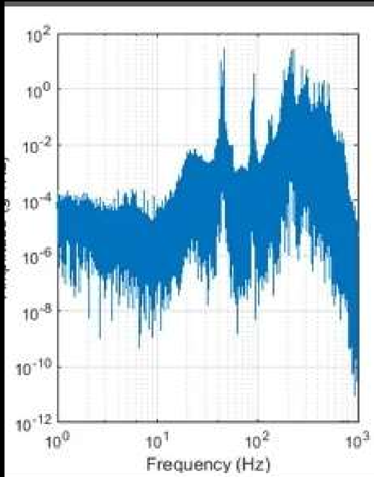
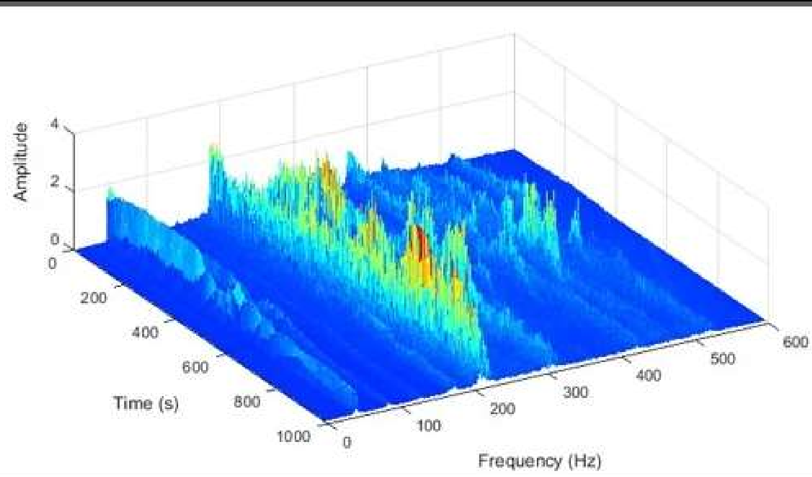
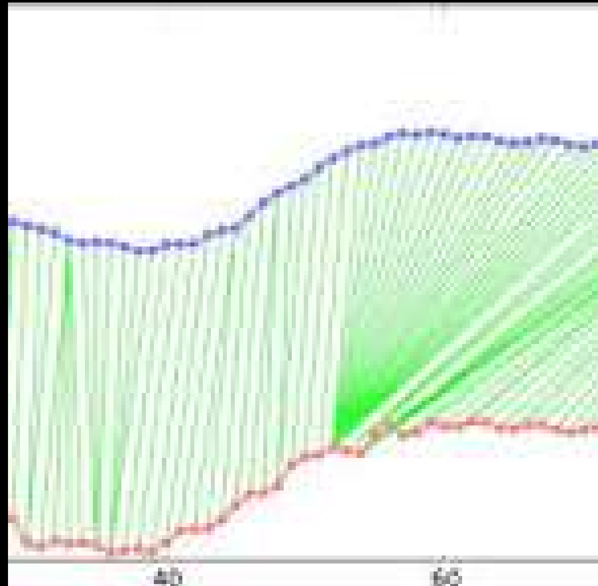
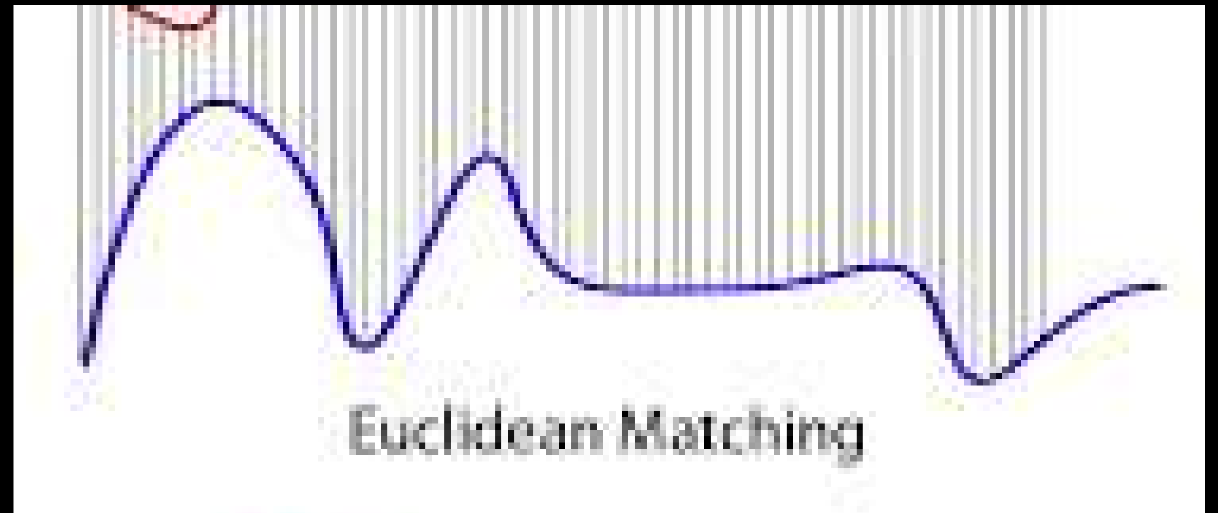
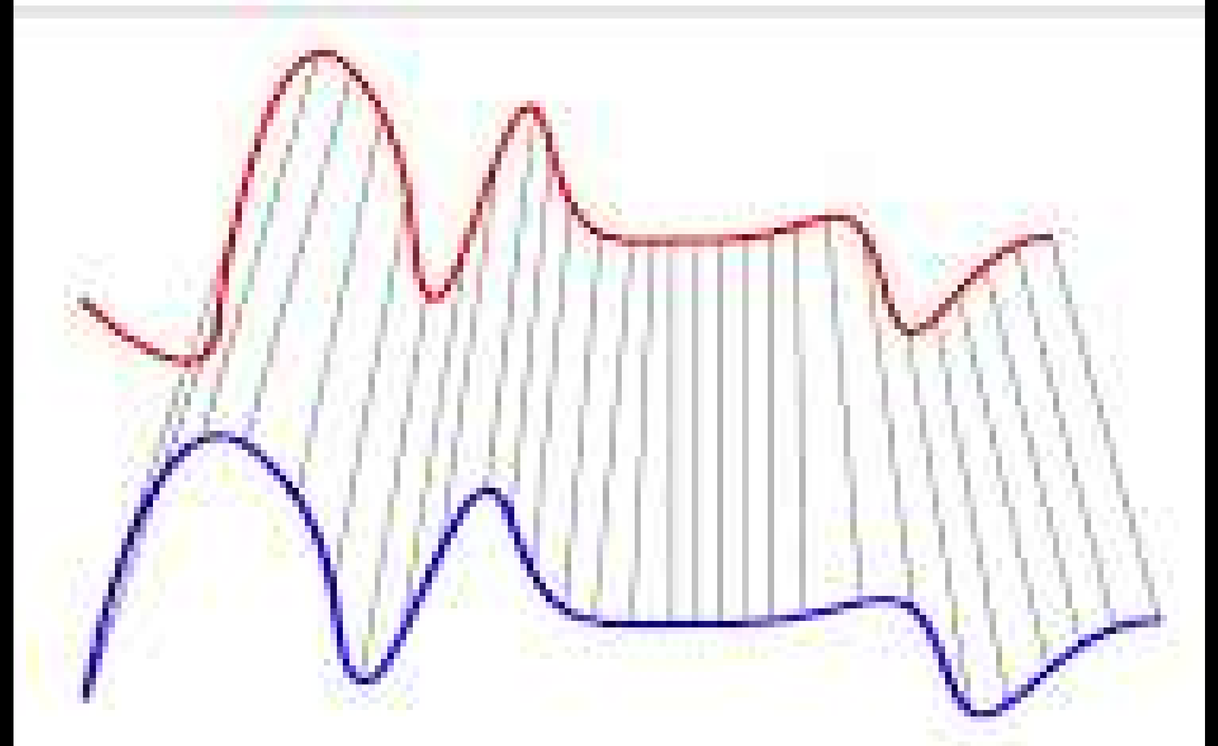




PSD **SPECTROGRAM**







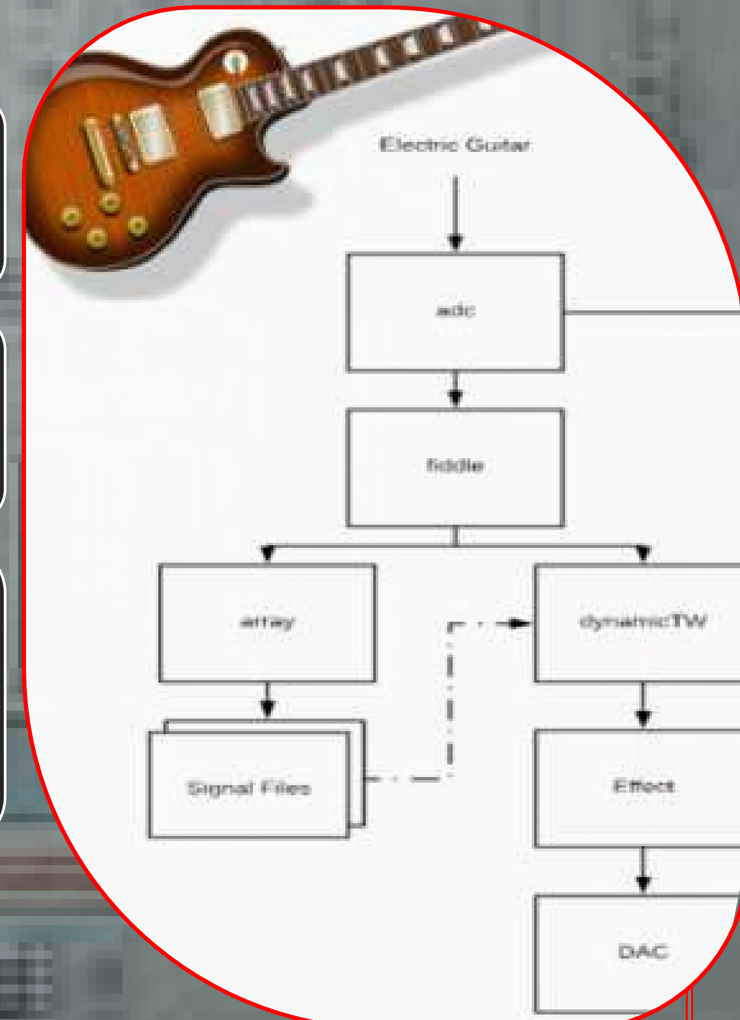
Dynamic Time Warping Triggered Guitar Effects Platform

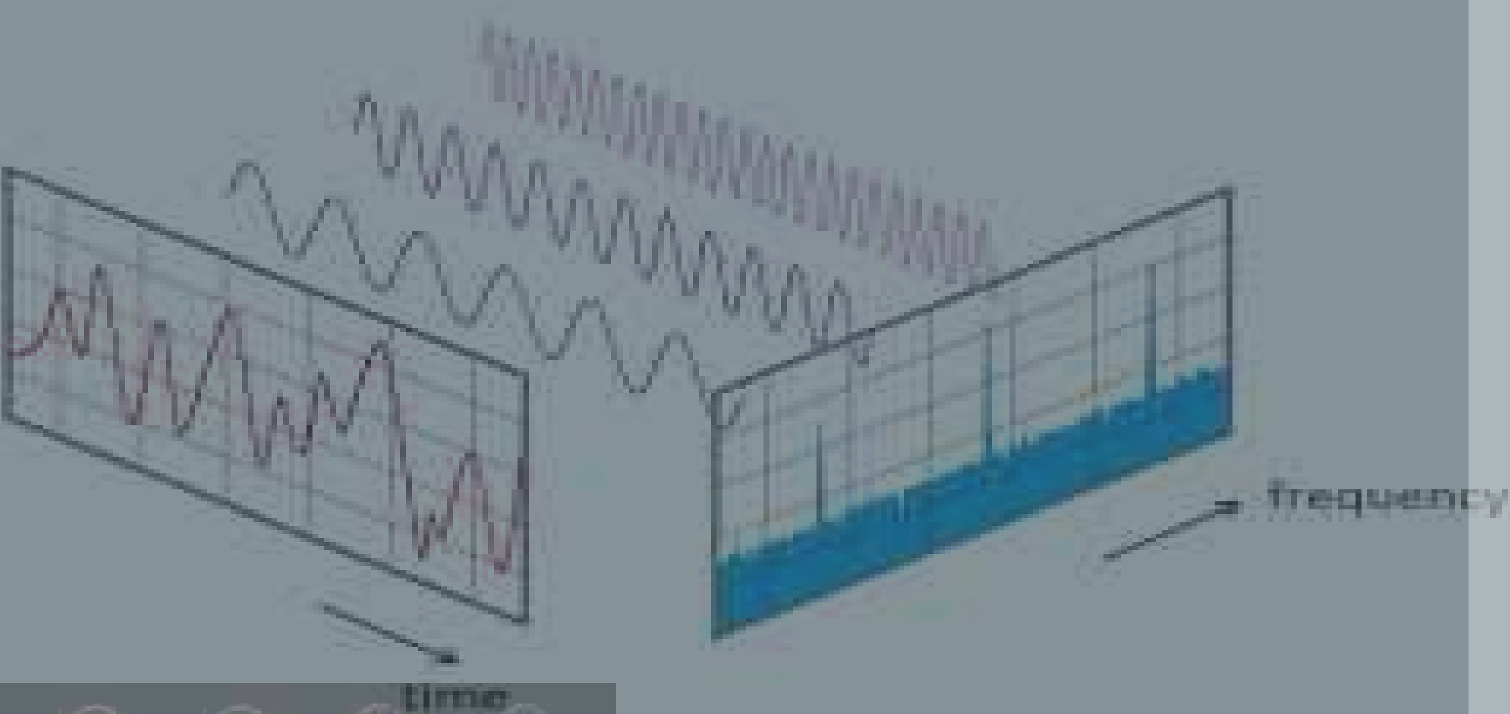
In live performance, guitar effect pedals are a versatile yet limiting asset, requiring presence of mind on the part of the performer. This platform offers an automatic solution to the restrictions that guitar effect pedals present.

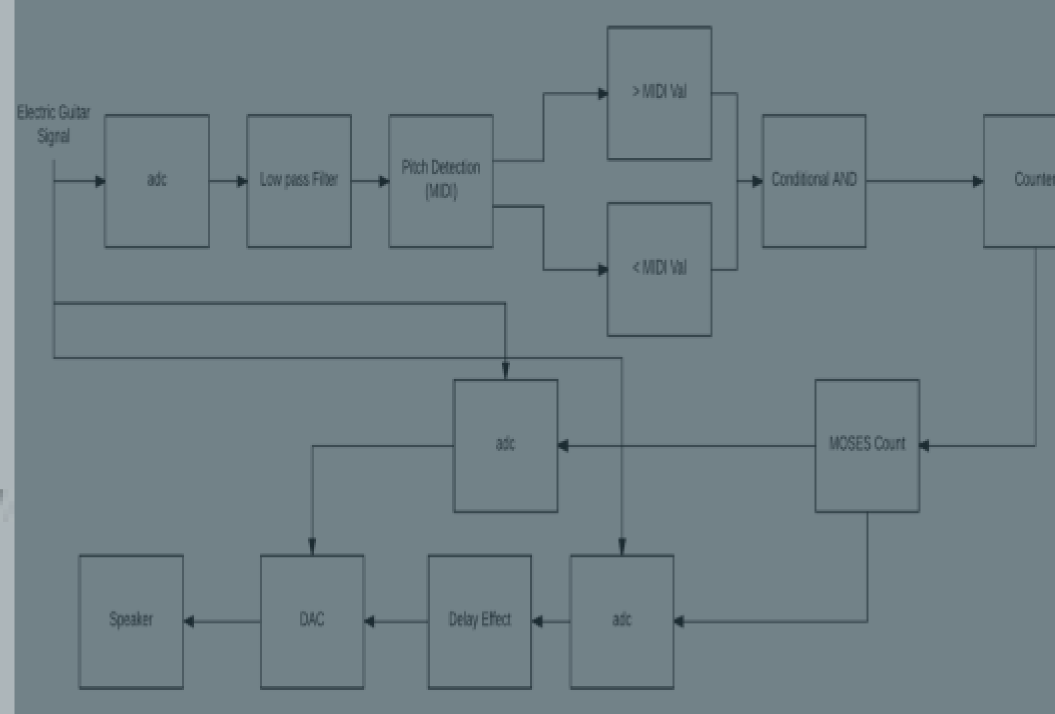
System Architecture:

read in a guitar signal during the 'learning' phase and isolate subsections of a performance needed to generate the DTW learned-threshold.

then implement an analysis based on a modified dynamic time warping (DTW) algorithm, to compare the DTW cost function of incoming live audio against the cost-thresholds determined in the pre recording phase.







This automation was achieved through the use of Pure Data, a GUI for audio manipulation applications, with embedded Python externals. When the algorithm detects a match, the platform runs the 'pure' digitalized audio signal through custom made PD effects patches!

Dynamic Time Warping External:

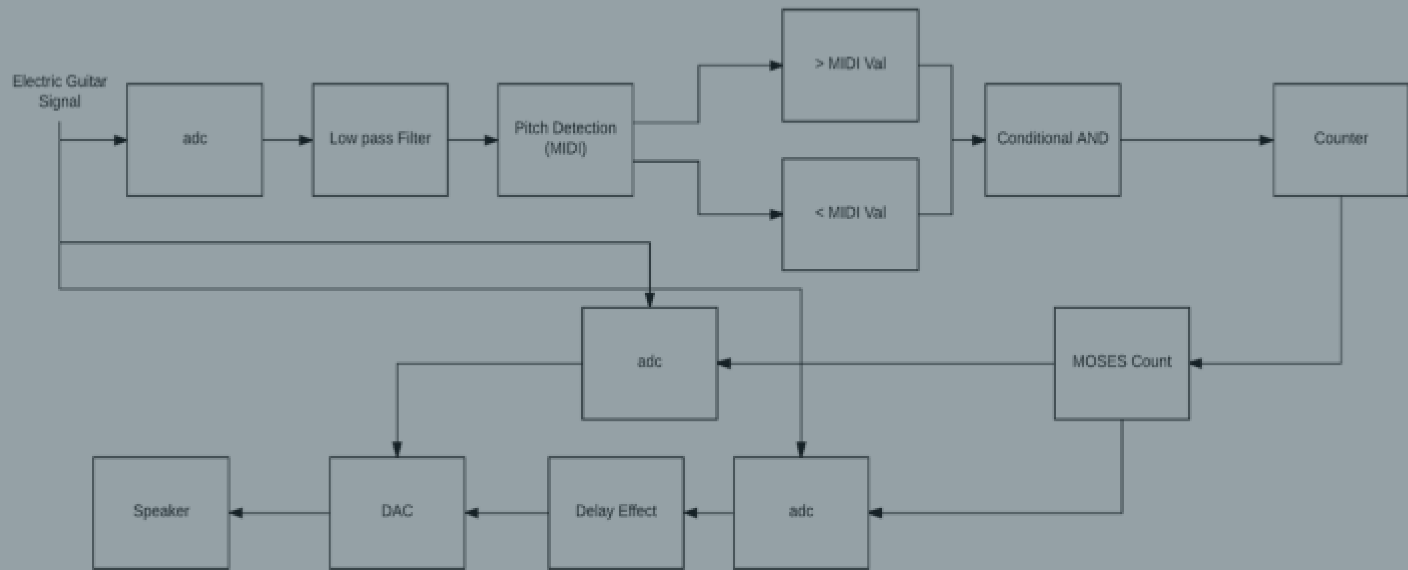
Feed in two song performances for learning phase and obtain Least Cost Path (LCP) for each sub signal

Compare Incoming live signal with sub signal of one of the recorded performance

When LCP value is less than or equal to the LCP obtained from learning phase, trigger guitar effect

How It works:

This automation was achieved through the use of Pure Data, a GUI for audio manipulation applications, with embedded Python externals. When the algorithm detects a match, the platform runs the 'pure' digitalized audio signal through custom made PD effects patches!

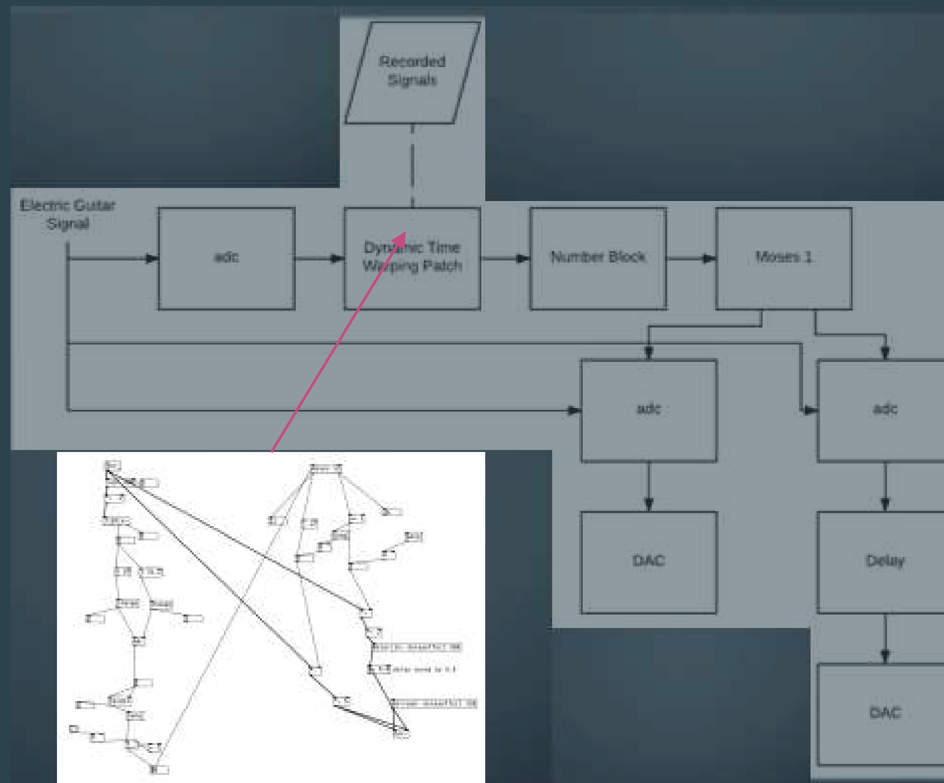


Dynamic Time Warping External:

•Feed in two song performances for learning phase and obtain Least Cost Path (LCP) for each sub signal

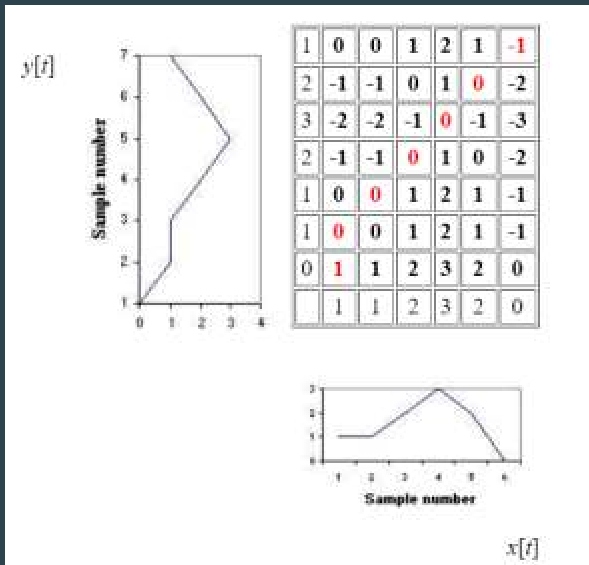
•Compare Incoming live signal with sub signal of one of the recorded performance

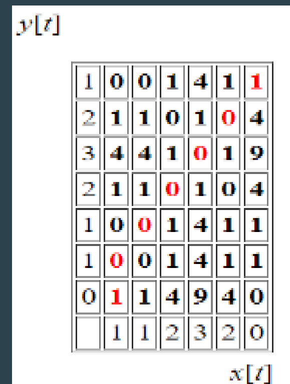
•When LCP value is less than or equal to the LCP obtained from learning phase, trigger guitar effect



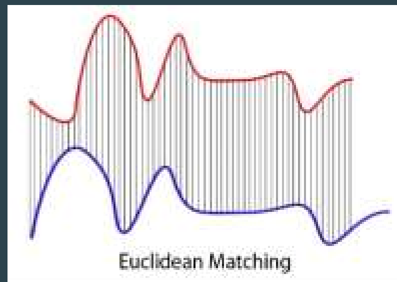
DTW Path

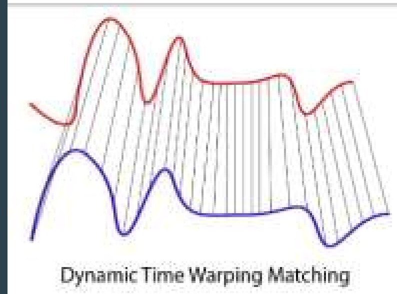
- The process can be thought of conceptually as arranging the two sequences on the sides of a grid
- Each cell within the grid will be filled in with a distance measure comparing the corresponding elements of the two sequences

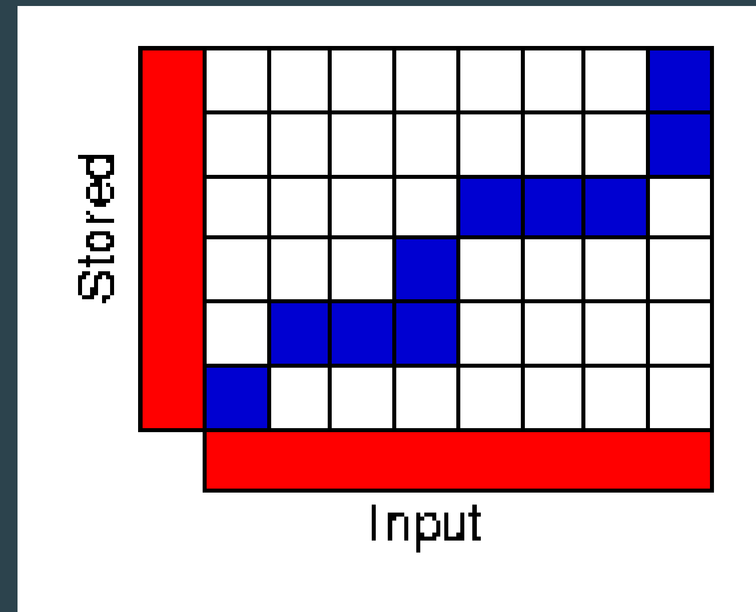




3	0 B: 1	0	1
2	0 B: 1	0 B: 2 L: 1 BL: 1	1 B: 7 L: 2 BL: 2½
1	1	1 L: 2	4 L: 6
	1	2	3







et X = (x1,x2,...,xN) and Y = (y1,y2,...,yM) be midi sequences from the Pure Data Fiddle object, where M>>N(this means that Y is the database sequence).Next, a local cost function 'c' is assigned to each element of the DTW grid. At this point the algorithm must find a subsequence Y(a∗ : b∗) := (ya∗,ya∗+1,...,yb∗) that minimizes the DTWdistance to our incoming signal over all possible subsequences of the recorded feature sequence.

Algorithmically speaking that is:

(a* ,b*):= argmin (DTW (X,Y(a:b))

The indices a∗ and b∗ in addition to the least cost alignment possible between the incoming signal and the subsection Y (a∗ : b∗) of the stored performance can be computed by a modification of the standard DTW algorithm.

In order to select a path of least resistance 'p∗', naturally one would have to calculate every conceivable path through the grid. Unfortunately this requires computational complexity that grows exponentially in bounds N and M. This process can be optimized to an O(NM) complexity computation. In a broad sense, the concept is to penalize paths between the database and the query that match the query to indices near the beginning or end of the database as to avoid a one to one match between signals.

Algorithm: (Accumulated cost matrix and DTW-distance)

Instantiate sequences: X(1:n) = (x1,...xn) and Y (1:m) = (y1,...ym)

Set

D(n,m) = DTW(X(1:n),Y (1:m))

D(n, m) is a N × M matrix 'D' called the *accumulated cost matrix* . A tuple (n, m) representing a matrix entry of the *cost matrix* 'C' or of 'D' will be referred to as a cell.

D satisfies the following:

D(n,1) =Σ (from k=1 to n) c(xk,y0) " n ∈ [1 : N]

D(1,m) = c(x0,ym) "m ∈ [1 : M]

D(n,m) = argmin{D(n−1,m−1),D(n−1,m),D(n,m−1)}+c(xn,ym) " n ∈ [2 : N] and m ∈ [2 : N]

One can also define an extended accumulated cost matrix:

Setting:

D(n,0) = ∞ " n ∈ [0 : N]

D(0,m) := 0 " m ∈ [0 : M].

The index b∗ can be determined from 'D' :

b∗ =argmin {D(N,b)}

To determine the starting index of the subsequence a∗ and the optimal warping path between the stored and incoming signals.

Input: Accumulated cost matrix D.

Output: Optimal warping path p∗.

The optimal path p∗ = (p1, . . . , pL) is computed in reverse order of the indices starting with pL = (N,b∗).

pose pl = (n, m) has been computed. In case (n, m) = (1, 1), one must have l to 1 and we are finished.

else:

pl−1 =

"n = 1 : (1 , m − 1)

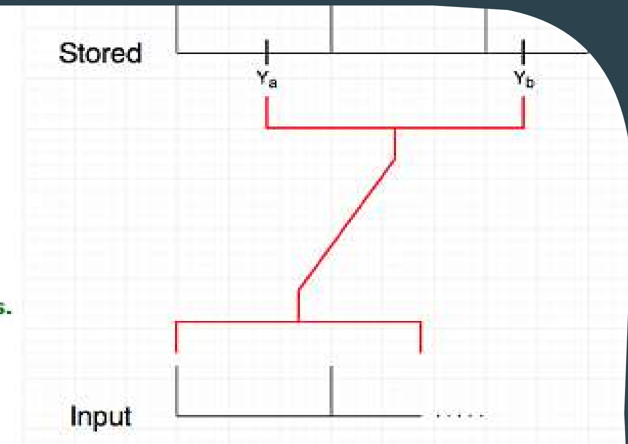
"m=1 :(n−1,1)

else: argmin{D(n − 1, m − 1), D(n − 1, m), D(n, m − 1)}

a∗ is the maximal index such that pl = (a∗,1)

All elements of the stored sequence 'Y' left of ya∗ and right of yb∗ are excluded from consideration and do not incur additional costs.

The optimal warping path between X and Y (a∗ : b∗) is given by (pl,...,pL)



'D' can be used to generate a list of subsequences of incoming signal that match the recorded trigger point.

Create distance function :

Δ : [1 : M] → R.. Δ(b) = D(N,b)

Δ assigns to each index b the minimal DTW distance 'Δ(b)' attainable between the stored sequence and the subsequence of the incoming signal that ends on index b.

"b ∈ [1 : M], the DTW-minimizing 'a' ∈ [1 : M] can be computed starting with pL = (N,b).

If Δ(b) is small $b ∈ [1 : M] and if a ∈ [1 : M] denotes the corresponding DTW-minimizing index, then the subsequence Y (a:b) matches the incoming section

Input: *incoming signal* X = (x1,...,xN), *database sequence* = (y1,...,yM),

cost threshold: 'τ'

Output: Ranked list of matches between incoming signal and subsections of database that have a match to the input below the threshold τ

Algorithm: (Match list tracker)

1.)Ranked list must initially be empty

2.)Calculate D

3.)Calculate distance function Δ using Δ(b) for each subsequence of the database 'Y'

4.)Select minimum b∗ ofΔ.

5.)If Δ(b∗) > τ then a *match has been detected*.

6.)Calculate corresponding match-subsequence index a∗ ∈ [1 : M]

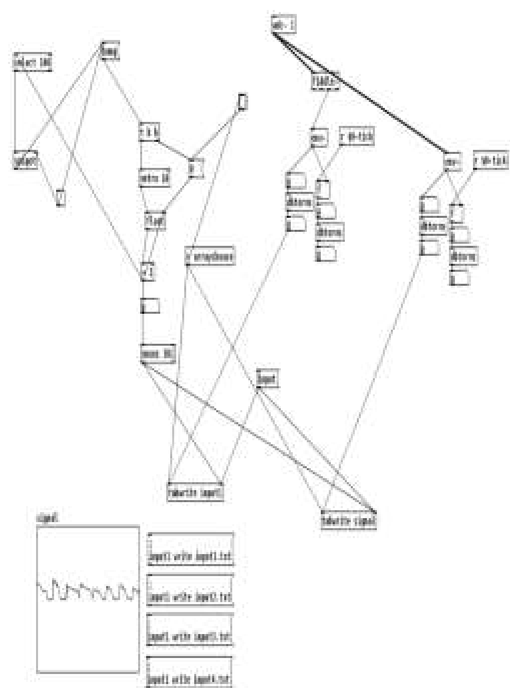
7.)add subsequence Y (a∗ : b∗) to ranked list

8.)Set Δ(b) = ∞ "b within a suitable neighborhood of b∗

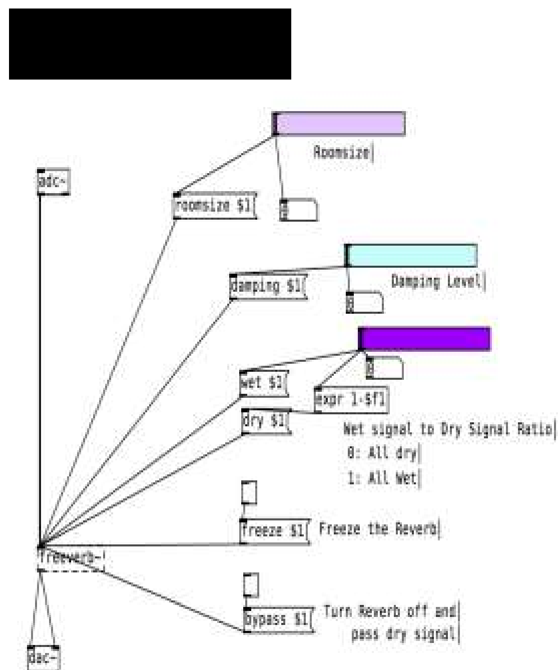
9.)Continue by calculating the next minimizing index until input ends.

The rule Δ(b) = ∞ is intended to exclude an a region bounded by the nearest local maximums to b∗ from computation. This prevents a match list that contains many subsequences that differ by only a slight shift.

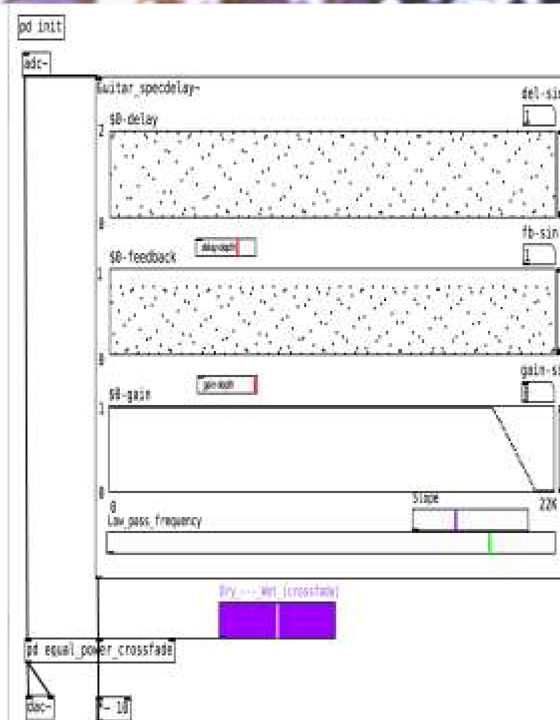
Recording (.wav) Patch:



Reverb Patch



Spectral Delay Patch



(visible part of)Fuzz Patch

adc~

*~ 40 Amplify sound from guitar

clip~ -0.5 0.5 Clip to produce audible distortion

dac~

Signal sent DI into Scarlet 2i2 to be ported into pure data and then back out the DAC into a Fender Super Champ X2 FSR 15-watt 1x10" Tube Combo Amp using the settings below and was captured by my I phone's mic.

Guitar

Pre Amp

A/D

x

Code

x

DA

Amplifier

Pure Data Patch Demos

```c
#include "m_pd.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE_ARRAY 100
#define TEST_SIZE 100
static t_class *dynamicTW_class; //handle for the class
float recording_array[SIZE_ARRAY] = {0};
int arr_position = SIZE_ARRAY - 1;
float saveValue = 0.0;
int triggerGlobal = 0;
/* struct to hold cost of arrival from left, bottom, and diagonal */
typedef struct _leftbottom{
    float left;
    float bottom;
    float diag;
}leftBD; //typedef name
typedef struct _dynamicTW{
    t_object x_obj;
    int flag; //differentiates if how result of lcp is stored
    int match; //if 0 signal does not match else matches
    float testArray[TEST_SIZE];
    float storedSignalOne[SIZE_ARRAY];
    float storedSignalTwo[SIZE_ARRAY];
    t_inlet *in_mod_A, *in_mod_B;
    float signal[SIZE_ARRAY];
    float lcpValue;
    float compareValue;
    float initMatrix[SIZE_ARRAY][SIZE_ARRAY];
    leftBD costValues[SIZE_ARRAY-1][SIZE_ARRAY-1];
}t_dynamicTW; //typedef name
void checkStorage(t_dynamicTW *x){ //check if values being stored correctly
    int i;
    post("in check storage");
    for(i = 0; i < TEST_SIZE; i++){
        post("Signal 1: %f", x->storedSignalOne[i]);
        post("Signal 2: %f", x->storedSignalTwo[i]);
    }
}
```

```c
void signalMatch(t_dynamicTW *x)
{
    float tenup = saveValue + (.0125 * saveValue);
    float zeroValue = 0.00;
    if (x->compareValue <= tenup && x->compareValue >= zeroValue)
    {
        x->match = 1;
        /* add code to trigger effect */
        post("Incoming Signal Matches Stored Signal. compareValue is %f and lcpValue is %f", x->compar
    }
    else
    {
        x->match = 0;
        post("Signal NO Match. compareVal is %f while lcpVal is %f", x->compareValue, saveValue);
    }
}
void reverseArray(t_dynamicTW *x)
{
    int i, j;
    i = SIZE_ARRAY - 1;
    j = 0;
    while (i > j)
    {
        float temp = x->signal[i];
        x->signal[i] = x->signal[j];
        x->signal[j] = temp;
        i--;
        j++;
    }
}
void fileReader1(t_dynamicTW *x, char *path)
{

//char const* const fileName = "C:\\Users\\Raki\\Documents\\GitHub\\dynamicTW\\TB\\input.txt" ;/* shou
    FILE *file = fopen(path, "r"); /* should check the result */
    char line[256];
    int i = 0;
    while (fgets(line, sizeof(line), file))
    {
        /* note that fgets don't strip the terminating \n, checking its
           presence would allow to handle lines longer that sizeof(line) */
        //post("file1: value at line %d is %s", i, line);
        x->storedSignalOne[i] = atof(line); // |||||||||||||||||||| REMOVE 1000
        i++;
    }
    fclose(file);
}
void fileReader2(t_dynamicTW *x, char *path)
{

//char const* const fileName = "C:\\Users\\Raki\\Documents\\GitHub\\dynamicTW\\TB\\input.txt" ;/* shou
    FILE *file = fopen(path, "r"); /* should check the result */
    char line[256];
    int i = 0;
    while (fgets(line, sizeof(line), file))
    {
        /* note that fgets don't strip the terminating \n, checking its
           presence would allow to handle lines longer that sizeof(line) */
        //post("file2: value at line %d is %s", i, line);
        x->storedSignalTwo[i] = atof(line); // |||||||||||||||||||||||||| REMOVE 1000
        i++;
    }
    fclose(file);
}
void replaceSignal2(t_dynamicTW *x)
{
    int i;
    for (i = 0; i < SIZE_ARRAY; i++)
    {
        x->storedSignalTwo[i] = x->signal[i];
    }
}
```

```c
 4. calculate least cost path
*/
void leastCostPath(t_dynamicTW *x)
{
    float temp, min;
    float result = 0;
    int i = SIZE_ARRAY - 2;
    int j = SIZE_ARRAY - 2;
    //    post("starting point value left is %f", x->costValues[i][j].left);
    //    post("starting point value bottom is %f", x->costValues[i][j].bottom);
    //    post("starting point value diag is %f", x->costValues[i][j].diag);
    while (i >= 0 && j >= 0)
    {
        float checkLeft = x->costValues[i][j].left;
        float checkBottom = x->costValues[i][j].bottom;
        float checkDiagonal = x->costValues[i][j].diag;
        temp = (checkLeft < checkBottom) ? checkLeft : checkBottom;
        min = (checkDiagonal < temp) ? checkDiagonal : temp;
        result += min;
        //post("min value is %f", min);
        if (min == checkDiagonal)
        {
            if (i - 1 < 0 && j - 1 < 0)
            {
                break;
            }
            else if (i - 1 < 0)
            {
                j--;
            }
            else if (j - 1 < 0)
            {
                i--;
            }
            else
            {
                i--;
                j--;
            }
            //post("changing index to [%d][%d]",j,i);
        }
        else if (min == checkLeft)
        {
            if (j - 1 < 0)
            {
                break;
            }
            else
            {
                j--;
            }
            //post("changing index to [%d][%d]",j,i);
        }
        else
        {
            if (i - 1 < 0)
            {
                break;
            }
            else
            {
                i--;
            }
            //post("changing index to [%d][%d]",j,i);
        }
    }
    if (x->flag == 0)
    {
        //        x->lcpValue = result;
        //        post("lcpValue: least cost path is %f", x->lcpValue);
        saveValue += result;
        post("storing to saveValue: current saveValue is %f", saveValue);
    }
    else if (x->flag == 1)
    {
```

```c
//testing if values are correct
    //    int qq, rr;
    //    for(qq = 0; qq< SIZE_ARRAY-1; qq++){
    //        for(rr=0; rr<SIZE_ARRAY-1; rr++){
    //            post("value at row %d and column %d for left is 

//          post("value at row %d and column %d for bottom is %f
    //            post("value at row %d and column %d for diag is 
    //            post(" ");
    //        }
    //    }
    leastCostPath(x); //finds least cost path
}
/*what to do when bang is hit*/
void dtw_onBangMsg(t_dynamicTW *x)
{
    //    post("*[dtw ] is set to go");
    //    if(x->signal[SIZE_ARRAY - 1] == 0){
    //        post("NONE");
    //    }
    //    else{
    //        post("ARRAY TEST: %f", x->signal[SIZE_ARRAY - 1]);
    //    }
    x->match = 0;
    x->flag = 0; //makes so that lcp value gets stored in lcpValue
    saveValue = 0.0;
    arr_position = SIZE_ARRAY - 1;
    fileReader1(x, "C:\\Users\\Raki\\Documents\\GitHub\\dynamicTW\\
    fileReader2(x, "C:\\Users\\Raki\\Documents\\GitHub\\dynamicTW\\
    dtw_genMatrix(x);
    fileReader2(x, "C:\\Users\\Raki\\Documents\\GitHub\\dynamicTW\\
    dtw_genMatrix(x);
    fileReader2(x, "C:\\Users\\Raki\\Documents\\GitHub\\dynamicTW\\
    dtw_genMatrix(x);
    saveValue = saveValue / 3;
    post("saveValue: Least Cost Path is %f", saveValue);
    //post("does it get here?");
    triggerGlobal = 1;
}
```

```c
void dtw_free(t_dynamicTW *x)
{
    inlet_free(x->in_mod_A);
    inlet_free(x->in_mod_A);
}
void dtw_onSet_A(t_dynamicTW *x, t_floatarg f)
{ /*function that gets called when an input is received */
    while (triggerGlobal == 0)
    {
        /*do nothing*/
    }
    post("Number A: %f sending to array. Arr_position is %d", f, arr_position);
    //recording_array[0] = f;
    if (x->match == 1)
    {
        post("Match has been detected. Freezing Program!");
        while (1)
        {
            /*freezes the program*/
        }
    }
    else if (x->match == 0)
    {

//if (x->signal[SIZE_ARRAY - 1] == 0 && arr_position <= SIZE_ARRAY){ //checks if array is filled. If no
        if (arr_position >= 0)
        { //checks if array is filled. If not then store incoming value to next index
            x->signal[arr_position] = f;
            arr_position--;
        }
        else
        { //If array is filled shift all values by 1 index and store at beginning of array
            int i;
            x->flag = 1; //makes it so that LCP result is stored in compared Value;
            for (i = SIZE_ARRAY - 1; i > 0; i--)
            {
                x->signal[i] = x->signal[i - 1];
            }
            x->signal[0] = f;
            //reverseArray(x); // works
            //post("Value at last index is %f", x->signal[SIZE_ARRAY-1]);
            replaceSignal2(x); //replaces the value in signal 2
            dtw_genMatrix(x);  //performs dtw
            signalMatch(x);    //checks is the signal is correct if it is trigger effect
            //dtw_free(x);
        }
    }
}
```

```c
void dtw_onSet_B(t_dynamicTW *x, t_floatarg f)
{ /*function that gets called when an input is received */
    post("Number B: %f sending to array", f);
    recording_array[0] = f;
}
//initializer for the class
void *dynamicTW_new(t_floatarg f1, t_floatarg f2)
{
//parenth contains creation arg. temp stuff will replaced with arrays
    t_dynamicTW *x = (t_dynamicTW *)pd_new(dynamicTW_class); //initialize struct of type dtw
    x->in_mod_A = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_float, gensym("ratio_A"));
    x->in_mod_B = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_float, gensym("ratio_B"));
    return (void *)x;
}
//function to set up the class and call initializer
void dynamicTW_setup(void)
{
    /*class_new(t_symbol *name, t_newmethod newmethod,
    t_method freemethod, size_t size, int flags, t_atomtype arg1, ...); */
    dynamicTW_class = class_new(gensym("dynamicTW"),        //defines the symbol in puredata
                                (t_newmethod)dynamicTW_new, //inializing method
                                0,
                                sizeof(t_dynamicTW),
                                CLASS_DEFAULT, //makes the box
                                A_DEFFLOAT,
                                A_DEFFLOAT,
                                0);
    class_addbang(dynamicTW_class, (t_method)dtw_onBangMsg);
    class_addmethod(dynamicTW_class,
                    (t_method)dtw_onSet_A,
                    gensym("ratio_A"),
                    A_DEFFLOAT,
                    0);
    class_addmethod(dynamicTW_class,
                    (t_method)dtw_onSet_B,
                    gensym("ratio_B"),
                    A_DEFFLOAT,
                    0);
}
```

```c
void dtw_onSet_A(t_dynamicTW *x, t_floatarg f)
{ /*function that gets called when an input is received */
    while (triggerGlobal == 0)
    {
        /*do nothing*/
    }
    post("Number A: %f sending to array. Arr_position is %d", f, arr_position);
    //recording_array[0] = f;
    if (x->match == 1)
    {
        post("Match has been detected. Freezing Program!");
        while (1)
        {
            /*freezes the program*/
        }
    }
    else if (x->match == 0)
    {
//if (x->signal[SIZE_ARRAY - 1] == 0 && arr_position <= SIZE_ARRAY){ //checks if array is filled
        if (arr_position != 0)
        { //checks if array is filled. If not then store incoming value to next index
            x->signal[arr_position] = f;
            arr_position++;
        }
        else
        { //if array is filled shift all values by 1 index and store at beginning of array
            int i;
            x->flag = 1; //makes it so that LCP result is stored in compared value;
            for (i = SIZE_ARRAY - 1; i > 0; i--)
            {
                x->signal[i] = x->signal[i - 1];
            }
            x->signal[0] = f;
            //reverseArray(x); // works
            //post("Value at last index is %f", x->signal[SIZE_ARRAY-1]);
            replaceSignal2(x); //replaces the value in signal 2
            dtw_genMatrix(x);  //performs dtw
            signalMatch(x);    //checks is the signal is correct if it is trigger effect
            //dtw_free(x);
        }
    }
}
void dtw_onSet_B(t_dynamicTW *x, t_floatarg f)
{ /*function that gets called when an input is received */
    post("Number B: %f sending to array", f);
    recording_array[0] = f;
}
//initializer for the class
void *dynamicTW_new(t_floatarg f1, t_floatarg f2)
{                                                        //parenth contains creation arg. temp
    t_dynamicTW *x = (t_dynamicTW *)pd_new(dynamicTW_class); //initialize struct of type dtw
    x->in_mod_A = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_float, gensym("ratio_A"));
    x->in_mod_B = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_float, gensym("ratio_B"));
    return (void *)x;
}
//function to set up the class and call initializer
void dynamicTW_setup(void)
{
    /*class_new(t_symbol *name, t_newmethod newmethod,
    t_method freemethod, size_t size, int flags, t_atomtype arg1, ...); */
    dynamicTW_class = class_new(gensym("dynamicTW"),        //defines the symbol in puredata
                                (t_newmethod)dynamicTW_new, //inializing method
                                0,
                                sizeof(t_dynamicTW),
                                CLASS_DEFAULT, //makes the box
                                A_DEFFLOAT,
                                A_DEFFLOAT,
                                0);
    class_addbang(dynamicTW_class, (t_method)dtw_onBangMsg);
    class_addmethod(dynamicTW_class,
                    (t_method)dtw_onSet_A,
                    gensym("ratio_A"),
                    A_DEFFLOAT,
                    0);
    class_addmethod(dynamicTW_class,
                    (t_method)dtw_onSet_B,
                    gensym("ratio_B"),
                    A_DEFFLOAT,
                    0);
}
```