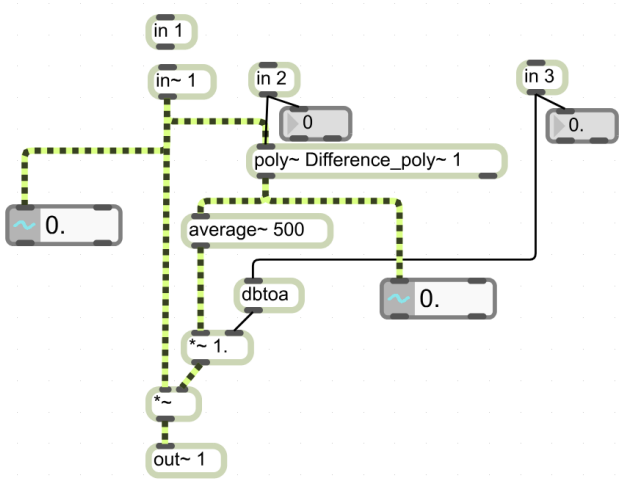


Figure 11 – A screenshot of the DiffPackagePoly subpatch. It is used to turn the instantaneous position value into an amplitude that corresponds to the velocity of that marker. The second inlet is the chosen number of samples that have passed for velocity calculation. The velocity is calculated as the difference between the current sample and a sample in the past that is separated by a number of samples that is user defined.The third input is the user defined dB of the RawDataInterface Fig. 39(a). From this patch, we can see that the amplitude is applied to the signal as opposed to being kept separate. It inherits from the RawDataPoly Fig. 10, and the contained DifferencePoly is displayed in Fig. 12.

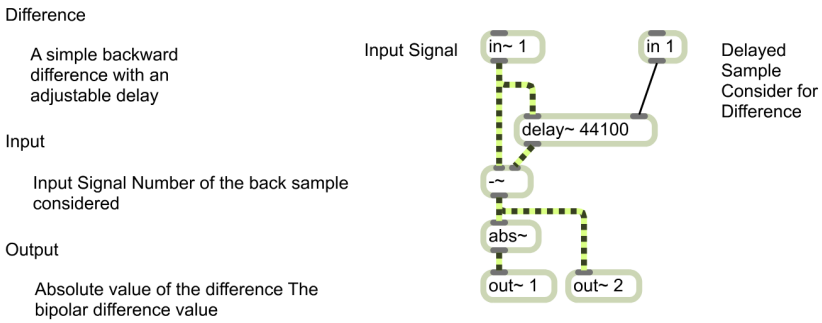


Figure 12 – A screenshot of the DifferencePoly subpatch. It is found inside of the DiffPackagePoly Fig. 11, and the non-signal input is the delayed sample as discussed in the DiffPackagePolyİt is continuing the process of velocity mapping. As we can see in this patch, through the use of abs∼, we are not interested in the velocity direction, only velocity magnitude. However out∼2 is the non-absolute value that is available but not used.

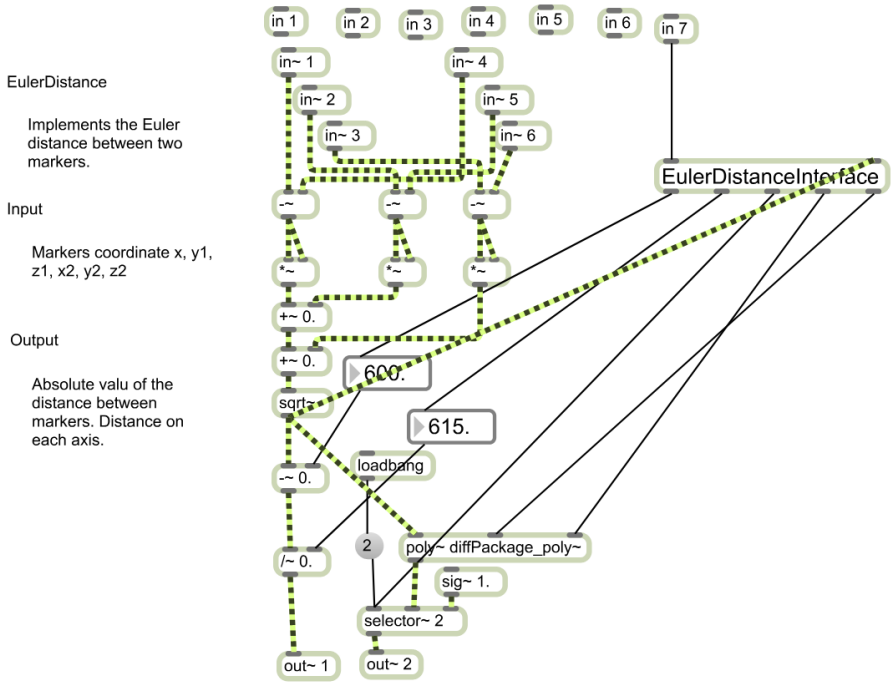


Figure 13 – A screenshot of the EulerDistancePoly subpatch. As we can see, to use it requires six inputs corresponding to two markers with each of their X, Y, Z dimensions. The patch determines the distance between the X, Y, Z values of the two markers, squares them, adds them together and takes the square root. This value is sent to the EulerDistanceInterface Fig 39(b) and the DiffPackagePoly Fig. 11. As with the RawDataPoly the two outputs of this patch are the instantaneous value and a "velocity" amplitude determined by taking the difference with a delayed sample.

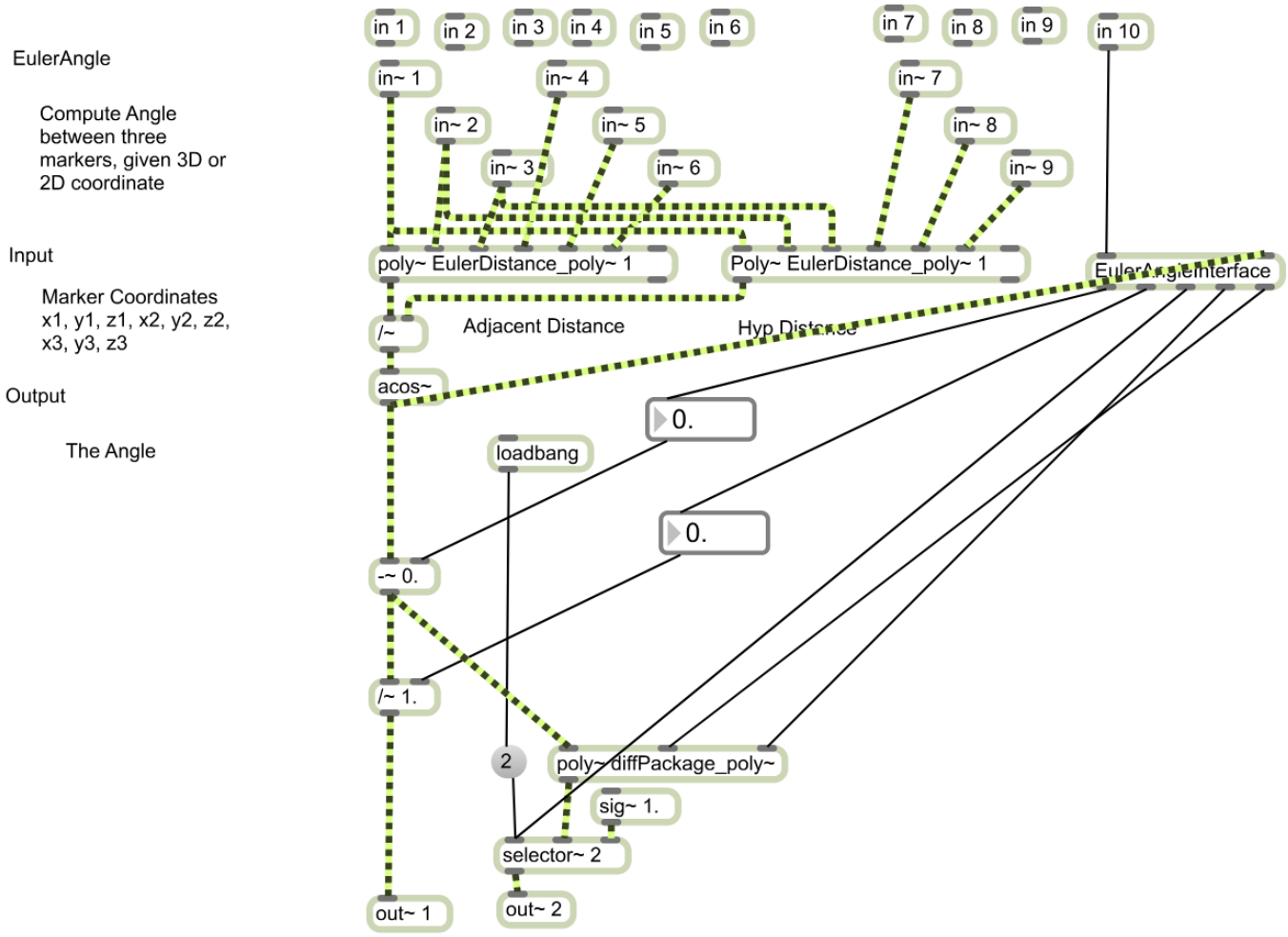


Figure 14 – A screenshot of the EulerAnglePoly subpatch. This patch implements preprocessing based upon the angle between markers. As we can see, it requires nine markers to work correctly, and the first marker is the "middle" marker– the one that defines the vertex of the angle. The EulerDistancePoly was shown in Fig. 13, and the DiffPackagePoly was shown in Fig. 11. From the two EulerDistancePoly, we receive instantaneous distances. I do not know what formula is being used that generates the angle from these two measures. It is worthy to note that the Vicon system generates some angle values automatically, occluding the need for this patch. The EulerAngleInterfaceis shown in Fig. 39(c).

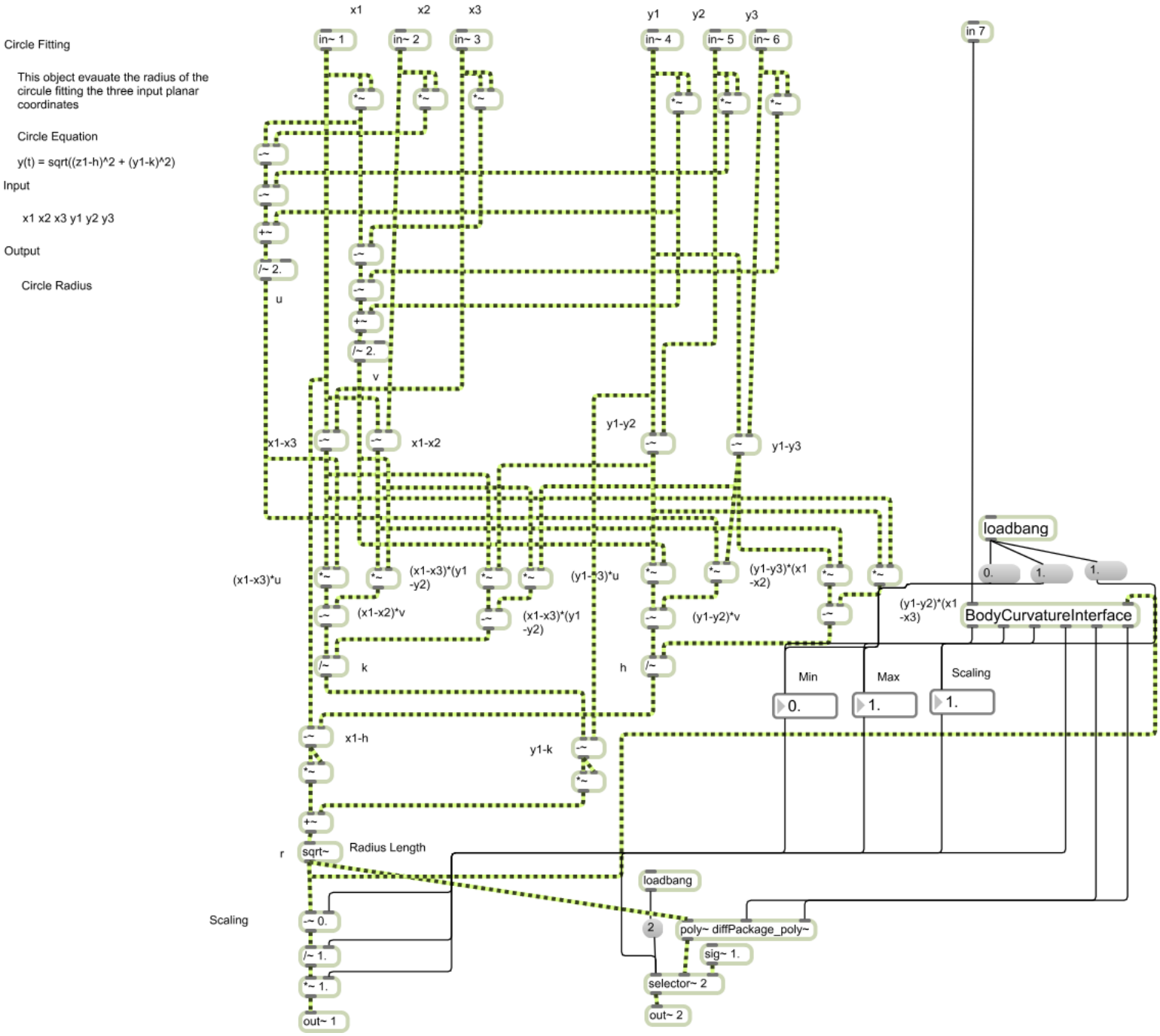


Figure 15 – A screenshot of the BodyCurvaturePoly subpatch. The purpose of this patch is to measure body curvature from two markers in three dimensions. It determines the radius of the circle that would fit the two markers. If they were far apart in one dimension, the radius of that circle would be rather large to fit it. On the contrary, if they were close together, the radius would be small. The BodyCurvatureInterface is shown in Fig. 40(a).

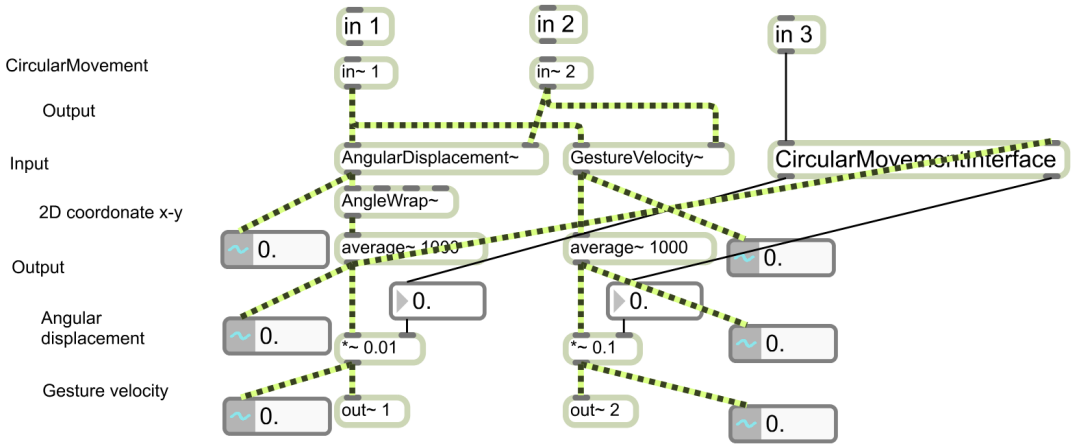


Figure 16 – A screenshot of the CircularMovementPoly subpatch. The inputs to this patch are the (x,y), (y,z), or (x,z) position values for one marker. The goal is to compute the the amount of ciruclar movement these two inputs undergo. The GestureVelocityFig. 17 is a patch which figures out the velocity using change in position between the current and a previous value. The AngularDisplacement subpatch Fig. 18 determines the angular displacement between the movements, and the AngleWrap subpatch Fig. 19 does a simple wrapping.

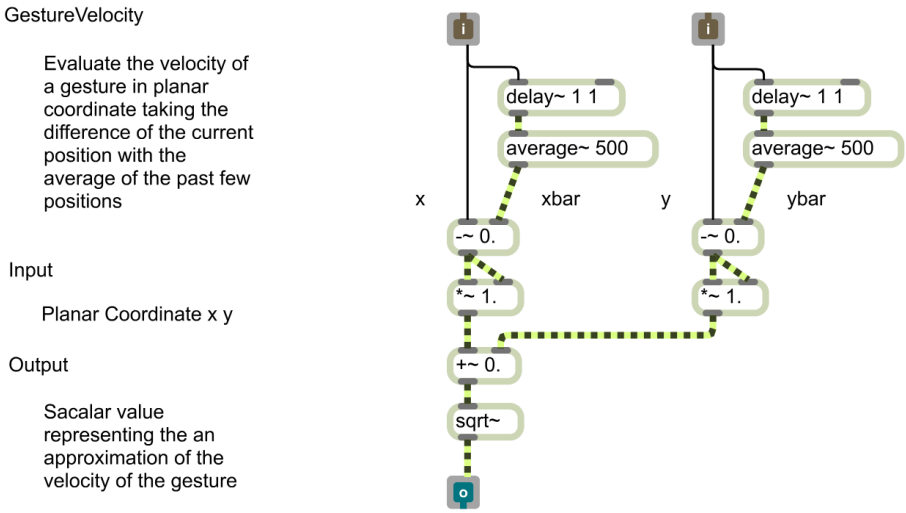


Figure 17 – A screenshot of the GestureVelocity subpatch. It computes the velocity of each marker independently, squares it, adds the two values together, and takes the square root to compute the magnitude of their combined velocity.

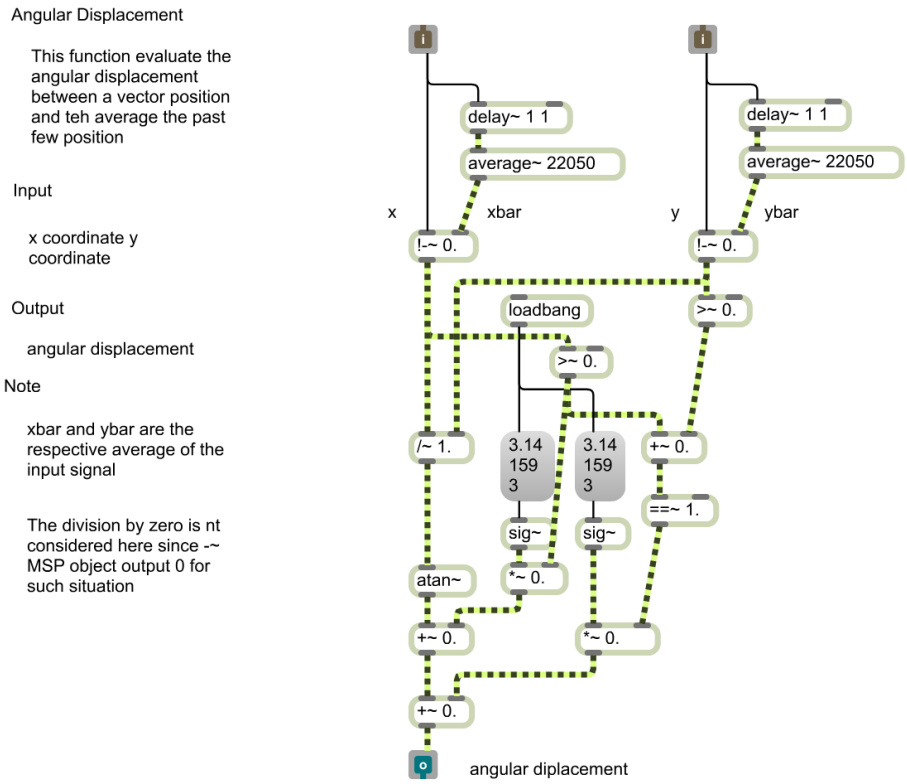


Figure 18 – A screenshot of the AngularDisplacement subpatch. The angular displacement is calculated by determining the position of the two markers, dividing the first by the second, and then taking the arctangent to determine the angle. I think that the extra stuff is just to avoid dividing by zeros (as can be problematic with arctangent).

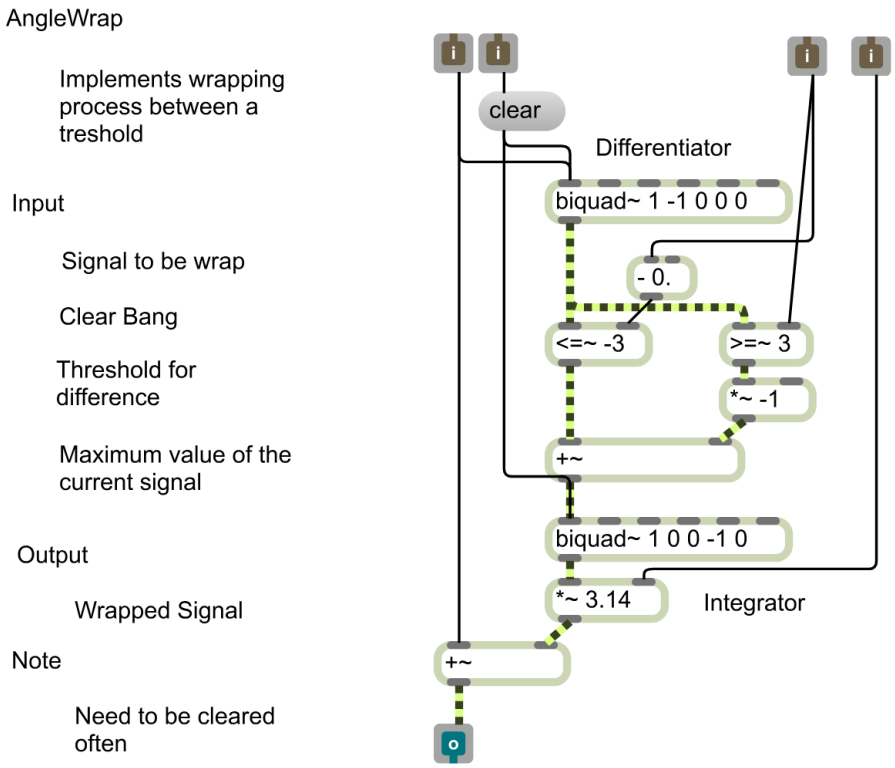


Figure 19 – A screenshot of the AngleWrap subpatch. Although there are four inputs, as we can see in the CircularMovementPoly Fig. 16, only the first is used, **making this patch rather useless**. Furthermore, angle wrapping should not be necessary when using arctangent, as everything is automatically between 0 and 2π. Furthermore, if the goal of this patch is to keep everything within a predefined set of angles, in Max 5, there is an object that accomplishes this easily (but I don't remember which one it is).

5.1 The PCA

The PCA is a tool defines the movement as a set of independent basis vectors that are summed together in the correct proportion. It may be useful for the study of movement in performance, and is implemented here as a preprocessing step that is available on sub-regions of the body (head, left body, right body, left leg, right leg) in Channels 6-10. This section will present the use of the PCA on the head as an example of how it is implemented. One should note that in the current implementation, the PCA is not flexible as showin in the PCAHeadReader Fig. 23.

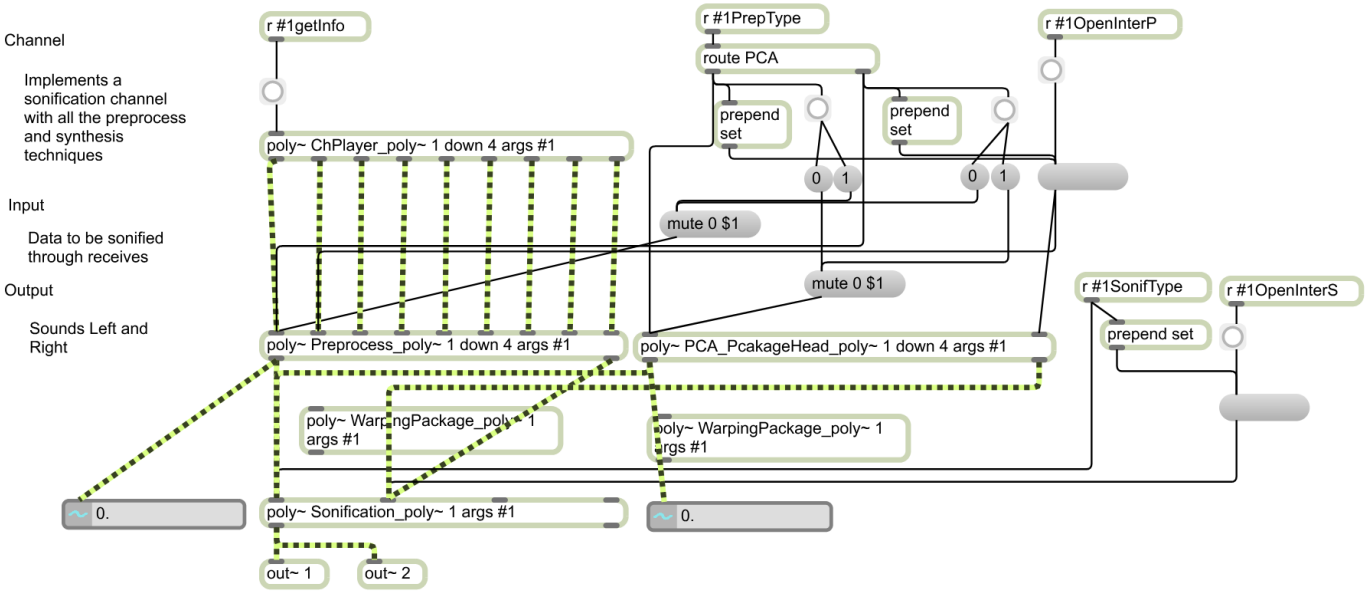


Figure 20 – A screenshot of the ChannelPCAPolysubpatch. It closely resembles the ChannelPoly(Fig. 6), but offers the possibility of a PCA on the head. As we can see, the choice of PCA sidesteps the PreprocessPoly but has two outputs which, like the PreprocessPoly are the instantaneous value without and with amplitude scaling to velocity. The PCAPackageHead Fig. 21 is where the actual PCA preprocessing happens.

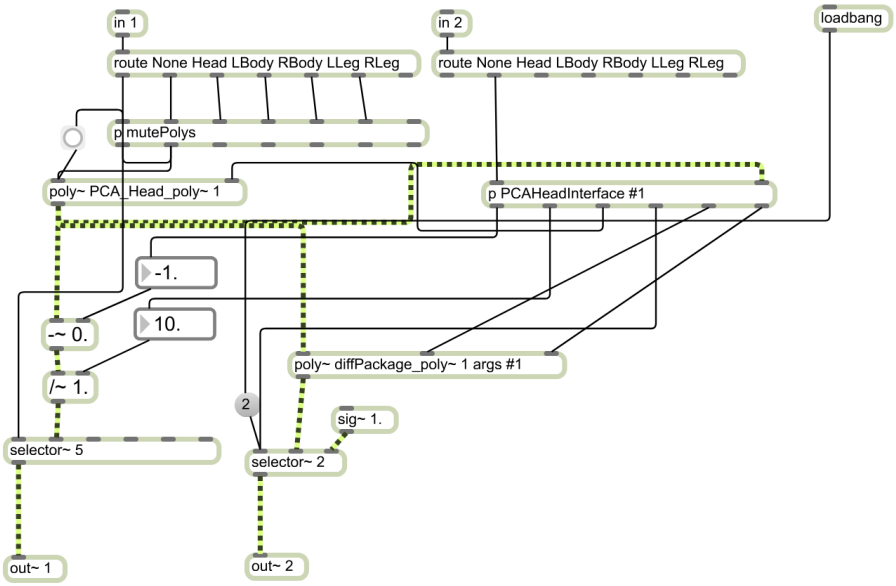


Figure 21 – A screenshot of the PCAPackageHead subpatch. The first inlet declares the PCA type to be used, and the second tells the PCAHeadInterface Fig. 40(b) to be sent to the front. These inlets come from the ChannelPCAPoly subpatch Fig. 20, and the outputs that go to the left and right inlets of the SonificationPoly Fig. 9 are the same as for the other preprocessing options PreprocessPoly Fig. 8. For the PCAHeadInterface(Fig. 40(b)), the first two inlets are used to scale the value between 0 and 1, while the third-fifth are used in the velocity to sound amplitude mapping. The PCAHeadPoly is shown in Fig. 22.

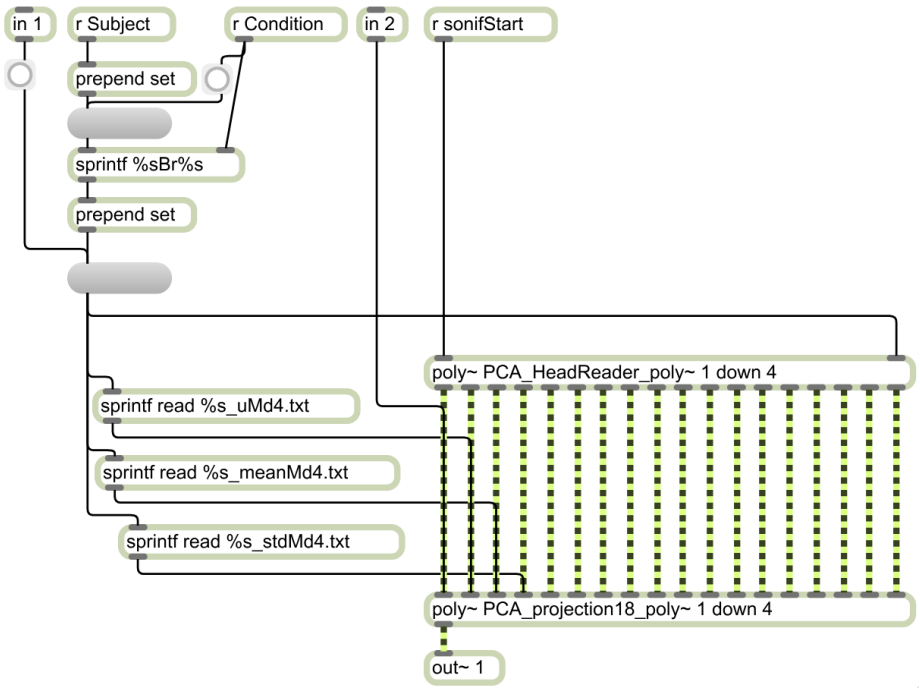


Figure 22 – A screenshot of the PCAHeadPoly subpatch. This patch implements the PCA via the PCAHeadReader and PCAProjection18Poly are shown in Figs. 23 and 24. The .txt files are created in the MatLab script and are the median, mean, and standard deviation I think, but I don't know of what. The output (out~ is the instantaneous value of the PCA on the head..

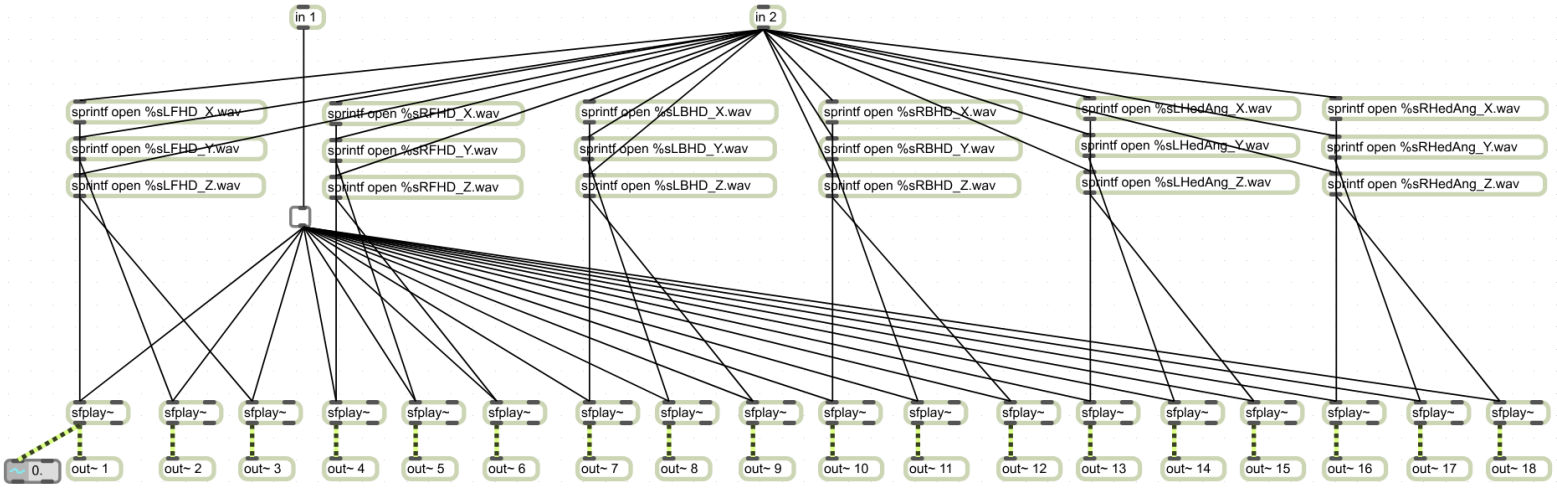


Figure 23 – A screenshot of the PCAHeadReader. As we can see, this PCA implentation is very dependent upon the marker indices of the Vicon system and does not offer flexibility in the chosen data inputs.

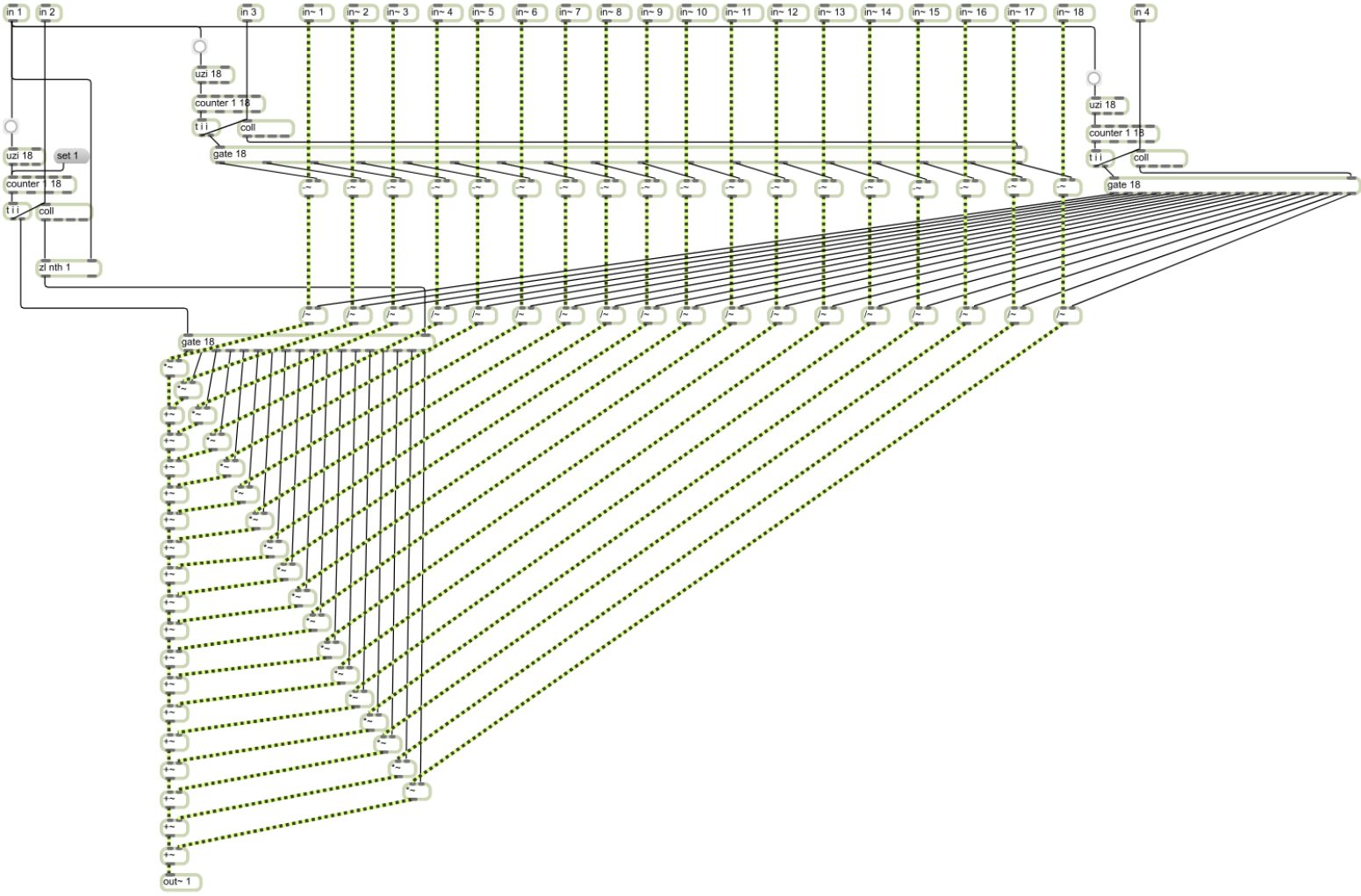


Figure 24 – A screenshot of the PCAProjection18Poly. This patch is where the real-time calculation of the PCA happens, the output of which is the PCA along one of the basis dimensions.

5.2 Time Warping

Time-warping is an important feature of any tool to study gesture quantitatively. It comes from the need to compare across performers who do not necessarily play in synchrony. As shown in Fig. 6 and Fig. 20, it is not currently attached to the primary interface, but I will present it anyways. It scales the timing of events relative to another subject. Details of the implementation are provided in [3].

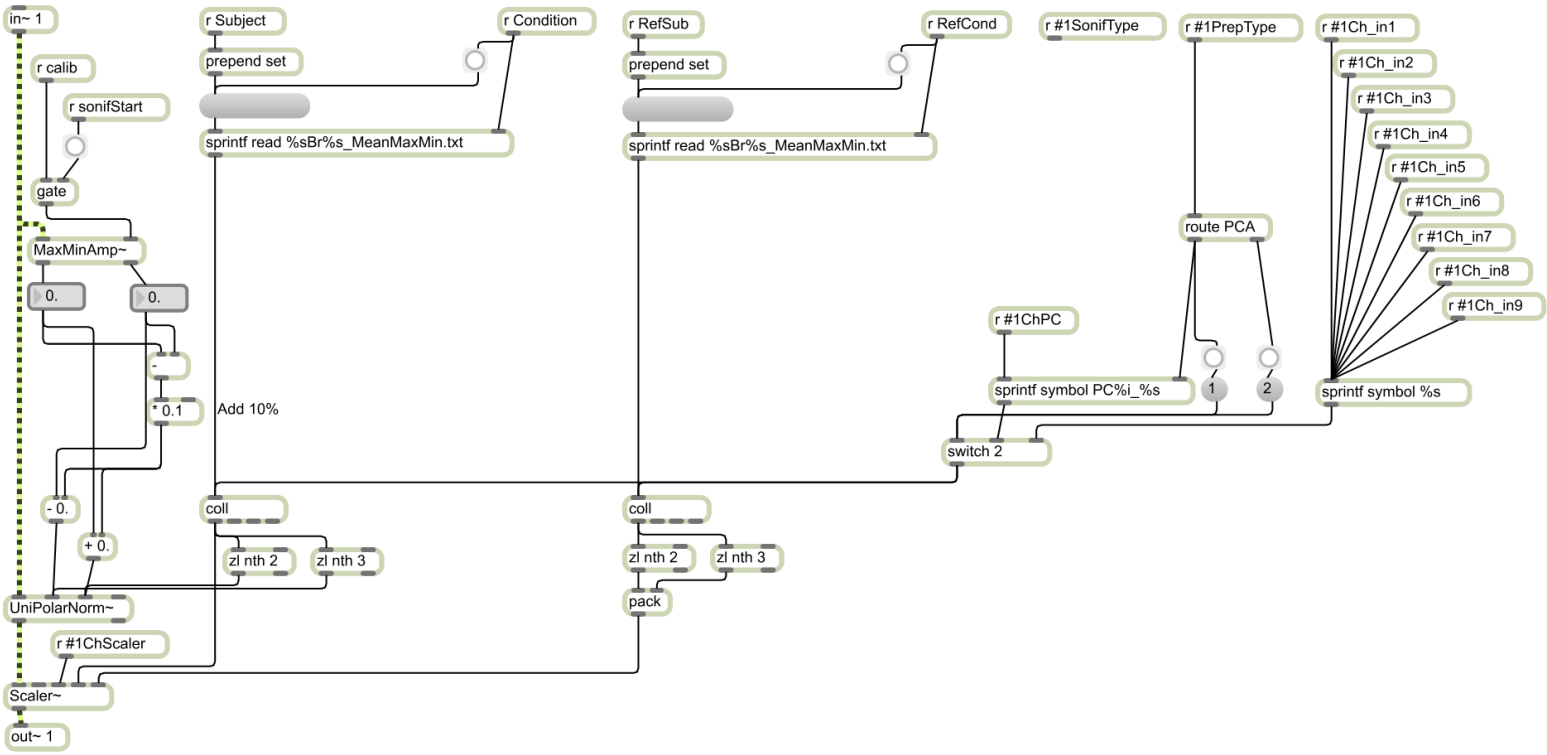


Figure 25 – A screenshot of the WarpingPackagePoly subpatch. Within the patch are the Scaler Fig. 26, MaxMinAmp Fig. 27, and UniPolarNorm Fig. 28.

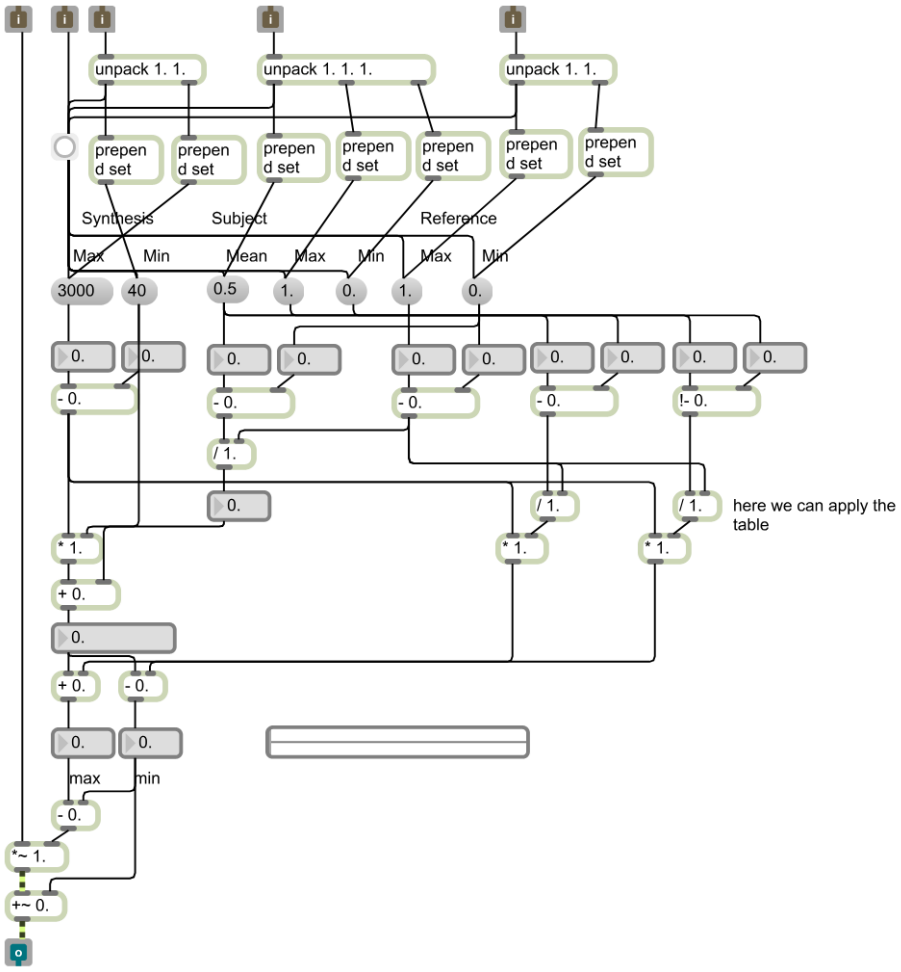


Figure 26 – A screenshot of the Scaler subpatch.

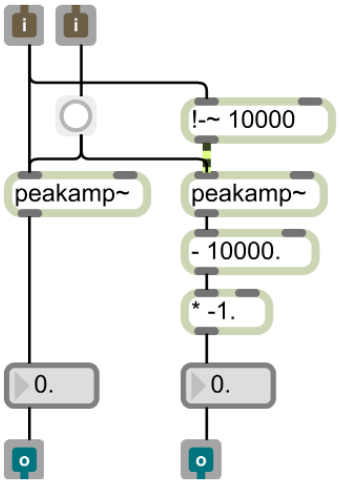


Figure 27 – A screenshot of the MaxMinAmp subpatch.

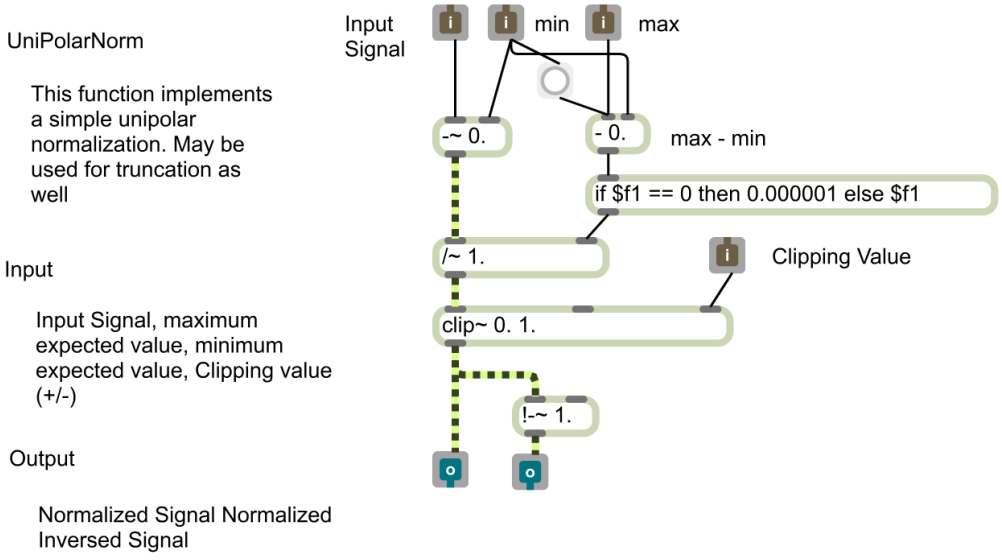


Figure 28 – A screenshot of the UniPolarNorm subpatch.

5.3 Miscellaneous Patches

The following two patches are in the source code file, but have not been located within patches. They may need to be deleted.

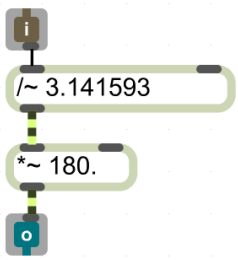


Figure 29 – A screenshot of the DegToRad subpatch

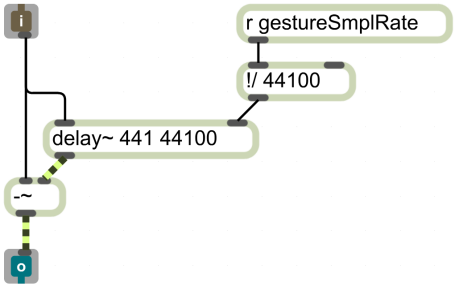


Figure 30 – A screenshot of the GestureDifference subpatch

6 Synthesis Mappings

After data has been preprocessed, it is mapped onto synthesis parameters. This process flow can be visualized best in the ChannelPoly subpatch Fig. 6. In the ChannelPoly, the first and second outputs of the PreprocessPoly are the value and value velocity as determined through comparison with a sample of variable delay. The value of the third input of PreprocessPoly is not known as there is a discrepancy in the number of outputs of the PreprocessPoly as shown in Fig. 8 due to a mistake I made. Regardless, the first two inputs into the SonificationPoly Fig. 9 will be discussed presently. As shown in the SonificationPoly there is one output for each of the synthesis choices except for the RissetSynthPoly which has two outputs. Regardless of the number, they are sent to the left and right speaker as output for the SonificationPoly. This output is shown in the ChannelPoly which is then sent to the SonificationCoreand out through the chosen output level channel shown in the desktop Fig. 1.

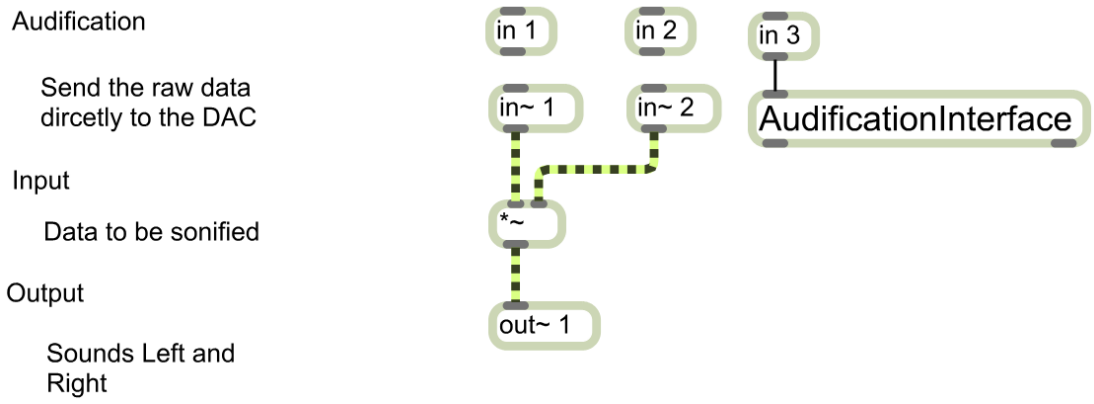


Figure 31 – A screenshot of the AudificationPoly subpatch. The value of the signal is multiplied by the amplitude and sent through the first outlet. The AudificationInterfaceFig. 41(e) is not connected right now. It is not clear how it is meant to work. Normally, audifications are sped-up data values in order to make the output audible [5, 6].

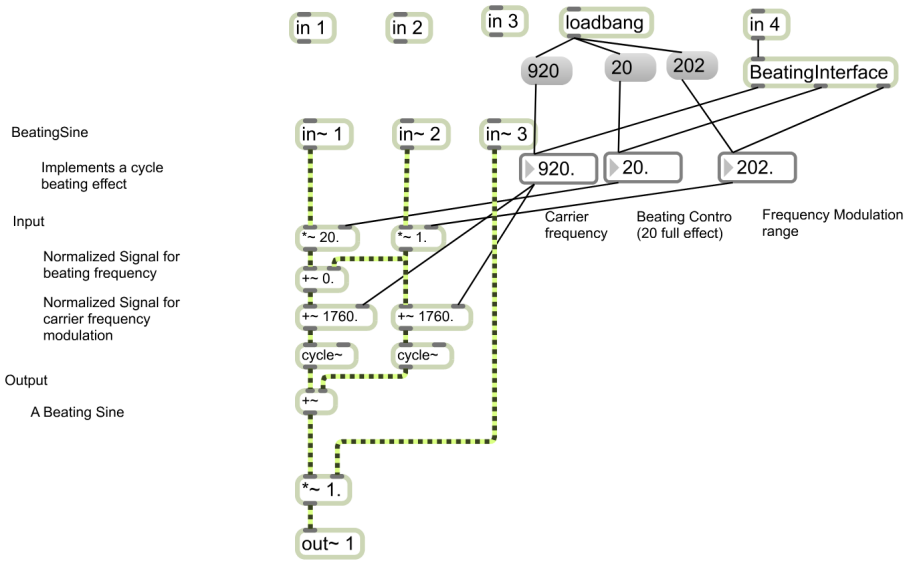


Figure 32 – A screenshot of the BeatingSinePoly subpatch. The BeatingInterface Fig. 41(a) controls the carrier frequency, the beating control and the frequency modulation. From this figure, it appears the mysterious third input in∼ 3 is the strict amplitude as opposed to the amplitude mapped to the signal, which is the second input in∼ 2.

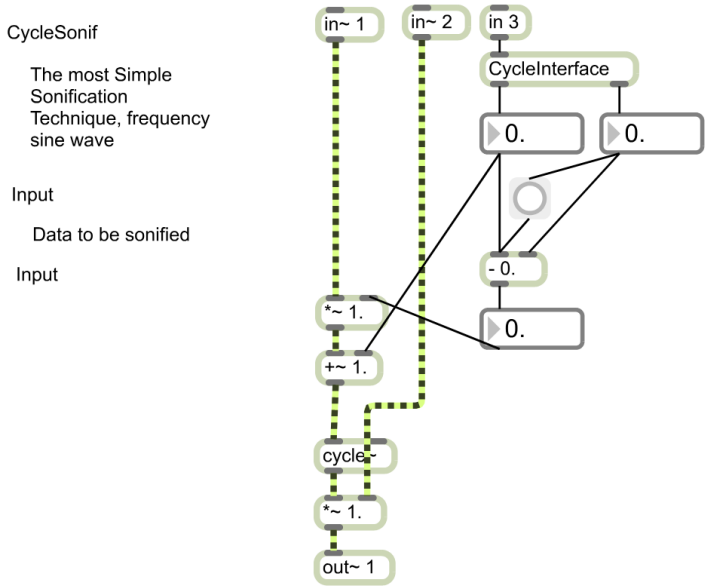


Figure 33 – A screenshot of the CycleSonifPoly subpatch. The CycleInterface Fig. 41(b) allows the user to choose a frequency range and offset, though it is not clear why an offset is necessary with a predefined frequency range. In the present patch, the second input in∼ 2 is the amplitude.

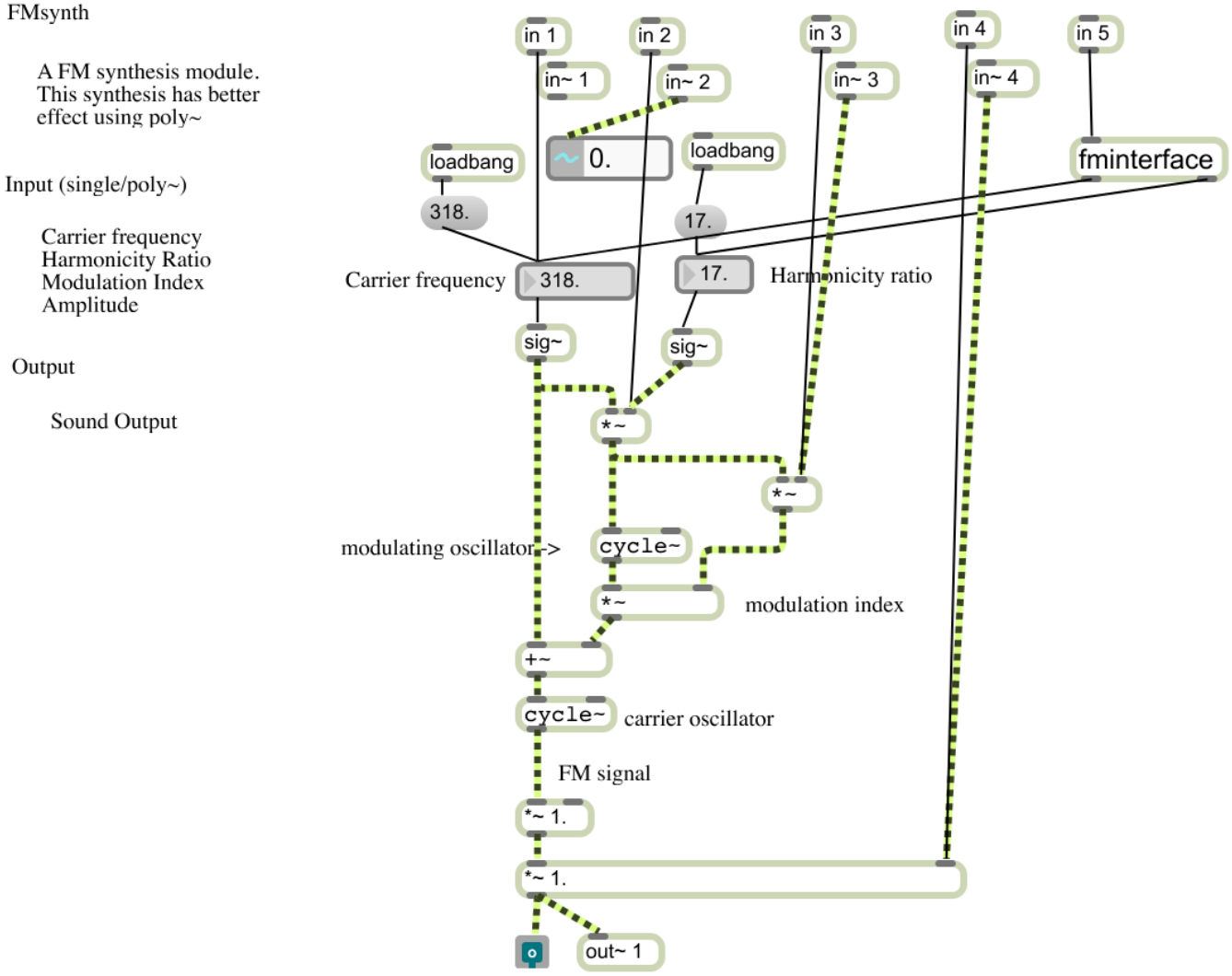


Figure 34 – A screenshot of the FMSynthPoly subpatch. Once again, clearly my picture of the SonificationPoly Fig. 9 should be retaken as the patch shows four inputs while the SonificationPoly only shows three. The FMInterface allows the user to control the carrier frequency and the harmonicity ratio. The third input controls the modulation index and the fourth controls the sound amplitude.

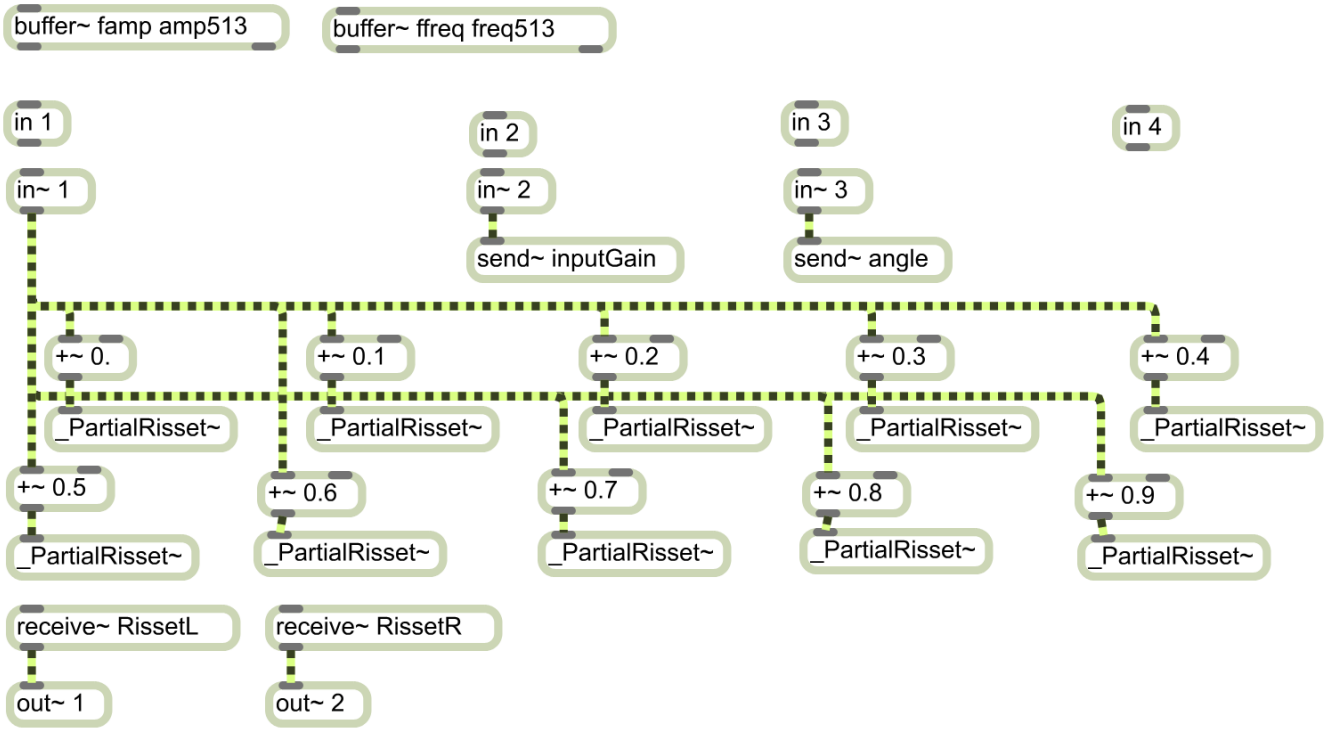


Figure 35 – A screenshot of the RissetSynthPoly subpatch. No subinterface for this patch was found, and as there are no input variables, it is assumed this synth requires only the signal inputs to work. As we can see, the second and third inputs in$\sim$ 2 and in$\sim$ 3 are sent as variables into the PartialRisset Fig. 36 subpatch directly, whereas the first input is sent as a signal that is sent to PartialRisset after being added by $0.1k$ where $k \in 0, ...9$.The reason for this is not known. RissetL and RissetR are the outputs for the left and right speaker inherited from the PartialRisset subpatch.

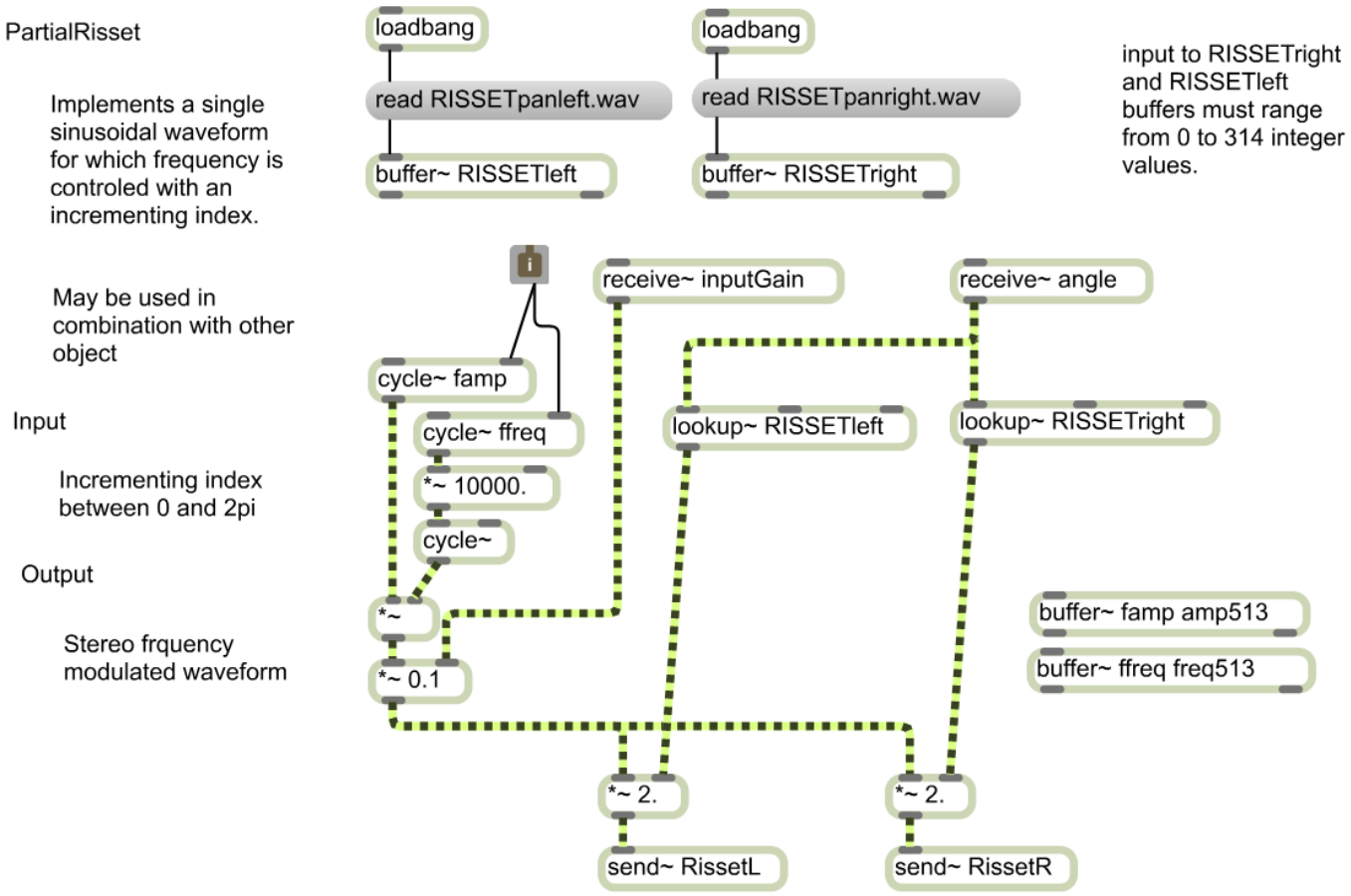


Figure 36 – A screenshot of the PartialRisset subpatch. The patch calls upon RISSETpanleft.wav and RISSETpanright.wav, which are used as a map of the *angle* signal inherited from the RissetSynthPoly Fig. 35. The buffers amp513 and freq513 are to be found in the source code. The input is the speed at which these buffers are being read through as a fraction of the first input of the RissetSynthPoly. This synthesis choice was first implemented in [7].

7 Weight Transfer and Video Interface

The mapping of weight transfer was mentioned as a preprocessing step under development. It is presented without comment as it exists currently in the WeightTransferPoly Fig. 37 and the WeightTransferInterface Fig. 40(c). The VideoInterface Fig. 38 is provided for the purpose of complete documentation. It has very simple features.

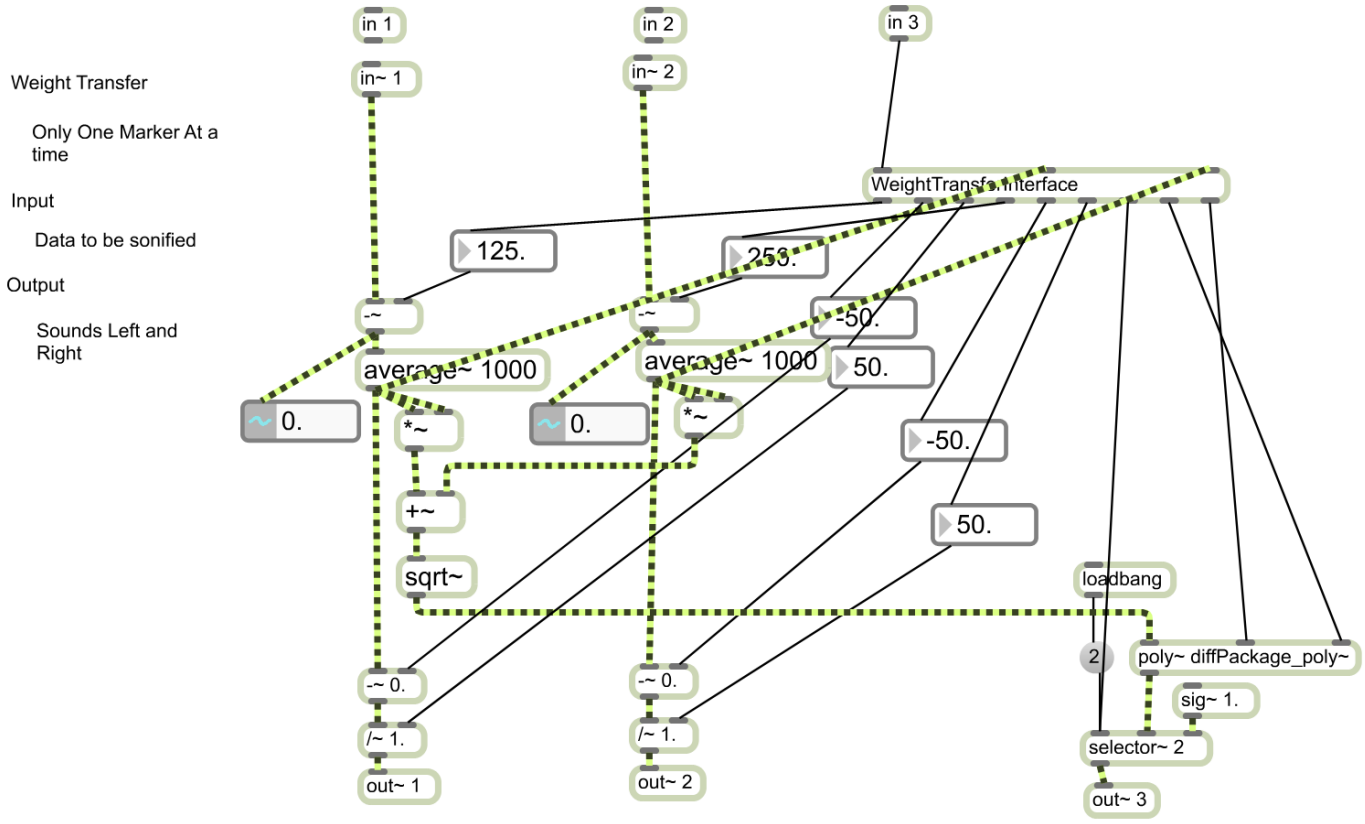


Figure 37 – A screenshot of the WeightTransferPoly, a preprocessing option that would appear int the PreprocessPoly subpatch. It is currently under development. An easy way to implement weight transfer would also be to use the RawDataPoly Fig. 10 with a hip-based input parameter. Considering this fact, it seems like this feature might be useless. The WeightTransferInterface is displayed in Fig. **??**.

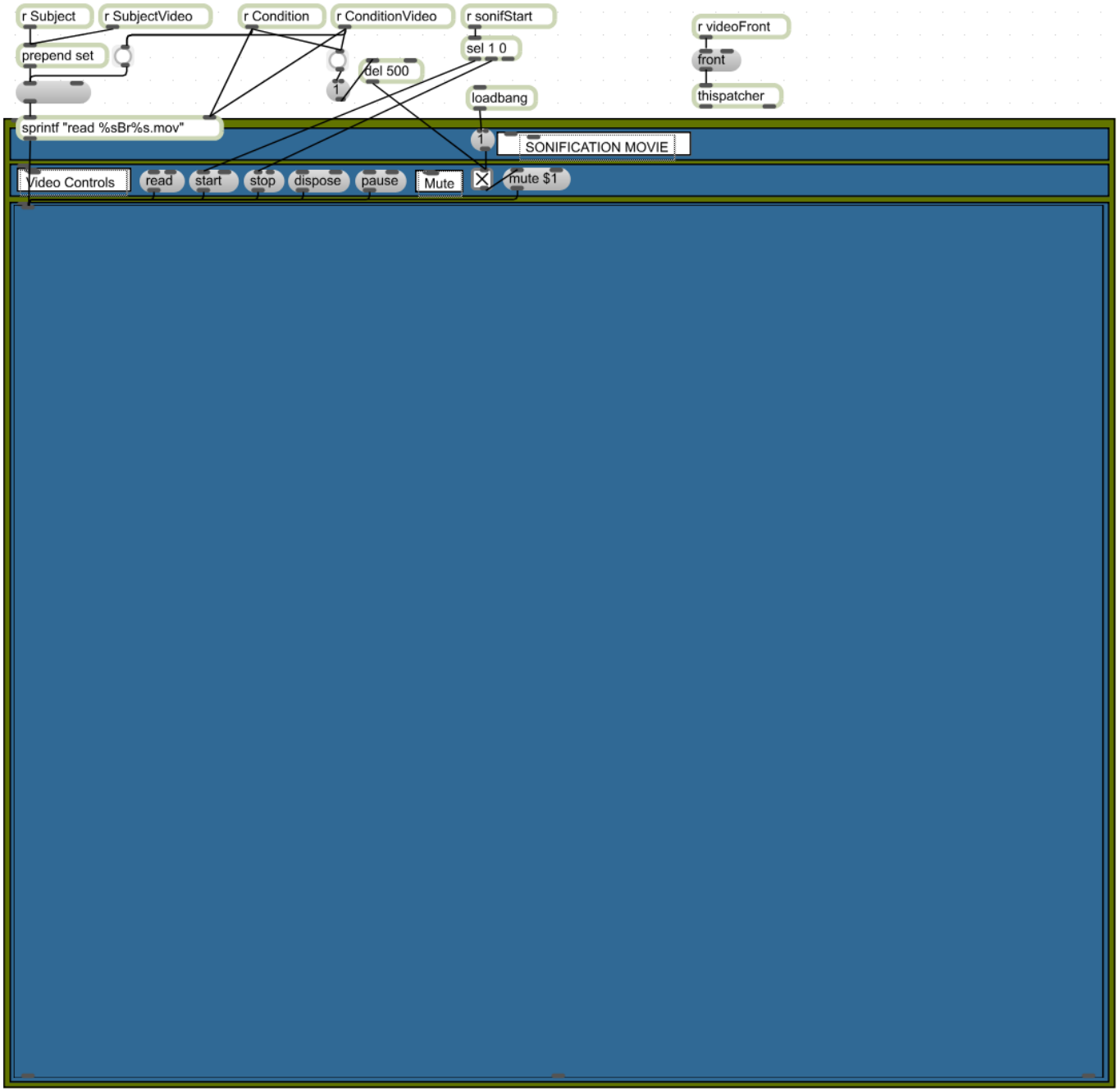
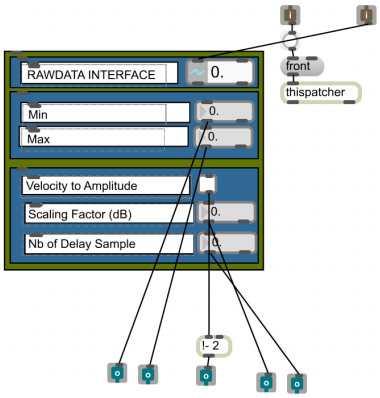


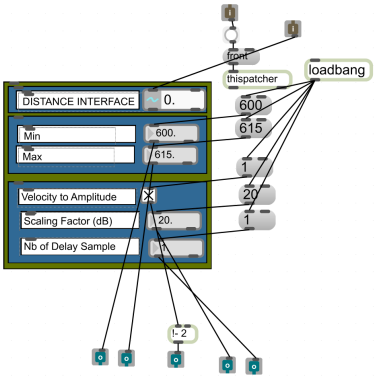
Figure 38 – A screenshot of the VideoInterface. The integration of video is an important feature of any tool for sonification of gesture. Sound is being used as a complementary tool, not as a replacement. The patch is shown "unlocked" in order to show the video controls.

8 The Subinterfaces

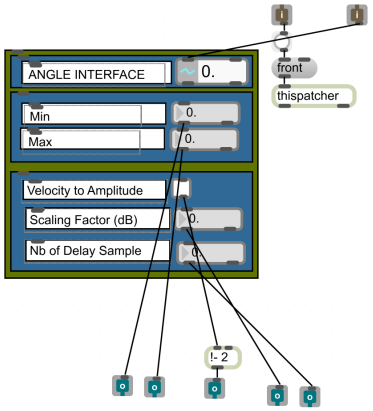
8.1 Preprocessing Subinterfaces



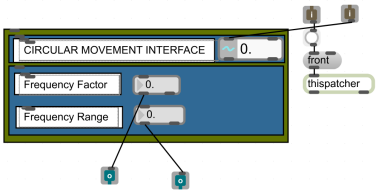
(a) A screenshot of the RawDataInterface, which sends user defined inputs to the RawDataPoly Fig. 10.



(b) A screenshot of the EulerDistanceInterface, which sends user defined inputs to the EulerDistancePoly Fig. 13.

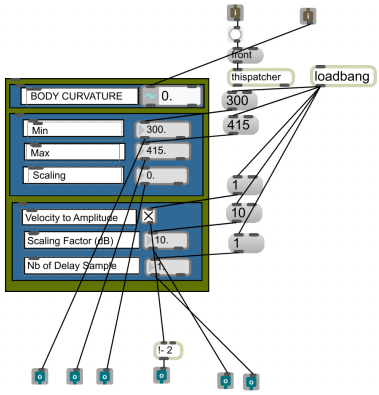


(c) A screenshot of the EulerAngleInterface, which sends user defined inputs to the EulerAnglePoly Fig. 14.

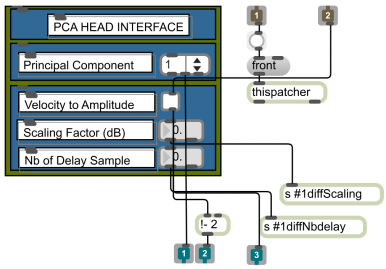


(d) A screenshot of the CircularMovementInterface, which sends user defined inputs to the CircularMovementPoly Fig. 16.

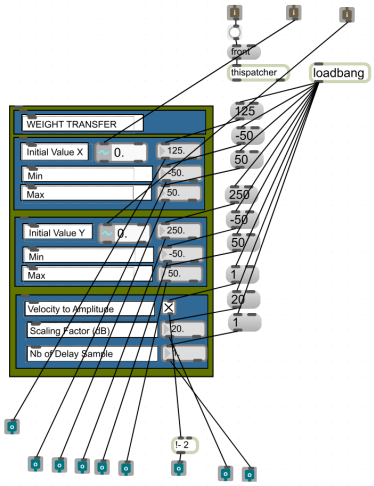
Figure 39 – Screenshots of the preprocessing subinterfaces. Once a user has chosen one of the preprocessing options, these subinterfaces are available for choosing the specific parameters of the preprocessing. All are displayed "unlocked" to be able to see their outputs.



(a) A screenshot of the BodyCurvatureInterface, which sends user defined inputs to the BodyCurvaturePoly Fig. 15.



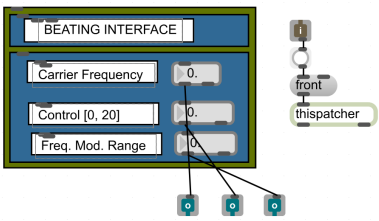
(b) A screenshot of the PCAHeadInterface, which sends user defined inputs to the PCAPackageHead Fig. 21.



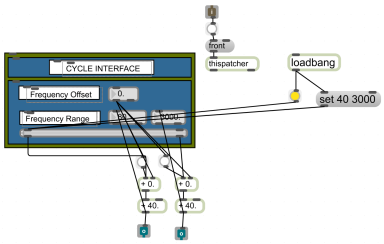
(c) A screenshot of the WeightTransferInterface, which sends user defined inputs to the WeightTransferPoly Fig. 37.

Figure 40 – Screenshots of the preprocessing subinterfaces. Once a user has chosen one of the preprocessing options, these subinterfaces are available for choosing the specific parameters of the preprocessing. All are displayed "unlocked" to be able to see their outputs.

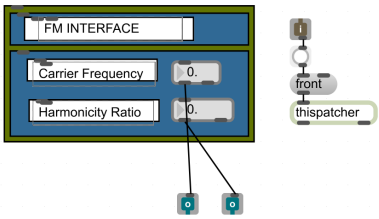
8.2 The Synthesis Subinterfaces



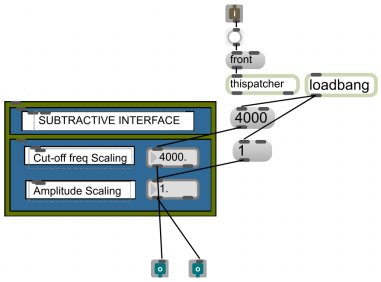
(a) A screenshot of the BeatingInterface, which sends user defined inputs to the BeatingSinePoly Fig. 32.



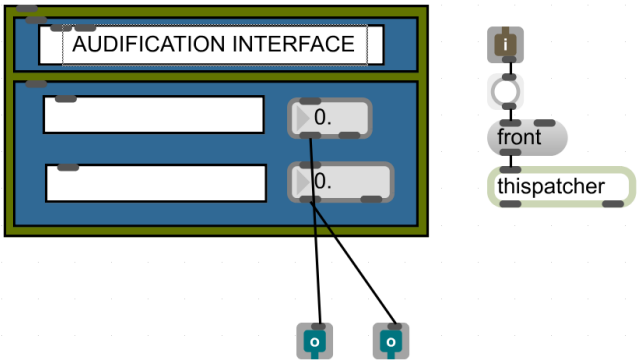
(b) A screenshot of the CycleInterface, which sends user defined inputs to the CycleSonifPoly Fig. 33.



(c) A screenshot of the FMInterface, which sends user defined inputs to the FMSynthPoly Fig. 34.



(d) A screenshot of the SubtractiveInterface, which sends user defined inputs to the SubtractiveSynthPoly Fig. **??**.



(e) A screenshot of the AudificationInterface, which sends user defined inputs to the AudificationPoly Fig. 31.

Figure 41 – Screenshots of the synthesis subinterfaces. Once a user has chosen one of the synthesis options, these subinterfaces are available for choosing the specific parameters of the synthesis. All are displayed "unlocked" to be able to see their outputs.

9 Notes and Future Work

This report was written as part of the final project for MUMT 620 Gestural Control of Sound Synthesis (Fall 2011). Alexandre Savard was of great help in getting the code up and running again. A little more needs to be done in order to make this fully operational as made clear in the text.

As agreed by myself and Alexandre, the desktop should be implemented in SuperCollider. SuperCollider is a code-based, open-source software platform for sound synthesis with GUI capabilities as well. Without doubt, the current system can be implemented in SC in a way that is more clear, simple, and concise. Further, SC is becoming increasingly common among scientific researchers, and is easily accessible as an open-source language.

References

[1] R. M. Winters, "Literature review and new directions for sonification of musicians' ancillary gestures," IDMIL, McGill University, Tech. Rep., 2011.

[2] R. M. Winters, M. Wanderley, A. Savard, and V. Verfaille, "A sonification tool for analysis of large databases of movement data from musical performance," IDMIL, McGill University, Tech. Rep., 2011.

[3] A. Savard, "When gestures are perceived through sounds: A framework for sonification of musicians' ancillary gestures," Master's thesis, McGill University, 2009.

[4] F. Grond, T. Hermann, V. Verfaille, and M. M. Wanderley, "Methods for effective sonification of clarinetists' ancillary gestures." in *Proceedings of Gesture Workshop 2009*, 2009, pp. 171–181.

[5] G. Kramer, Ed., *Sonification, Audification, and Auditory Display.* SantaFe Institute, 1994.

[6] F. Dombois and G. Eckel, "Audification," in *The Sonification Handbook*, T. Hermann, A. Hunt, and J. G. Neuhoff, Eds. Berlin, Germany: Logos Publishing House, 2011, ch. 12, pp. 301–324. [Online]. Available: http://sonification.de/handbook/chapters/chapter12/

[7] V. Verfaille, O. Quek, and M. M. Wanderley, "Sonification of musicians ancillary gestures," in *Proceedings of the International Conference on Auditory Display*, 2006, pp. 194–197.