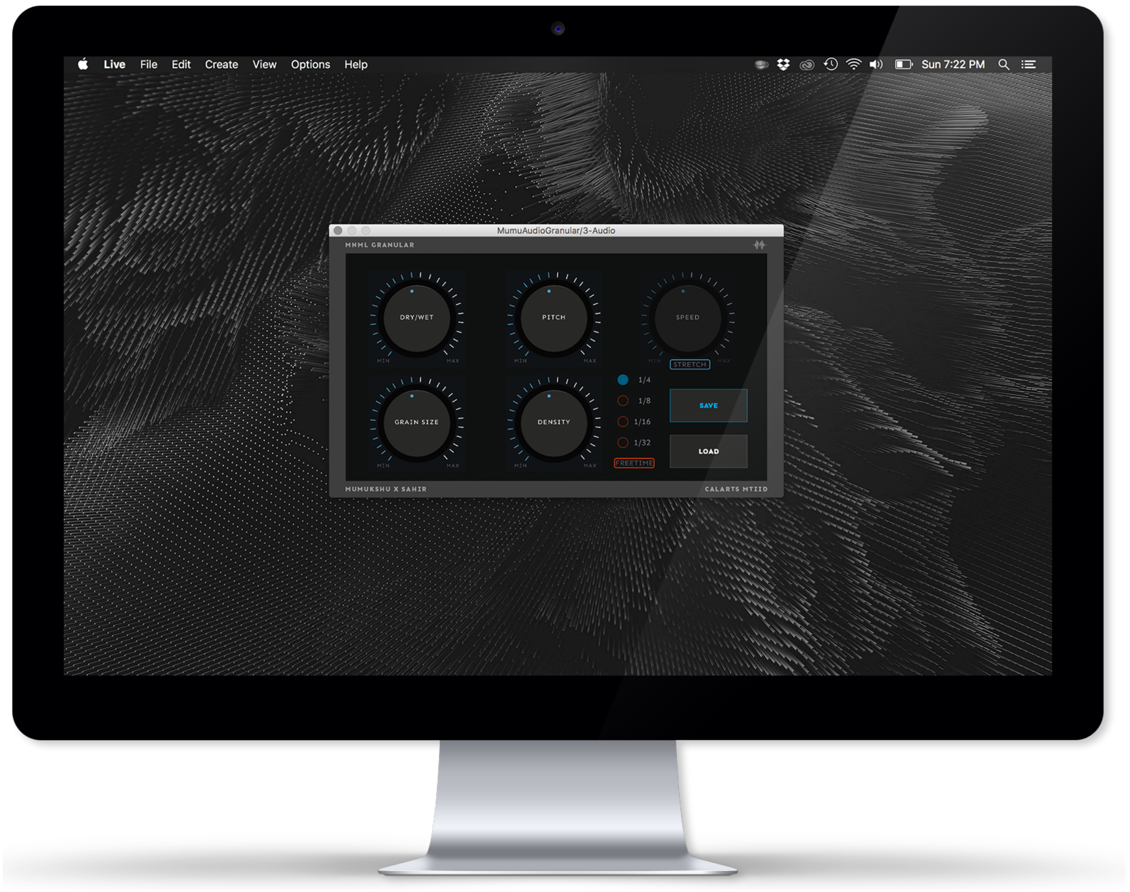
MNML GRANULAR & UNSUPERVISED SYNTHESIS PROGRAMMING



Jacob Penn - BFA Music Technology

Acknowledgments

Thank you to the entire MTIID faculty for inspiring and challenging the way I think and interact with the world.

Submitted to:

Ajay Kapur

Jordan Hochenbaum

Owen Vallis

Spencer Salazar

Parag Mital

Special thanks to:

My friends and family for all the support.

Nathan Shaw and my CalArts classmates for being so open to sharing your knowledge.

Curtis Roads & Ross Bencina for their detailed work on granular synthesis.

Abstract

Key topics: C++, Granular Synthesis, Digital Signal Processing, Machine Learning, Custom GUIs

Currently, digital audio systems require the manual tuning of hundreds of various parameters in order to achieve a target sound. This paper proposes a method to automate this process through the unsupervised programming of *MNML Granular,* a cross platform VST/AU plugin which delivers users a unique interface for the pitch-shifting, time-stretching, and rhythmic chopping of real time signals. Granular processing provides many interesting types of synthesis through it’s ability to separate the pitch and time domains. This paper offers insight into the structures of granular engines, as well as potential strategies and improvements for real time granular processing.

Contents

Introduction

Granular Synthesis - Background

Granular synthesis is a common digital signal processing technique which consists of the decomposition of an audio signal into small wavelets. This fine control of wavelets allows for the separation of pitch and time. Previous to granular synthesis the time and frequency domains were intrinsically connected for audio engineers, but with the speed of modern day computing and the strength of granular synthesis and filtration, any studio with a modern household computer can utilize these techniques. Although there is a multitude of granular synthesis systems on the market (e.g. Camel Audio, Ableton Live, Twisted Tools, Audio Mulch) there is room for individuality and customization in building a granular engine from scratch. All granular engines require the manual tuning of many parameters which will be exposed to the user, this paper proposes a method of granular processing focused on the manipulation of real time signal.

The history of granular techniques stem from scientific and philosophical thinking. *Atomism: a doctrine that the physical and mental universe is composed of simple indivisible minute particles.*[1] Atomism has been attributed as the root of granular thinking by granular pioneer Curtis Roads, in his paper *Evolution of Granular Synthesis*. He also states, "*The modern concept of sound particles can be traced to Einstein's phonons, which he predicted in 1907. But the phonons consist of inaudible packets of ultrasonic energy at feeble amplitudes. It was Einstein's pupil, Dennis Gabor, who in the 1940's had the fundamental insight that brought the particle model into the domain of perceived sound. The composer Iannis Xenakis learned of Gabor's experiments, and in 1959, he made an experiment in which he approximated granular synthesis by means of tape splicing.*"[2]

Iannis Xenakis was a Greek-French composer who helped revolutionize music after World War II. He is known for his pioneering approach to composition which included the fusion of mathematical processes with 20th century classical music. Although Xenakis rejected serialism, he contributed to this crucial period in musical development.[3]

Post World War II, composers were unable to understand what had happened in society, disgusted, they devised a radically new system for music composition based on using all twelve notes of the keyboard, the twelve tone matrix, which served as a scientific path to sonically interesting and pleasing results. Xenakis' approach differed from that of his fellow composers, as he rejected the twelve tone matrix and used the application of mathematical models to create a seemingly random composition out of very rigid and detailed instructions. Xenakis argued that allowing players to freely break rules with no guide wouldn't open a door to a new world, but instead limit performers to either fall back to their comfort zone, or alternatively to the polar opposite of the intuition. He created the first granular music composition by splicing many bits and pieces of tape string recordings in his pieces *Analogique A-B* in1958-1959.

The approach Xenakis used for granular synthesis, although very sonically interesting, proved to be too tedious and difficult for deep investigation. It wasn't until computers gained enough power to design digital granular systems that granular synthesis gained traction. Curtis Roads, a composer, computer programmer, and CalArts alum, is attributed to designing the first digital granular synthesis engine. Inspired by Xenakis, he built his first granular system in the 1970s and has continued to lay the ground work for other composers and engineers to explore granular synthesis.

Machine Learning & Synthesis - Background

"Machine learning is programming computers to optimize performance criterion using example data or past experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or past experience. The model may be predictive to make predictions in the future, or descriptive to gain knowledge from data, or both."[4]

Machine learning is a topic of great importance in computer science, and one whose importance will continue to grow with the complexity surrounding digital systems and data. In digital audio synthesis, sounds are sculpted and generated through the manual setting of hundreds of possible parameters. Although there is an incredible amount of fine detail

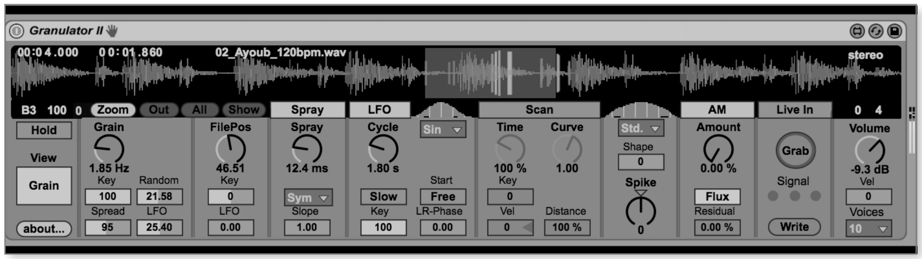


Fig. 1
Granulator II by Mono-lake

Fig. 2
Asynchronous Granulation

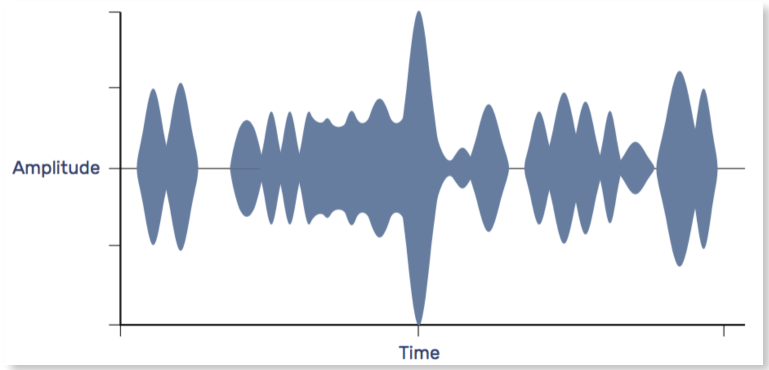
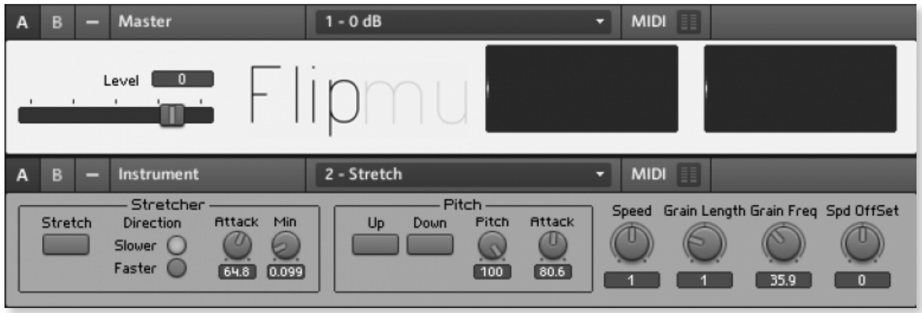


Fig. 3
Grain Stretch by FlipMu



which is available to the user, the creation of novel sounds quickly becomes a daunting task which requires deep consideration and expertise. Machine learning has the potential to aid composers through the synthesis process by very quickly changing and testing parameters in order to create target audio.

Review

Granular Synthesis - Related Work

Non-Real Time - *Granulator II* by Monolake

Granulator II by Monolake (Fig. 1) is a quintessential example of the strengths of non-real time granular synthesis. It delivers users with what is referred to as *asynchronous* granular processing, a common approach within granular synthesis environments. A possible output of an *asynchronous* granular system is an audio stream with inconsistent amplitude. (Fig. 2)

In this application, an audio file is loaded into the granular system to serve as the audio source in synthesis programming, the asynchronous system then plays wavelets from the source audio in user constrained irregular intervals, from many grains simultaneously. This type of granulation creates new sonic textures far from the original source material. As opposed to synchronous granular systems, asynchronous is a common approach in systems where sound synthesis is the goal. Because of the frequent use of granulation in this type of system, granular processing has become commonly referred to as granular synthesis. The granular source material has replaced the need of oscillators in the synthesis environment. In real time systems it is possible to refer to wavelet decomposition of sound as granular processing as opposed to granular synthesis.

Real Time - *Grain Stretch* by FlipMu

Grain Stretch by FlipMu (Fig. 3), on the other hand, is the perfect example of real time granular processing. This system delivers a *synchronous* granular stream (Fig. 4), which allows users to separate the pitch and time domains of real time signal. This is achieved by the granular systems ability to recreate audio signal from multiple playback sources simultaneously. Audio is streamed into the system, which delivers wavelet decomposition of the signal, and a user definable schema to re-synthesize the audio.

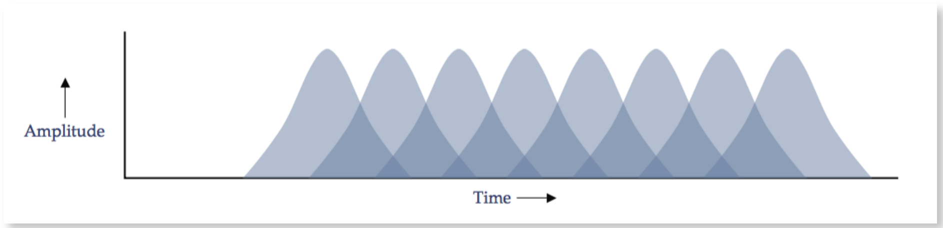


Fig. 4
Synchronous Granulation

Fig. 5
Granular Transposition

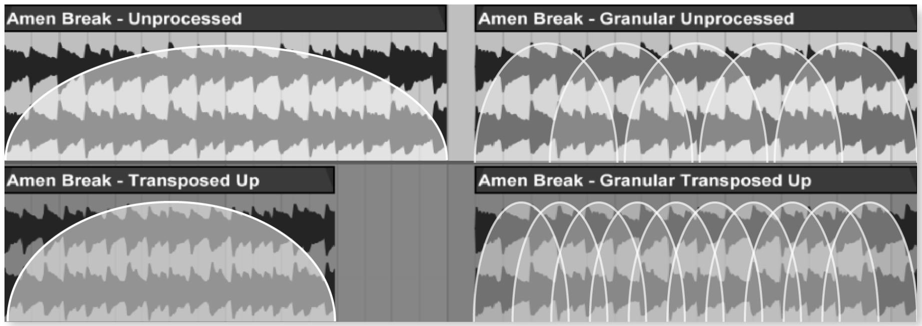
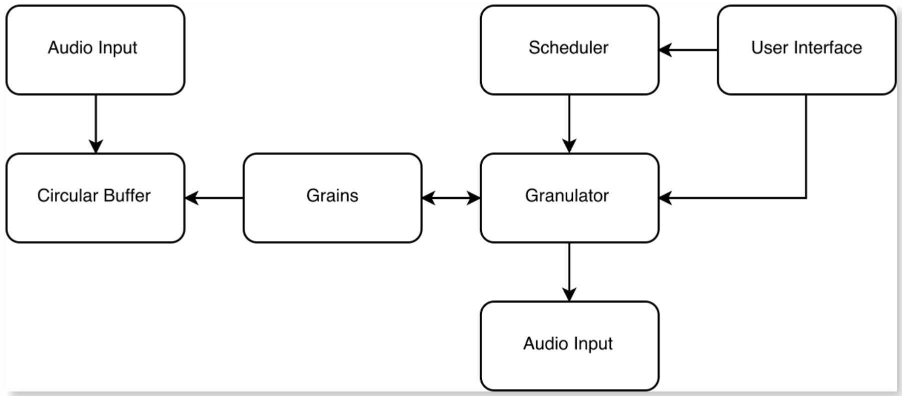

Amen Break - Unprocessed
Amen Break - Granular Unprocessed
Amen Break - Transposed Up
Amen Break - Granular Transposed Up

Fig. 6
Granular Structure



Most people are aware of the connection of the time and frequency domains through the use of record players. If a record is played back faster than it's intended speed, the pitch of the signal is raised, alternatively, if the record is played back slower, the pitch is lowered. Granular systems use this to their advantage by overlapping windowed versions of the signal in a manner which simultaneously raises or lowers the pitch, while keeping the time domain of the signal intact.

As you can see in the granular transposition on the right (Fig. 5), the time domain has remained intact to the original signal. This is because of the granular processing system's ability to overlap multiple transposed grains. In the transposed signal, transposed grains are played throughout the buffer of the original audio signal, overlapping and repeating pieces of audio to preserve the time domain.

Methods - Granular

Overview

The signal flow of a granular system requires four main pieces, a circular buffer for holding audio data from an input, a group of grains for retrieving audio data from the buffer, a granulator object for synthesizing audio from the grains, and a scheduler which triggers the granulators queries. First, audio is streamed into the system and into the buffer. Next, according the settings provided by the user, the scheduler and granulator drive the system, sending queries and synthesizing audio for the output.

Circular Buffers

The beginning of the project starts with the creation of a circular buffer to hold real time audio data. Circular buffers are common in digital audio applications, as they serve as the basis for many time based audio effects (e.g. Vibrato, Flanger, Chorus, Delay). A block of computer memory is pre allocated to allow the simultaneous writing and reading of audio data in the program. Because in *MNML Granular* there is no responsibility of the circular buffer to do anything except hold audio data, the circular buffer is quite simple, with no read heads. (Fig. 7) A pointer float array is dynamically allocated to allow variable delay line lengths, and the process function operates at sample rate to write into the circular buffer. (Fig. 8)

Fig. 7
GranularBuffer.h

```
class GranularBuffer {
public:
    GranularBuffer();
    ~GranularBuffer();
    void   setBufferLength(float sampleRate, float seconds);
    int    getBufferLength();
    int    getWriteIndex();
    void   prepareToPlay();
    void   process(float input);
    float* m_pBuffer;
private:
    int    m_nBufferSize;
    int    m_nWriteIndex;
};
```

Fig. 8
GranularBuffer.cpp - process() function

```
void GranularBuffer::process(float input)
{
    m_pBuffer[m_nWriteIndex] = input;
    m_nWriteIndex++;
    if(m_nWriteIndex > m_nBufferSize)
        m_nWriteIndex = 0;
}
```

Fig. 9
GranularBuffer.cpp - prepareToPlay() function

```
void GranularBuffer::prepareToPlay()
{
    if (m_pBuffer)
        delete [] m_pBuffer;
    m_pBuffer = new float[m_nBufferSize];

    if (m_pBuffer)
        memset(m_pBuffer, 0, m_nBufferSize*sizeof(float));

    m_nWriteIndex = 0;
}
```

Before playback, the pBuffer is preallocated and the *memset* function is called to initialize all it's values to zero. (Fig. 9) On playback each sample of audio is written into a new position in the circular buffer, if the write index of the buffer reaches the end of the available space in memory, the write index is reset to zero and the buffer begins to rewrite over previous audio data. In the granular system, the length of the delay line is important to the time stretching effect on the audio. If time stretching is applied for a period longer than the buffer length, the effect will be disrupted by a discontinuity due to the input of the circular buffer overwriting the desired audio data of the time stretch.

Grains

At the heart of a granular processing algorithm live the grains. The grains are responsible for generating audio data from the granular buffer and returning this data to the granulator object upon request. In *MNML Granular,* grains apply a Tukey Window function to the audio data in order to retrieve wavelets from the granular buffer. (Fig. 10, 11)

Notice the use of an external "playSwitch," which controls starting the execution of the window. (Fig. 12) The variable used for this function lives in the grain class, and is crucial to logistical playback. The most important function of the grains is to respond to the granular object's query to the granular buffer. This query involves the length of the grain, the position of the buffer the grain is to read from, and the pitch or playback speed of the grain. The grain object must also handle an interpolation type in order to estimate inter sample values, as well as be able to tell the granulator object whether or not it is currently busy generating audio data from another query. m_bBusy is the variable which the Tukey Window uses to determine whether or not to run through the function.

Notice the four grainIndices, these are used to control a quadratic interpolation of the audio signal. (Fig. 13) If playback speeds or the pitch of the grain are raised or lowered from 1, an interpolation method must be used in order to determine inter sample values of the audio signal. (Fig. 14)

The play function of the grain runs at sample rate, and is responsible for executing all the queries from the granulator object. (Fig. 15)

First, the granular buffer is passed into the grain object. The four points surrounding the desired sample value are indexed within the buffer and pulled into local variables of the function. The grainIndex is determined right before the beginning of the execution of the grain, through the init() function. (Fig. 16)

Pitch is a value which will determine the speed grains play through the granular buffer, 1 is equivalent to the true pitch of the sound, while .5 is half the frequency of the original, and 2 is double the frequency. Because this granular processor operates on a real time signal, keeping the beginning of the granular playback as close to the granular buffer write head is crucial. Considerations must be made when the playback speed of the grain are greater than one. Once the grain is aware of its playback speed, and the

Fig. 10
Tukey Window Equation

$$w(x) = \begin{cases} \frac{1}{2}\{ 1 + cos\ (\frac{2\pi}{r}[x - r/2\ ])\ \}, & 0 \leq\ x < \frac{r}{2} \\ 1, & \frac{r}{2} \leq x < 1 - \frac{r}{2} \\ \frac{1}{2}\{ 1 + cos\ (\frac{2\pi}{r}[x - 1 +\ r\ /2\ ])\ \}, & 1 - \frac{r}{2} \leq x \leq 1 \end{cases}$$

Fig. 11
Tukey Window Graphical Plot

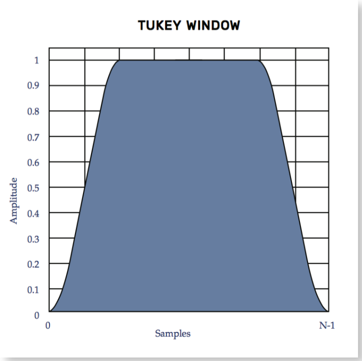


Fig. 12
TukeyWindow.cpp - doTukeyWindow() function

```
float TukeyWindow::doTukeyWindow(float input, bool& playSwitch){
    if (playSwitch == true){
        if (incrementer < (timeInSamples/2)){
            multiplier = 0.5 * (1+cos(((2*PI)/timeInSamples)*(incrementer-(timeInSamples/2))));
            incrementer++;
        }
        else if (incrementer == (timeInSamples/2)){
            multiplier = 1;
            incrementer++;
        }
        else if (incrementer > (timeInSamples/2)){
            multiplier = 0.5 * (1+cos(((2*PI)/timeInSamples)*((incrementer - 1)+(timeInSamples/2))));
            incrementer++;
        }
        if (incrementer == timeInSamples){
            multiplier = 0;
            playSwitch = false;
            incrementer = 0;
        }
        return input * multiplier;
    }
}
```

Fig. 13
Grain.h

```
class Grain
{
public:
    Grain();
    ~Grain();
    void setDelta(float samples);
    void setWindowSize(int sampleRate, float seconds);
    void init(float pitch, GranularBuffer& buffer);
    bool isItBusy();
    void setGrainBusy(bool isItBusy);
    float play(int sampleRate, GranularBuffer& buffer);
    int returnIncrementer();

private:
    bool m_bBusy;

    TukeyWindow m_Window;
    float m_fDelta;
    float m_fWindowSize;

    float m_fGrainIndex;
    int   m_nGrainIndex_f1;
    int   m_nGrainIndex_f2;
    int   m_nGrainIndex_p1;

    float m_fBufferSpeed;
};
```

Fig. 14
Four Point Intersample Interpolation

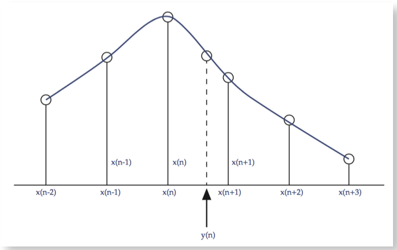


Fig. 15
Grain.cpp - play() function

```
float Grain::play(int sampleRate, GranularBuffer&
buffer)
{
        float yn = buffer.m_pBuffer[point];
        float yn_f1 = buffer.m_pBuffer[f1point];
        float yn_f2 = buffer.m_pBuffer[f2point];
        float yn_p1 = buffer.m_pBuffer[p1point];

        float fFracDelay = grainIndex – (int)grainIndex;

        float output = FourPointInterpolate(yn, yn_p1,
                           yn_f1, yn_f2, fFracDelay);

        grainIndex = grainIndex + bufferSpeed;

        if (grainIndex > buffer.getBufferLength())
        {
            grainIndex -= buffer.getBufferLength();
        }

        return m_Window.doTukeyWindow(output, m_bBusy);

}
```

Fig. 16
Grain.cpp - init() function

```
void Grain::init(float pitch,
GranularBuffer& buffer)
{
    bufferSpeed = pitch;
    grainIndex =
          buffer.getWriteIndex() –
                            m_fDelta;
    if (grainIndex < 0)
    {
        grainIndex +=
          buffer.getBufferLength();
    }
}
```

position in the buffer, it is ready to be triggered and return the signal. The inter sample amount of the particular grain's read head within the granular buffer is calculated through *fFracDelay = grainIndex - (int)grainIndex*. The output sample of the system is then created through the four point interpolation of the four audio samples, using the fractional inter sample amount to find the desired point. The grainIndex is then incremented through the circular buffer according the "pitch" of the grain. At this point, we have our desired output sample, however it is still at the same amplitude as the input signal. Before returning to the granulator, the grain applies the window function to the audio data according to the window size specified from the granulator, and whether or not to run through the window. If *m_bBusy* is false, the return will always be 0, if the granulator object sets the grains *m_bBusy = true*, then the grain will start at the beginning of the window, return windowed audio data, reset it's state of *m_bBusy* to false, and resume returning zero.

Scheduler

The scheduler is completely unaware of the granular synthesis which is occurring, and functions solely as a timer to drive granular playback within the granulator. It's functionality is very simple, as it essentially acts like a sample accurate timer in the audio space. (Fig. 17)

The *setInteronset* function takes in the user controlled inter-onset rate of the grains, because this is a synchronous granular system, the inter-onset time has no randomization and will always deliver a consistent stream of triggers specified by the user interface. (Fig. 18) The granulator object will drive the scheduler through it's play function, and can query it's state through the bang function. (Fig. 19, 20)

The bang function also holds the functionality to force a grain to fire, which is later needed to drive tempo synchronous streams.

Fig. 17
GrainScheduler.h

```
class GrainScheduler
{
public:
    GrainScheduler();
    ~GrainScheduler();
    void setInteronset(float sampleRate, float seconds);
    void prepareToPlay();
    void play();
    bool bang();
    void forceBang();
private:
    int  m_nInteronsetInSample;
    int  m_nCounter;
    bool m_forceBang = false;
};
```

Fig. 18
GrainScheduler.cpp - setInteronset() function

```
void GrainScheduler::setInteronset(float sampleRate, float seconds){
    m_nInteronsetInSample = sampleRate * seconds;
}
```

Fig. 19
GrainScheduler.cpp - play() function

```
void GrainScheduler::play(){
    m_nCounter++;
    if (m_nCounter > m_nInteronsetInSample)
    {
        m_nCounter = 0;
    }
}
```

Fig. 20
GrainScheduler.cpp - bang() function

```
bool GrainScheduler::bang(){
    if (m_forceBang == true)
    {
        m_forceBang = false;
        return true;
    }
    else if (m_nCounter == m_nInteronsetInSample)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Granulator

The granulator is the highest level object within the granular system. Its responsibility is to coordinate all controls between the user UI, the grain scheduler, the granular buffer, and the the grains themselves. In *MNML Granular* the entire plugin itself is partially responsible for handling all of this functionality, but there is a high level granulator object which abstracts the process block of the granulation. The granulator object takes all of this data from *MNML Granular* and synthesizes the output. (Fig. 21)

First the system needs member variables of all necessary parts of the granular system. (Fig. 22)

Next the plugin needs to initialize all of these objects and prepare them for playback. This happens on initialization of the system, or anytime sample rate is changed. (Fig. 23)

Once all of the objects are instantiated and ready for granular playback, the processBlock method can begin to process audio data through the granulator based on user controlled parameters. The processBlock function takes audio samples into the system, passes them through the granulator, and then back out to the output. (Fig. 24)

This is the first type of granulation which the granulator object can perform. (Fig. 25) Input data is read into the system, and then played back in the most real time possible with modulated amplitude and pitch, the time domain of the signal remains the same as the original. Stepping through this function we see how the logic of the granular system operates. First, the system determines the schema for the delta offset calculation is that of real time operation (stretching = false). Next, the system determines whether or not a *schedulerBang* has occurred. If no bang has occurred, the granulator need not perform any action at this point. If the bang is true, the granulator parses through the available grains it has at its disposal. It finds the first available grain, and then initializes it with the user specified settings. First, the window size of the grain is taken from the user interface, and passed into the grain. Next, the position of the grain in the buffer is determined. If the grain's playback pitch is <= 1, the delta offset may remain zero, as the read action occurs after the write action of the granular buffer, and grains can be executed in fully real time. If the pitch of the grain is > 1, then this system uses a method to keep the grain as close to the read head as possible.

Fig. 21
Granulator.h

```
class Granulator
{
public:
    Granulator();
    ~Granulator();

    float process( Grain* grains, GranularBuffer &buffer, bool stretching, bool
schedulerBang, float numGrains, float sampleRate, float grainSize, float pitch,
float stretchSpeed);

private:
    bool  m_bState;
    float m_fStretchDelta;
    bool  m_bStretchStarted;
```

Fig. 22
PluginProcessor.h

```
Granulator m_GranulatorL;
Granulator m_GranulatorR;

GranularBuffer m_gBufferL;
GranularBuffer m_gBufferR;
const float m_fMaxBufferLength = 10.0;

GrainScheduler m_SchedulerL;
GrainScheduler m_SchedulerR;

const int m_nNumberGrains = 6;
Grain* m_GrainP_ArrayL;
Grain* m_GrainP_ArrayR;
```

Fig. 23
PluginProcessor.cpp - prepareToPlay() function

```
void MNMLGranularAudioProcessor::prepareToPlay (double sampleRate, int
samplesPerBlock)
{
    m_GrainP_ArrayL = new Grain[m_nNumberGrains];

    m_GrainP_ArrayR = new Grain[m_nNumberGrains];

    m_gBufferL.setBufferLength(sampleRate, m_fMaxBufferLength);
    m_gBufferL.prepareToPlay();

    m_gBufferR.setBufferLength(sampleRate, m_fMaxBufferLength);
    m_gBufferR.prepareToPlay();

    m_SchedulerL.prepareToPlay();
    m_SchedulerR.prepareToPlay();
}
```

Fig. 24
PluginProcessor.cpp - processBlock() function

```
for ( int i = 0; i < buffer.getNumSamples(); i++ )
{
      if (channel == 0)
      {
        m_gBufferL.process(channelData[i]);
        m_SchedulerL.play();
        const bool bangL = m_SchedulerL.bang();
        const float output = m_GranulatorL.process(grainp_ArrayL,
                                                   m_gBufferL,
                                                   stretchButtonStateScaled,
                                                   bangL,
                                                   m_nNumberGrains,
                                                   m_fSampleRate,
                                                   grainSizeScaled,
                                                   pitchScaled,
                                                   stretchSpeedScaled );


        channelData[i] = (output * Tab1_dryWetScaled) + ((1 -
                                              Tab1_dryWetScaled) * channelData[i]);
      }
}
```

Fig. 25
Granulator.cpp - process() function (no time-stretching)

```
if (!stretching)
{
   if (schedulerBang)
   {
        for (int i = 0; i < numGrains; i++)
        {
            if(grains[i].isItBusy() == false)
            {
                grains[i].setWindowSize(sampleRate, grainSize);
                float delta = 0.0;
                if (pitch >= 1.0)
                {
                    delta = (pitch-1.0)*(grainSize*sampleRate);
                }
                grains[i].setDelta(delta);
                grains[i].init(pitch, buffer);
                grains[i].setGrainBusy(true);
                break;
            }
        }
    }
}
```

Delta = (pitch-1.0)(grainSize*sampleRate);*

This calculation ensures that the last sample of the windowed audio data will land back onto the write head of the granular buffer, with minimal time offset. This works by determining the amount in samples which the grain would play past the write head, and setting it back in the buffer accordingly to ensure that no audio discontinuities occur by reading past the write head. (Fig. 26)

After the grain's delta offset is calculated, and the grain is initialized with its position in the buffer, it's set busy. When the grain is set busy it begins applying its window function to its set position in the audio buffer. The other potential mode for the granular engine is that of time stretching. (Fig. 27)

In this method of granular processing, the delta offset of the grains position in the buffer is controlled by a speed factor which is determined in the user interface. A bool, stretchStarted, keeps track of the onset of stretching mode. If stretching mode becomes true, stretchDelta is initialized to 0, and the delta begins to grow and wrap around the buffer. This exponential growth of the offset is what allows the separation of pitch and time, allowing the pitch of the signal to stay intact while audibly the sound can be slowed down. Multiple grains of the same pitch are played in a density which allows sustained amplitude of the origin signal, but moves through the buffer more slowly than real time.

The final function of the granulator is to stitch back together the audio output of all the grains into a single stream. This is done simply by summing the output of all of the grains together. (Fig. 28) Only grains which are currently active will contribute to the generation of this audio data, as any grain which is inactive will return a zero and not affect the computation.

Fig. 26
Effect of Delta Offset Calculation

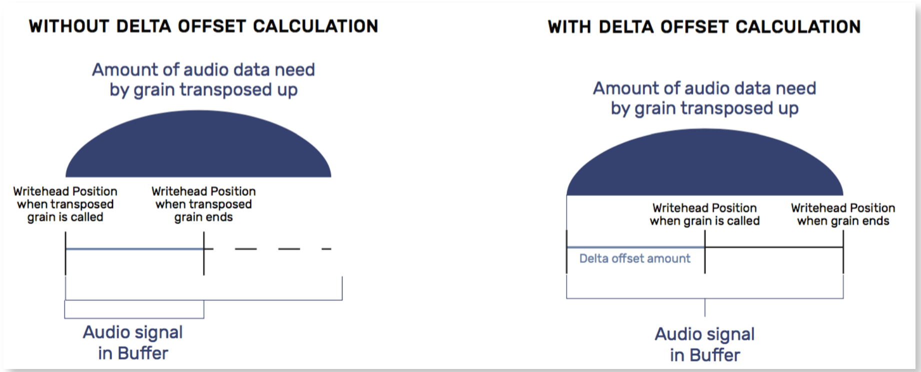

WITHOUT DELTA OFFSET CALCULATION
Amount of audio data need by grain transposed up
Writehead Position when transposed grain is called
Writehead Position when transposed grain ends
Audio signal in Buffer
WITH DELTA OFFSET CALCULATION
Amount of audio data need by grain transposed up
Writehead Position when grain is called
Writehead Position when grain ends
Delta offset amount
Audio signal in Buffer

Fig. 27
Granulator.cpp - process() function (time-stretching)

```
if (stretching)
{
    if (!stretchStarted){
        stretchDelta = 0;
        stretchStarted = true;
    }
    stretchDelta += stretchSpeed;
    if (schedulerBang == true){
        for (int i = 0; i < numGrains; i++){
            if(grains[i].isItBusy() == false){
                grains[i].setWindowSize(sampleRate, grainSize);
                grains[i].setDelta(stretchDelta);
                grains[i].init(pitch, buffer);
                grains[i].setGrainBusy(true);
                break;
            }
        }
    }
}
```

Fig. 28
Granulator.cpp - process() function

```
float output = 0;
for (int i = 0; i < numGrains; i++)
{
    output += grains[i].play(sampleRate, buffer);
}
```

Fig. 29
PluginProcessor.cpp - processBlock() function (before granulation)

```
AudioPlayHead::CurrentPositionInfo posInfo;
updateBpm(posInfo);
```

Fig. 30
PluginProcessor.cpp - updateBPM() function

```
void MNMLGranularAudioProcessor::updateBpm(CurrentPositionInfo posInfo)
{
    AudioPlayHead* playHead = getPlayHead();
    playHead->getCurrentPosition (posInfo);

    currentBPM = posInfo.bpm;
    ppqPosition = posInfo.ppqPosition;

    double timeSigNumerator = posInfo.timeSigNumerator;

    downBeatCounterZ = downBeatCounter;
    downBeatCounter = fmod(ppqPosition, timeSigNumerator);
}
```

Fig. 31
PluginProcessor.cpp - processBlock() function (before granulation)

```
updateParams();

if (!freetimeSynctimeParam->getValue())
{
    m_SchedulerL.setInteronset(m_fSampleRate, Tab1_densityScaled);
    m_SchedulerR.setInteronset(m_fSampleRate, Tab1_densityScaled);
} else if (freetimeSynctimeParam->getValue()) {
    m_SchedulerL.setInteronset(m_fSampleRate, Tab2_currentTempoSyncNumerator/
                                                              currentBPM);
    m_SchedulerR.setInteronset(m_fSampleRate, Tab2_currentTempoSyncNumerator/
                                                              currentBPM);
    }
    if (freetimeSynctimeParam->getValue())
    {
        if (downBeatCounter < downBeatCounterZ)
        {
            m_SchedulerL.prepareToPlay();
            m_SchedulerL.forceBang();
            m_SchedulerR.prepareToPlay();
            m_SchedulerR.forceBang();
        }
    }
```

Rhythmic Tempo Syncing

One of the strengths of *MNML Granular* is its ability to sync to the host tempo. On the UI of the plugin, there is a switch which activates either a free time or a syncopated time mode. If syncopated timing is selected, the functionality of the GrainSchedulers are adjusted in order to deliver tempo synced triggering of the grains. This is handled in the process block before the "bang" of the scheduler is passed into the granulator. (Fig. 29)

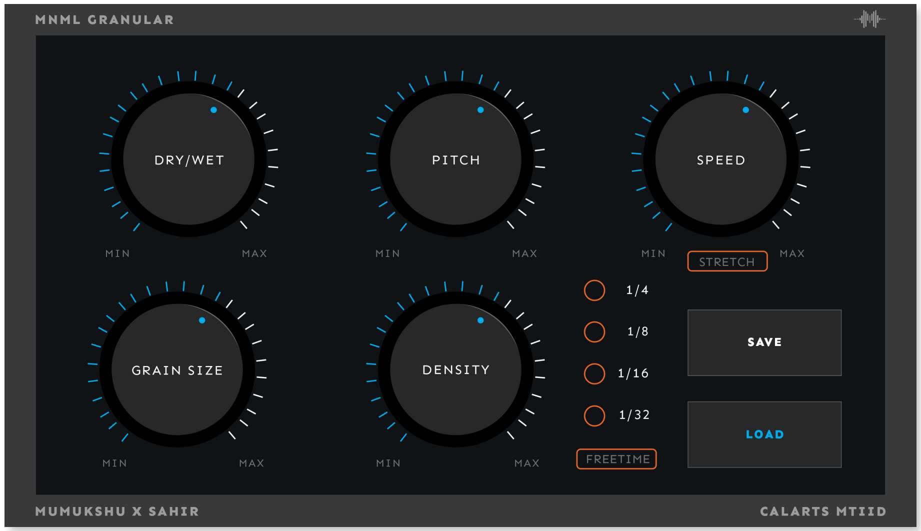
The current position of the playhead inside of the DAW is pulled into the plugin and passed into an updateBPM() function. (Fig. 30)

Keeping track of timing inside of digital audio systems between blocks can be a difficult challenge. In this system, the *downBeatCounter* is never guaranteed to equal 0 at any point, even if the playhead rolls across the downbeat. The easiest solution to syncing grains to the downbeat is to check whether the current block of audio data's downbeat counter is less than the previous one, this means that the audio playhead has rolled over the downbeat inside the DAW, and a grain should play during this block. (Fig. 31)

The updateParameters() function grabs the current sync rate and timing mode the user has selected in the UI. If free time is selected, the inter onset rate of the granular system is set to the value of the density knob in the UI. If sync time is selected, the system pulls the "currentTempoSyncNumerator" and divides this by the currentBPM to find the seconds per beat, it then multiplies this against the sampleRate in order to determine the inter-onset time in samples.

If the system is in synced time mode, it continuously syncs to the down beat of the DAW, ensuring that grains stay synced to the host tempo. The prepare to play function resets the index of the counter to 0, while the forceBang() function guarantees a single grain will be triggered on this down beat. The counter then continues operating as expected and then re locks to the host BPM on the next down beat.

Fig. 32
MNML Granular User Interface



User Interface

The end goal of *MNML Granular* is to deliver users an enjoyable experience while using the system, for this reason a custom UI was created for the plugin, which is meant to be intuitive and easy on the eyes. (Fig. 32)

Conclusion of Granular Methods

The methodology proposed in this section has outlined a robust and flexible system of granular synthesis, which could be applied to both real and non-real time systems. The goal of this project was to explore digital signal processing in depth, as well as gain an understanding of multi-threaded audio/UI applications. The source code of *MNML Granular* has been adopted into research for the machine learning of synthesis parameters, the methods of which will now be discussed.

Fig. 33
SelfProgrammer Application Structure

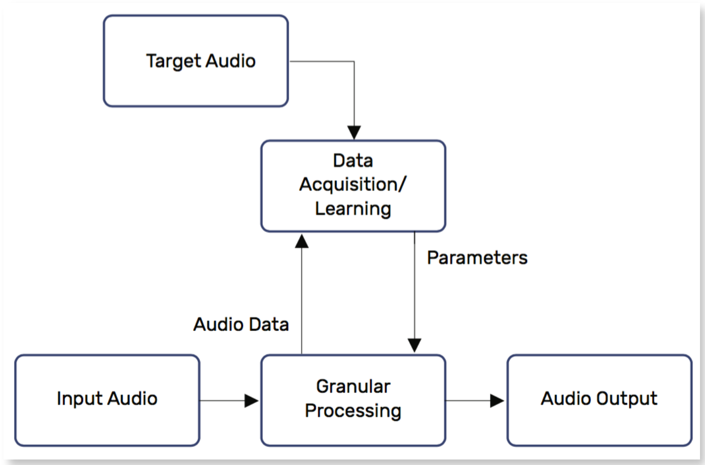
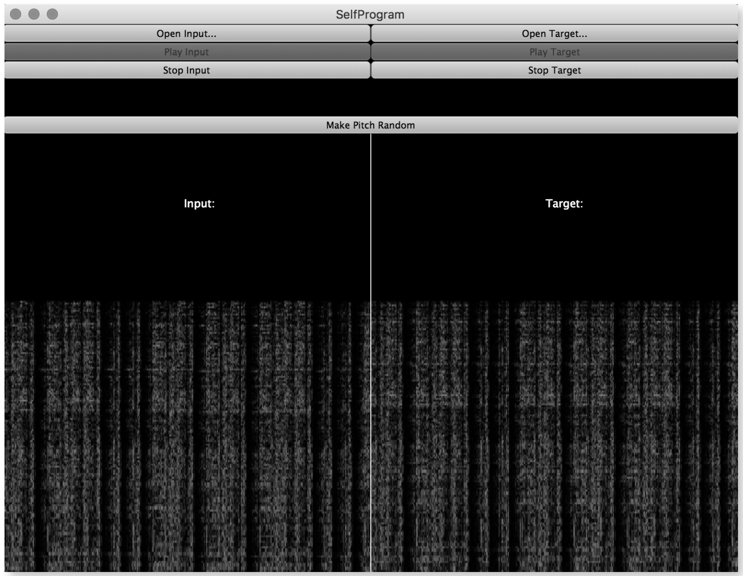


Fig. 34
SelfProgrammer User Interface



Methods - Machine Learning

Overview

SelfProgrammer is a standalone real time machine learning for audio application. It uses the source code of *MNML Granular* in order to explore the possibility of machine learning synthesis parameters. Currently, very complex synthesis structures are capable of creating a seemingly unlimited amount of sounds, however, hundreds of parameters must be manually programmed to reach a target sound. *SelfProgrammer* delivers a highly controlled environment to test the ability of machine learning for digital audio systems, and success in this environment could prove hopeful for success in more complex systems.

The structure of *SelfProgrammer* is that of an optimization algorithm. A target audio file is loaded into the system, as well as an input. The input is passed through the *MNML Granular* system. (Fig. 33, 34)

Data Acquisition

SelfProgrammer uses FFT analysis in order to gather information about the input and target audio file. (Fig. 35) FFT (Fast Fourier Transform) is an algorithm used to compute the of DFT (Discrete Fourier Transform) of an audio signal. This calculation transforms a set of discrete sample points in time and space, into a representation of the signal in the frequency domain. This conversion delivers *SelfProgrammer* the amplitude of signal across the frequency spectrum, which gives good insight into the qualities of a waveform.

After both an input and a target audio file are loaded into the system, the system begins working when the user presses "start input." First, the buffer of the target audio file is passed into a ring buffer for performing FFT. (Fig. 36)

After being passed into the ring buffer using *pushNextSampleIntoFifoTarget*, the buffer is also copied into a second buffer, *bufferToLearn*, this will be used in between calls to getNextAudioBlock(), in the data acquisition and learning section of the program. After the target audio data has been passed to it's own FFT buffer, the same is done for the

Fig. 35
DFT Equation

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-$$

Fig. 36
MainComponent.cpp - getNextAudioBlock() function (target signal)

```
targetTransportSource.getNextAudioBlock (bufferToFill);
for (int channel = 0; channel < numChannel; channel++)
{
    float* data = bufferToFill.buffer->getWritePointer(channel);
    for (int i = 0; i < numSamples; i++)
    {
        if (channel == 0)
        {
            pushNextSampleIntoFifoTarget (data[i]);
        }
    }
}
bufferToLearn->setSize(2, numSamples);
bufferToLearn = bufferToFill.buffer;
```

Fig. 37
MainComponent.cpp - pushNextSampleIntoFifo() function

```
void pushNextSampleIntoFifo (float sample) noexcept
{
    if (fifoIndex == fftSize)
    {
        if (! nextFFTBlockReady)
        {
            zeromem (fftData, sizeof (fftData));
            memcpy (fftData, fifo, sizeof (fifo));
            nextFFTBlockReady = true;
        }

        fifoIndex = 0;
        }
    fifo[fifoIndex++] = sample;
}
```

input audio source, however, the input audio source is run through the *MNML Granular* algorithm before it's sent to the FFT. The input is also passed to the speaker output so the user can hear the effect of the granular. The ring buffer keeps the FFT data blocks at the proper lengths, a power of two, which allows for optimized FFT processing. (Fig. 37)

The schema for the audio processing in this environment is the same as that in *MNML Granular,* however in this environment there is no user control over the synthesis parameters. (FIg. 38) Instead, the computer must determine the ideal settings of the granular system in order to minimize error between the input and target audio datas. This processing starts when runFFT() is called at the end of this audio block, after the input audio data has also been copied into a separate buffer.

Fig. 38
MainComponent.cpp - pushNextSampleIntoFifo() function

```
for (int channel = 0; channel < numChannel; channel++)
{
    float* data = bufferToFill.buffer->getWritePointer(channel);
    for (int i = 0; i < numSamples; i++)
    {
        if (channel == 0)
        {
            m_gBufferL.process(data[i]);
            m_SchedulerL.play();
            const bool bangL = m_SchedulerL.bang();

            data[i] = m_GranulatorL.process( grainp_ArrayL, m_gBufferL,
                                             0, bangL, m_nNumberGrains,
                                             m_fSampleRate, grainSize,
                                            grainPitch, 1 );

            pushNextSampleIntoFifo (data[i]);
         }
    }
}

bufferToProcess->setSize(2, numSamples);
bufferToProcess = bufferToFill.buffer;
runFFT();
```

Error Calculation

The heart of the learning algorithm is the error calculation. The error is used to determine whether the learning is working, and drive the system through it's logistical checks. As stated before, in *SelfProgrammer* the FFT data from two audio signals are compared to determine changes in parameters. (Fig. 39) The squared error along all frequency bins are summed and divided by the amount of bins to determine an average difference of the audio signals. (Fig. 40)

After the error is gathered for the system, the application makes it's next decision regarding learning. (Fig. 41) The bool *learnedIt* controls the flow of the system. If the error is found to be above a certain threshold, *learnedIt* is set false and the system is initialized. *numLearningAttempts* is set to zero, and parameter optimization begins.

Fig. 39
MainComponent.cpp - runFFT()

```
if (nextFFTBlockReady && nextFFTBlockReadyTarget)
{
    nextFFTBlockReady = false;
    forwardFFT.performFrequencyOnlyForwardTransform (fftData);
    drawNextLineOfSpectrogramInput();

    nextFFTBlockReadyTarget = false;
    forwardFFT2.performFrequencyOnlyForwardTransform (fftDataTarget);
    drawNextLineOfSpectrogramTarget();

    error = 0;
    for (int i = 0; i < fftSize/2; i++)
    {
      error += ((fftData[i] - fftDataTarget[i]) * (fftData[i] -
                                                   fftDataTarget[i]));
    }
    error = error/(fftSize/2);
}
```

Fig. 40
Sum of Squared Error Equation

$$SSE=\sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

Fig. 41
MainComponent.cpp - runFFT() function

```
if (error > 50)
{
    param = (rand() % 1000) / 500.0;
    learnedIt = false;
    numLearningAttempts = 0;
}
```

Fig. 42
MainComponent.cpp - runFFT()

```
while(!learnedIt)
{
    accelerateLearn();

    if (nextFFTBlockReadyTarget && nextFFTBlockReadyInput)
    {
        nextFFTBlockReadyTarget = false;
        forwardFFT2.performFrequencyOnlyForwardTransform (fftDataTarget);
        drawNextLineOfSpectrogramTarget();

        nextFFTBlockReadyInput = false;
        forwardFFT.performFrequencyOnlyForwardTransform (fftDataInput);
        drawNextLineOfSpectrogramInput();

        error = 0;
        for (int i = 0; i < fftSize/2; i++)
        {
           error += ((fftDataTarget[i] - fftDataInput[i])*(fftDataTarget[i] -
                                                     fftDataInput[i]));
        }
        error = error/(fftSize/2);

        if(deltaParamChange < 0.001 && error > 1 && numLearningAttempts < 100)
        {
            param = (rand() % 1000) / 500.0;
            numLearningAttempts++;
        }
        else if ((deltaParamChange < 0.001 && error < 1)
                                         || numLearningAttempts > 100)
        {
            learnedIt = true;
        }
        else
        {
            numLearningAttempts++;

            float change = ((error - errorZ) / deltaPitchChange) * 0.0001;
            param -= change;

            deltaParamChange = (param - parameterZ);
        }
        errorZ = error;
        parameterZ = param;
    }
}
```

Parameter Optimization

Once it is established that the error in the system is great, and that optimization must occur, the application enters an optimization loop, recurring until the demands of the system are met. (Fig. 42)

Entering the optimization loop, the function *accelerateLearn* is called. This function reprocesses the copied audio blocks of the current buffers with changes to the parameters. Every time the FFT of the newly reprocessed audio block is run and error is recalculated, parameter optimization occurs.

From this point, the system makes a new decision on the best course of action. If the difference between the last parameter and the current parameter is less than 0.001, but the error is greater than 0.1, this suggests that we have hit a local optimization, and the parameter should be reinitialize to begin optimization from a different point.

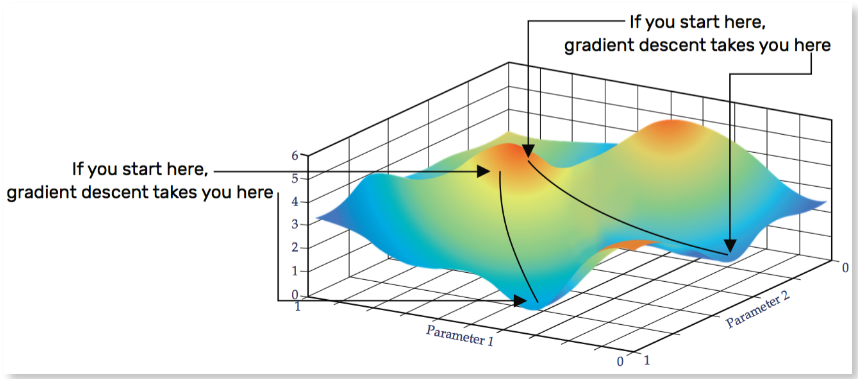
If the change in parameter is less than 0.001, and error is < 0.1, the learning has been successful, the system exits and audio is passed to the output buffer. Alternatively, if the learning has run over 100 attempts, the system also exits and passes the buffer to the output. The cap of learning attempts exists because of our real time audio system. If the too much optimization is processed in between blocks audio dropouts occur due to the demanding speed of audio playback. The best approach is to allow the audio buffer to pass and re-attempt the learning with the next block.

If the system is not satisfied, another round of optimization occurs, using gradient descent.

Fig. 43
MainComponent.cpp - runFFT()

$$\theta = \theta - \Delta\varepsilon/\Delta\beta) * \alpha$$

Fig. 44
Gradient Descent in Error Space



Gradient Descent

Gradient descent is an optimization algorithm used in machine learning applications in order to quickly navigate through functions in error space. (Fig. 43, 44) It computes changes in error with respect to changes in the parameter and updates the parameter in order minimize this error. Iteratively, *theta* or the parameter, is updated to be equal itself minus the difference of the error between the current and previous buffer, divided by the difference of the current and previous parameter, multiplied by a scalar for controlling speed of descent.

Conclusion of Machine Learning Methods

The methodology proposed in this section has outlined a system for the machine learning of synthesis parameters. The system has been testing in both single and multiple parameter settings, which will be discussed in the results.

Results - Granular

In the use case of *MNML Granular,* musicians are embracing the addition of rhythmic syncopation and chopping to granular processing.

JP Yépez, a creative technologist and educator with an MFA in Music Technology stated, "*It's clear that there is a disconnection between developers of musical applications and the needs of electronic musicians. The addition of tempo syncopation inside of MNML Granular offers composers a highly musical experience. Where many audio applications focus solely on signal processing, MNML Granular offers a simple addition to granular synthesis which can make a large impact on the workflow of musicians.*"

MNML Granular demonstrates multiple useful audio processing techniques. The original input signal is seen to have a constant amplitude. (Fig. 45) Through *MNML Granular* we see the addition of various types of processing. First, eighth note syncopation shows a very strong correlation to the rhythmic time markers in the DAW. (Fig. 46) Next, we see the input signal transposed an octave up in free time mode. We're able to see arhythmic control of the grains, as well as retain a constant amplitude which is the transposed version of the input signal with the same time domain. (Fig. 47) The last figure shows the stretch feature of the plugin, which once instantiate is able to slow down as well as grab and repeatedly play certain parts of the buffer. (Fig. 48)

Fig. 45
Original Input Signal

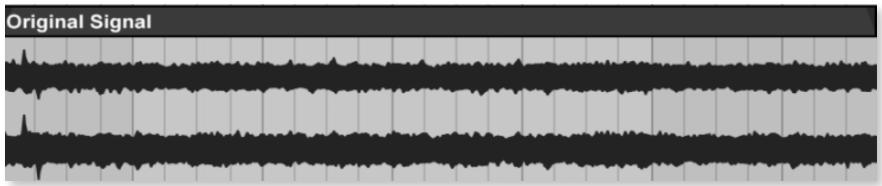


Fig. 46
Eighth Note Syncopation

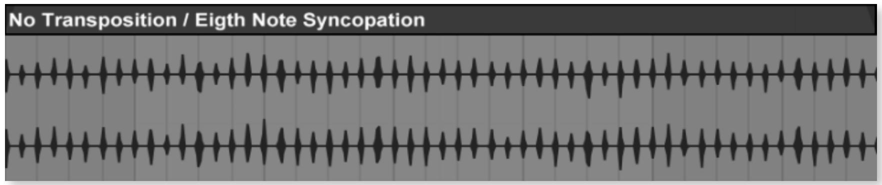


Fig. 47
Transposed One Octave Up / Free Time Syncopation

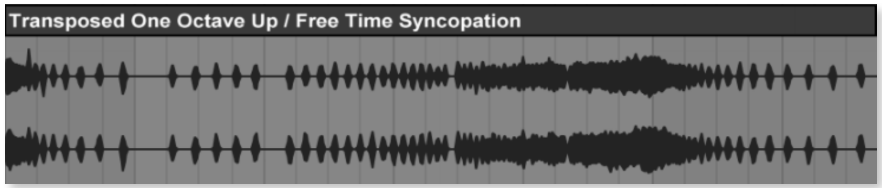


Fig. 48
Syncopated Signal Rhythmically Chopped and Frozen/Stretched

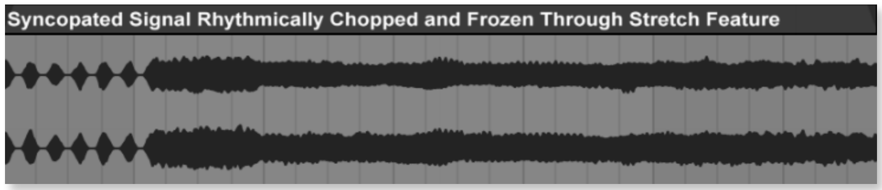

Syncopated Signal Rhythmically Chopped and Frozen Through Stretch Feature

Fig. 49
Granular Settings Unoptimized Input and Target (Single Parameter)

```
Default settings of SelfProgrammer:         Settings needed to match target:

        Grain Size: 0.120281                        Grain Size: 0.120281
        Density:    0.067375                        Density:    0.067375
        Pitch:      1.000000                        Pitch:      2.000000
```

Fig. 50
Unprocessed Input and Target Spectrograms

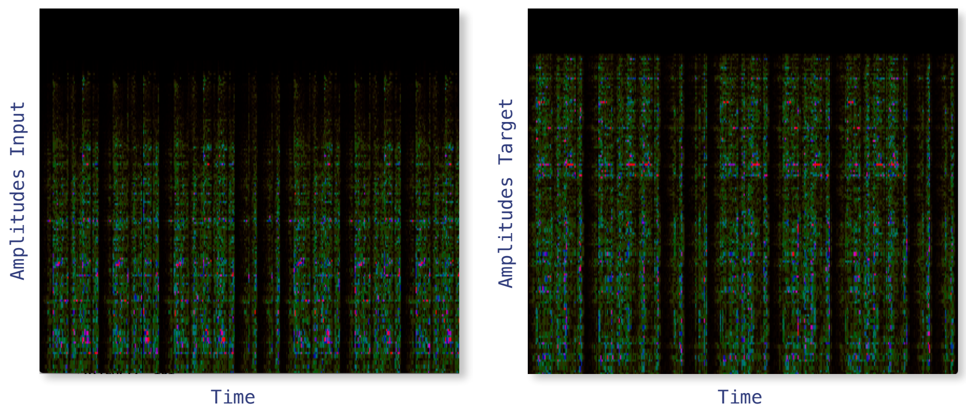
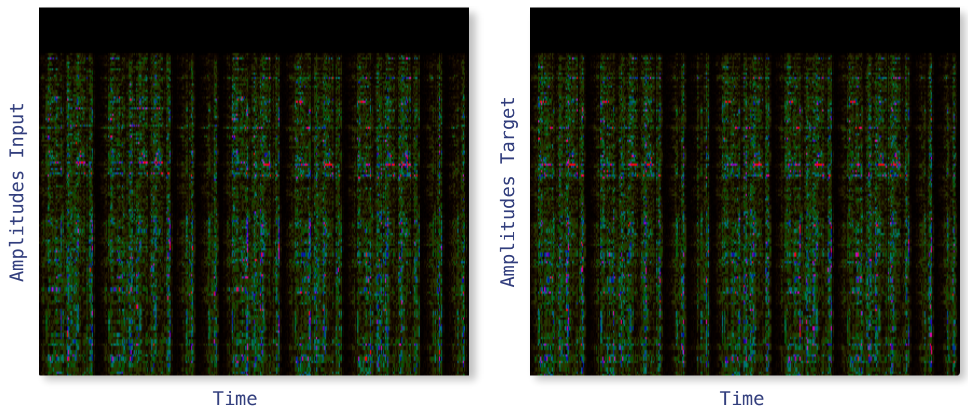


Fig. 51
Processed Input and Target Spectrograms



Results - Machine Learning

Single Parameter

In the first test of the machine learning system, a single parameter is optimized. First, a drum loop is preprocessed through the granular system, it shares all of the same parameters with what will be the input signal, except one, *pitch*. (Fig. 49)

Visualizing the spectrogram of the unprocessed input and target we see a similar signal, however, in the target there is a clear shift of the strongest amplitudes due to the pitch shifting incurred from the granular processing. (Fig. 50)

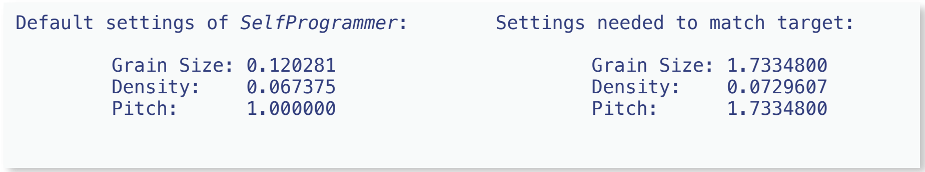
After processing the input through the machine learning system, we see the same shift in the input signal. (Fig. 51) Upon examination of the parameter learned within the system, in the spectrogram displayed, the pitch parameter is found to be 2.01527.

Fig. 52
Table of Machine Learning Results (Single Parameter)

Attempt	1	2	3	4	5	6	7	8	9	10
Result	2.00069	1.97000	1.06345	1.99800	1.90975	1.81067	1.10454	2.05039	1.94350	2.01324

Upon ten runs through the granular system, (Fig. 52) the pitch parameter is consistently able to match that of the target. Aside from a 20% possiblity for the machine learning system to confuse octaves of the signal, the success of the single parameter suggests that the machine learning system is performing as expected and can be tested upon multiple parameters.

Fig. 53
Granular Settings Unoptimized Input and Target (Multiple Parameters)



```
Default settings of SelfProgrammer:        Settings needed to match target:

        Grain Size: 0.120281                       Grain Size: 1.7334800
        Density:    0.067375                       Density:    0.0729607
        Pitch:      1.000000                       Pitch:      1.7334800
```

Fig. 54
Unprocessed Input and Target Spectrograms

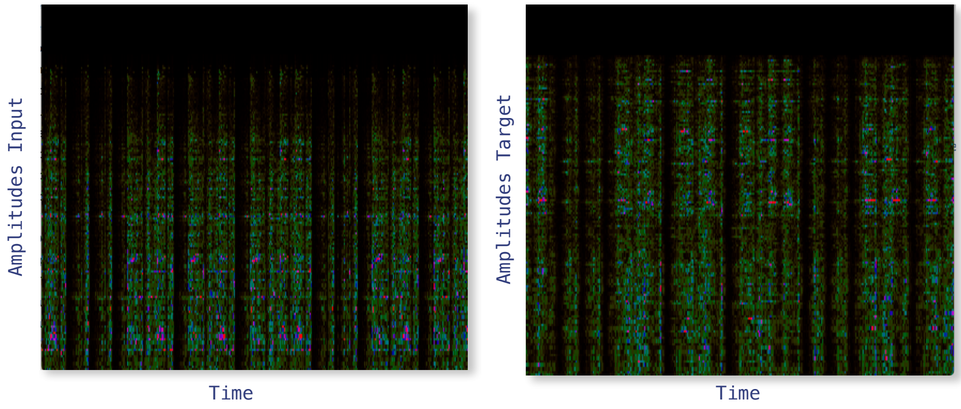
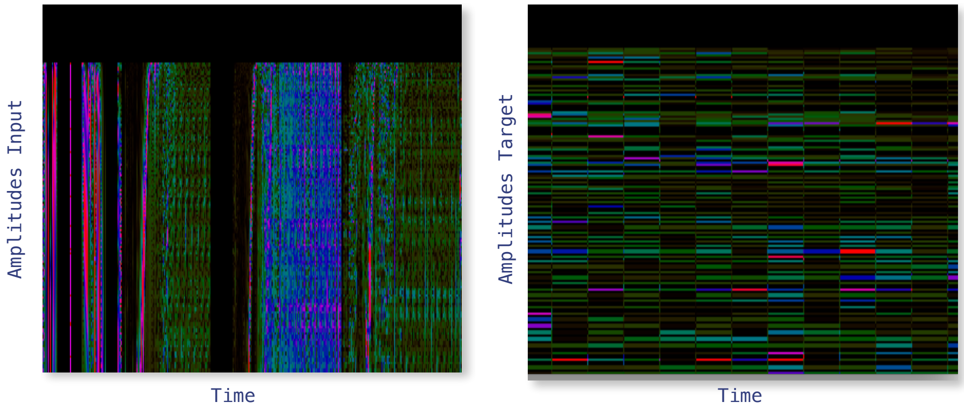


Fig. 55
Processed Input and Target Spectrograms



Multiple Parameters

After success in the environment of the single parameter, the structure of the application is refactored in order to handle the ability to manage and optimize multiple synthesis parameters. In the *runFFT()* function, the gradient descent calculation is applied to a randomly selected parameter.

For testing the system, again, a drum loop is preprocessed through the granular system, and it's settings saved for comparison against the results of the learning algorithm. (Fig. 53) However, this time multiple parameters are changed from that of the default settings within *SelfProgrammer*.

In the FFT plots of the unprocessed input and target signals we again see similar frequency responses, which have been skewed due to the granular processing. (Fig. 54)

In the FFT plots of the processed input signal, we see highly sporadic movements, with rapid changes in the amplitude of the bins, as well as complete dead zones in the time domain. In the target signal we see the same bins of the buffer being repeatedly plotted in equally divided segments. (Fig. 55) This indicates the buffers are being continually reprocessed to the cap of the available number of learning requests, and that the learning is never satisfying the system.

The fact that the system is continually hitting the cap of the learning algorithm suggests that the there has been a struggle to scale to multiple parameters. This is likely due to an incorrect usage of the gradient descent function, as well as a machine learning system too naive for the necessary application. The dead buffers and unorthodox FFT plot of the processed input also suggest problems or invalid values in the granular processing, which potentially corrupt the error calculation.

Conclusions

The aim of this paper was to investigate the machine learning of synthesis parameters through the automated programming of a custom rhythmically focused granular engine. The research involved an exploration into circular buffers, audio grains, window functions, sample rate schedulers, granular architecture, and rhythmic tempo syncing. It also explored machine learning architecture, audio data acquisition, error calculation, gradient descent, and parameter optimization.

The positive response of musicians to the rhythmic syncing of granular processing shows that there is room for innovation in the creation of new granular synthesis engines. The successful machine learning of a parameter within this system shows promise in the future of automated synthesis programming. Although the system had trouble with multiple parameters, it has revealed the next steps on the path to automated synthesis programming.

Bruce Dawson, an audio software developer with an MFA in music technology and BS in computer science said, "*as interaction between machine learning and musical functionality draws closer together, the possibility of autonomous synthesis infrastructures becomes a real possibility -- we live in the age of consistent digital and musical advancement and I forsee the continuous development of groundbreaking algorithmic processes becoming a staple of music for the forecoming centuries. Studies today such as Jacob's contribute to the musical world of tomorrow, and through these contributions to music and computing fields, the realization of the algorithmic future of music computing is astonishing and all too real.*"

Future work will involve the creation of a more robust audio synthesis environment, capable of creating audio from scratch, as well as research into a more advanced machine learning system. Possible improvements to thesystem could include more features for error calculation, a neural net to control changes in parameters, as well as optimization across entire audio files as opposed to single audio blocks.

References

1. Mariam Webster. "Atomism Definition." Mariam-Webster.com (2016)
2. Roads, Curtis. "The Evolution of Granular Synthesis: An Overview of Current Research." International Symposium on The Creative and Scientific Legacies of Iannis Xenakis (2006)
3. Lebrecht, Norman. "The Maestro Myth: Great Conductors in Pursuit of Power. Secaucus." NJ: Carol Pub. Group (1999)
4. Alpaydin, Ethem. "Introduction to Machine Learning." Cambridge, MA: MIT, (2010)
5. Bencina, Ross. "Implementing Real-Time Granular Synthesis."(2001)
6. Iannis Xenakis, "Formalized Music: Thought and Mathematics in Composition." Bloomington and London: Indiana University Press (1971)
7. Yee-King, Matthew. "SYNTHBOT: AN UNSUPERVISED SOFTWARE SYNTHESIZER PROGRAMMER." Informatics University of Sussex (2008)
8. Tubb, Robert, and Simon Dixon. "The Divergent Interface: Supporting Creative Exploration of Parameter Spaces." NIME. (2014)
9. Tatar, Kıvanç, Matthieu Macret, and Philippe Pasquier. "Automatic Synthesizer Preset Generation with PresetGen." Journal of New Music Research (2016)
10. Cartwright, Mark, and Bryan Pardo. "SynthAssist: Querying an Audio Synthesizer by Vocal Imitation." NIME. (2014)
11. Tubb, Robert, and Simon Dixon. "A zoomable mapping of a musical parameter space using hilbert curves." Computer music journal. (2014)
12. Yuksel, Kamer Ali, Batuhan Bozkurt, and Hamed Ketabdar. "A software platform for genetic algorithms based parameter estimation on digital sound synthesizers." Proceedings of the 2011 ACM Symposium on Applied Computing. (2011)
13. Donahue, Chris. "Applications of genetic programming to digital audio synthesis." The University of Texas at Austin, Department of Computer Science (2013)
14. Truax, Barry. "Composing with real-time granular sound." Perspectives of New Music (1990)
15. Truax, Barry. "Composing with time-shifted environmental sound." Leonardo Music Journal (1992)
16. Tatar, Kıvanç, Matthieu Macret, and Philippe Pasquier. "Automatic Synthesizer Preset Generation with PresetGen." Journal of New Music Research (2016)
17. Roads, Curtis. "Granular synthesis of sound." (1985)
18. De Poli, Giovanni, and Aldo Piccialli. "Pitch-synchronous granular synthesis." Representations of musical signals. MIT Press. (1991)