Different approaches to audiowave and melody restoration

Autors:

Juan Gonzalo Quiroz Cadavid
jquiro12@eafit.edu.co

Agustín Nieto García
anietog1@eafit.edu.co

Juan José Parra Díaz
jjparrad@eafit.edu.co

Kevin Arley Parra Henao
kaparrah@eafit.edu.co

Francisco José Correa Zabala
fcorrea@eafit.edu.co

Systems Engineering Department, EAFIT University
Medellín

Abstract

In this investigation we made two approaches to musical reconstruction, one reconstructing the melody with machine learning and another reconstructing audio waves with interpolation. In this document there can be found those approaches, and an analysis of the results. With the obtained results we could get to the conclusion that, for audio wave restoration, cubic interpolation is a great approach if we have strong security on the points we have. For melody restoration with machine learning we didn't get the expected results, as it's not melodic restoration but a creation of melodies.

Introduction

In this investigation we are going to present different approaches attempting to resolve the problem of reconstructing a song that has been damaged in some way. For this purpose we are going to introduce different methods of interpolation and machine learning that pretend to interpolate a function. We represent the song as a function of time and frequency. Another idea that we want to explore is which interpolation method and which machine learning method are the best for their purpose.

We pretend to apply interpolation methods learned in class and others that we found in our investigation to repair songs. For this reason we introduce the representations of the songs and try to decide which one is better for this purpose. Then, with the song processed and the damaged points recognized, we finally apply interpolation and machine learning methods to try to recompone the song, reconverting to an audio format and plotting the results.

Objectives

General:

- Find out which interpolation and machine learning method is more appropriate for musical reconstruction.

Specifics:

- Reconstruct a song with damaged audio waves using interpolation.

- Reconstruct a song with damaged melody using machine learning.

Theoretical Framework

Audio

Sound is a vibration that propagates in a medium as a wave. As this phenomenon behaves as a wave, it has some properties as wavelength, amplitude and frequency. The combination of multiple sounds generates a new sound, whose soundwave is the merging of the original sounds' waves.

In any sound, there are four elements that can be distinguished: Tone (frequency), Intensity (amplitude), duration and pitch. The range of frequencies in which a sound is possible for humans to hear is between 16 and 20000 Hz.

Audio representation in computers

Whenever we want to store audio in a computer, as it is an analog signal, there has to be a digitalization process. For this goal, a microphone is used. It receives the vibrations from the sound, converts them to an electric signal and analyses it from time to time. In this way, the computer ends up with a group of values separated by the same interval of time. The more frequent this analysis occurs, the more accurate the sound recorder will be.

This process is called Sampling. A sample, in the audio context, is a measurement that is taken in a precise moment of time. For converting analog sound to digital, a number of samples are taken every second so that, in that way, we end up with a group of samples that describe a wave pattern, the wave of the audio that is being recorded. The more samples are taken, the more accurate this conversion from analog to digital will be.

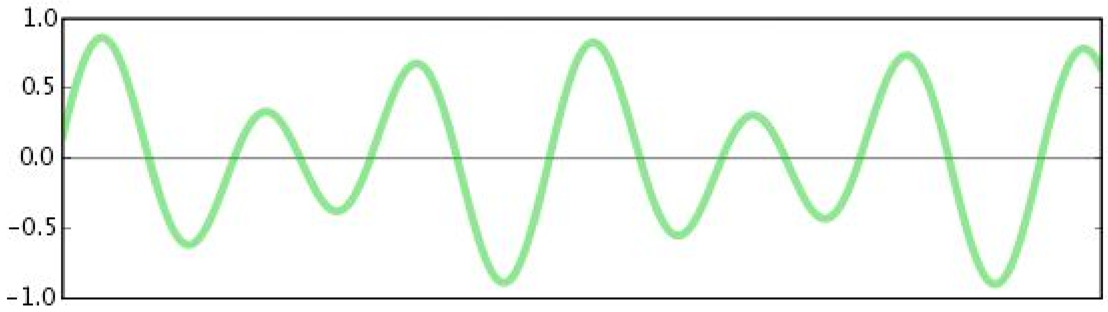


Figure 1: Analog audio waveform
Taken from https://manual.audacityteam.org/man/digital_audio.html

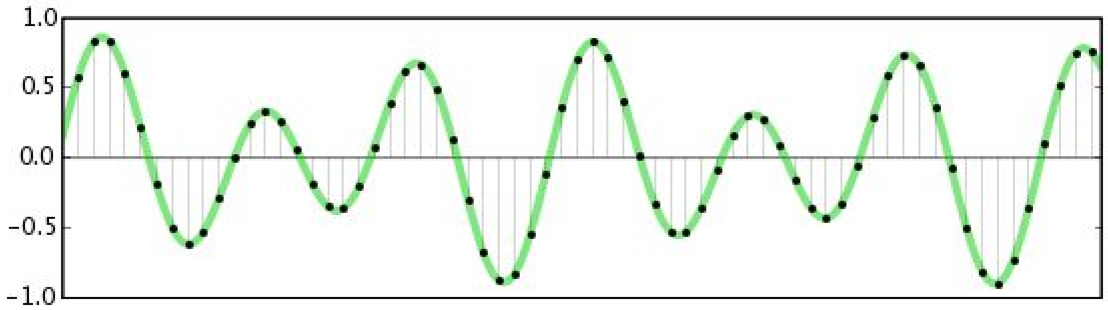


Figure 2: Sampled Analog audio
Taken from https://manual.audacityteam.org/man/digital_audio.html

In figure 1 we can see a specific audio waveform, which in the process of digital conversion, as in figure 2, gets sampled each certain time.

Audio Formats

When it is wanted to store these soundwaves in the computer, we can often find multiple file formats that can help us. Three commonly used formats are .wav, .mp3 and .mid.

A .wav file, Waveform Audio Format, is a format in which audio is stored as a soundwave. The wave that enters the computer is the same that is stored in this format.

An .mp3 format, also stores audio in its soundwave form, but it is not as accurate as .wav files. Opposed to them, this format uses a compression algorithm that greatly reduces the size of the files, deleting the sounds that are not hearable for humans and using a lower sampling rate.

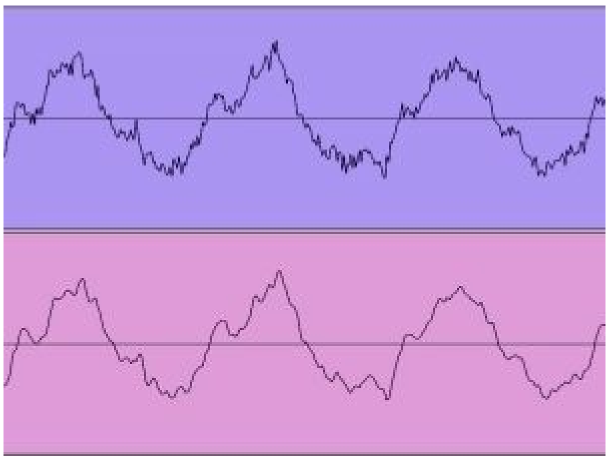


Figure 3: WAV and MP3
Taken from https://randycoppinger.com/2012/10/16/wav-vs-mp3/

In Figure 3 we can see the same audio stored in different formats. As said before, the soundwave from the .wav file (purple) does not use any kind of compression and is a more accurate representation of the original sound than the .mp3 file (pink).

A .mid file, Musical Instrument Digital Interface, is a format that does not store audio itself but instructions to play it. Instead of storing audio as a soundwave, this format stores the velocity of the audio, instrument and which notes are supposed to be played at which points in time.

Interpolation

Interpolation is a method of constructing new data points within the range of a discrete set of known data points. As we often have a number of data points, obtained by sampling or experimentation and it is often required to interpolate, i.e., estimate the value of that function for an intermediate value of the independent variable. Sometimes it's better to interpolate without using all the points, because of the error propagation.

It is about to determine the behavior of a model. This situation is modeling by using a set of points that represents two or more variables, in which one or more variables depend of the others. This behavior is characterized as a function. The idea is to approximate one function to estimate the behavior of a variable in a determined interval [1]. Also we need to determine the value of the phenomenon for a value that isn't in the initial data. The main methods to do this are: Newton, Lagrange and Neville.

To introduce interpolation, is important to introduce the following theorem [1]:

Given $n + 1$ *points* $\left(x_0, y_0\right), \left(x_1, y_1\right), \left(x_2, y_2\right), \ldots, \left(x_n, y_n\right)$ *with the condition that* $x_i \neq x_j$ *for each* i, j *such that* $0 \leq i \leq n$ *and* $0 \leq j \leq n$, *then there exits a unique polynomial* $p(x)$ *of degree at most n with the property*:

$p\left(x_i\right) = y_i \quad i = 0, 1, 2, \ldots, n$

Newton interpolating polynomial:

From previous theorem, is deduced that the Newton interpolation polynomial has the form:

$$p_n(x) = b_0 + b_1\left(x - x_0\right) + b_2\left(x - x_0\right)\left(x - x_1\right) + \ldots + b_n\left(x - x_0\right)\left(x - x_1\right)\ldots\left(x - x_n\right)$$

However, the process to obtain each b term is so long. For this reason we introduce another technique that is known as Newton interpolating polynomial with divided differences. Which is an incremental process in which we obtain an interpolating polynomial in each step [1].

Lagrange interpolating polynomial:

The following theorem defines the Lagrange interpolating polynomial [1]:

Given $n + 1$ *points* $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, *there exits a unique polynomial* $p(x)$ *of degree at most n such that*:

$$p(x_i) = f(x_i) \quad i = 0, 1, 2, \ldots, n$$

The polynomial is given by

$$p(x) = L_0(x) f(x_0) + L_1(x) f(x_1) + L_2(x) f(x_2) + \ldots + L_n(x) f(x_n)$$

$$= \sum L_k(x) f(x_k) \quad k = 0, 1, 2, \ldots, n$$

where for each k, $L_k(x)$ *is*

$$L_k(x) = \frac{(x - x_0)(x - x_1)\ldots(x - x_{k-1})(x - x_{k+1})\ldots(x - x_n)}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})(x_k - x_{k+1})\ldots(x_k - x_n)}$$

$$L_k(x) = \prod \frac{(x - x_i)}{(x_k - x_i)} \quad i = 0, 1, 2, \ldots, n \quad \text{with } i \neq k$$

Machine Learning

Machine Learning is a field of Artificial Intelligence. It's the process of Learning from data, this leaning could be used to learn how to differentiate those data or it could be used to learn to predict new data, both approaches are based on prior data. This represents the fourth paradigms of Machine Learning:

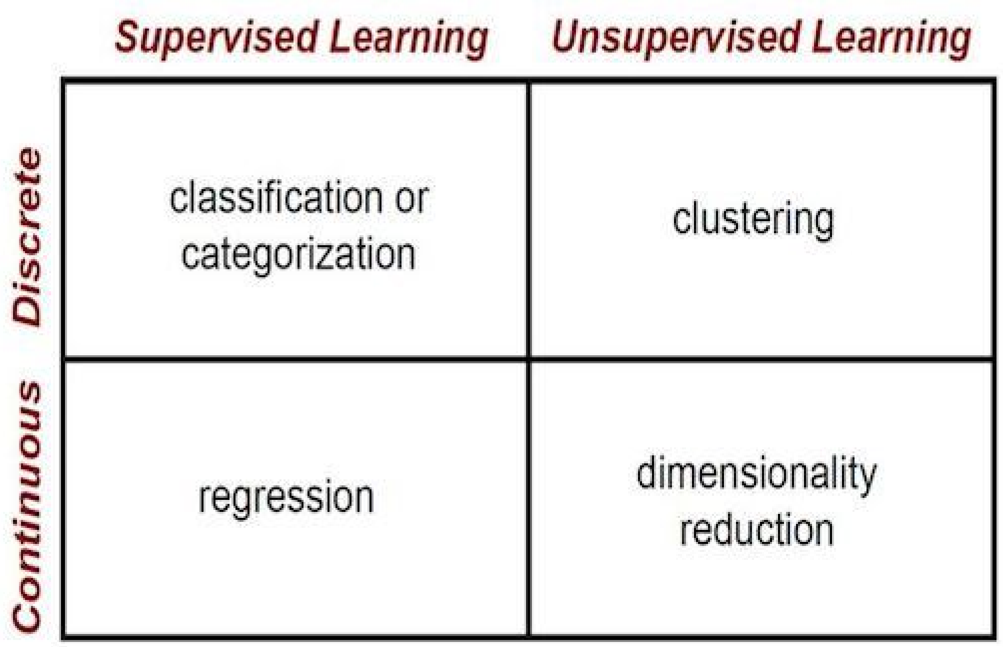


Figure 4: Supervised vs unsupervised learning.

Taken from: https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d

Our work area would be in Quadrant One and three where we worked with a set of points (x, y), keep in mind that 'x' represents all the variables involved or Features and 'y' would be in this case the answer to that set of values in x. This gives us a marked set of data, where we know the input (X's values) and their respective output (Y's values). These data are used in supervised learning, where we use a set of features and their respective result for Train the different Models.

The difference between those Quadrants (One and three) are the way we can interpetarted the problem, whether we have the task to determine the belonging class of an element or we have to predict the possible outcome to a certain set of features.

Development

WAV: Audio Wave restoration with Interpolation

Points selection

For this approach, we decided to use .wav files, as those files contain actual sound, which is represented as an array of amplitudes (y axis) over time (x axis) and behaves like a function at the small scale, as waves are represented by sinusoidal functions. With this decision some problems arise, such as the big amount of points that will need to be interpolated, for example, a song of 4 minutes, 10 seconds and a sample rate of 48000 has around 12 million points of data. Thus we have to implement efficient methods with a minimal computational complexity and which can also minimize the propagation of error during evaluation, as the Runge's Phenomenon [2] and error propagation effects can arise.

Damage methods

Already having the groups of points, we needed a way to damage them so that the song restoration could make some sense and we could make comparisons. For this instance, we decided to damage the songs with two main methods: zeroing some points and adding some noise. The damage methods are an important field of study if recognition algorithms are going to be developed and data is needed for recognition however as that's not what we're centering our investigation on, we tried to keep it simple.

```
zerofill(y, percent): # a percent [0, 1] of the points become 0
    for i in range(len(y)):
        if random.random() < percent: # the random number [0, 1] is
below the percent
            y[i] = 0 # later on we expanded this function in order to
fill with zero blocks, not units

noiseadd(y, percent, rate): # noise is added to a percent [0, 1] of the
points
```

```
    for i in range(len(y)):
        if random.random() < percent:
      # y[i] becomes a value between (1 - rate) * y[i] and (1 + rate) *
y[i]
            y[i] = y[i] * random.uniform(1 - rate, 1 + rate)
```

Here are some examples of how a damaged audiowave would look like. The first one is an example of a zeroed song with a percent of 0.5 and the second one is a noisy song with a percent of 0.7 and a rate of 0.2.

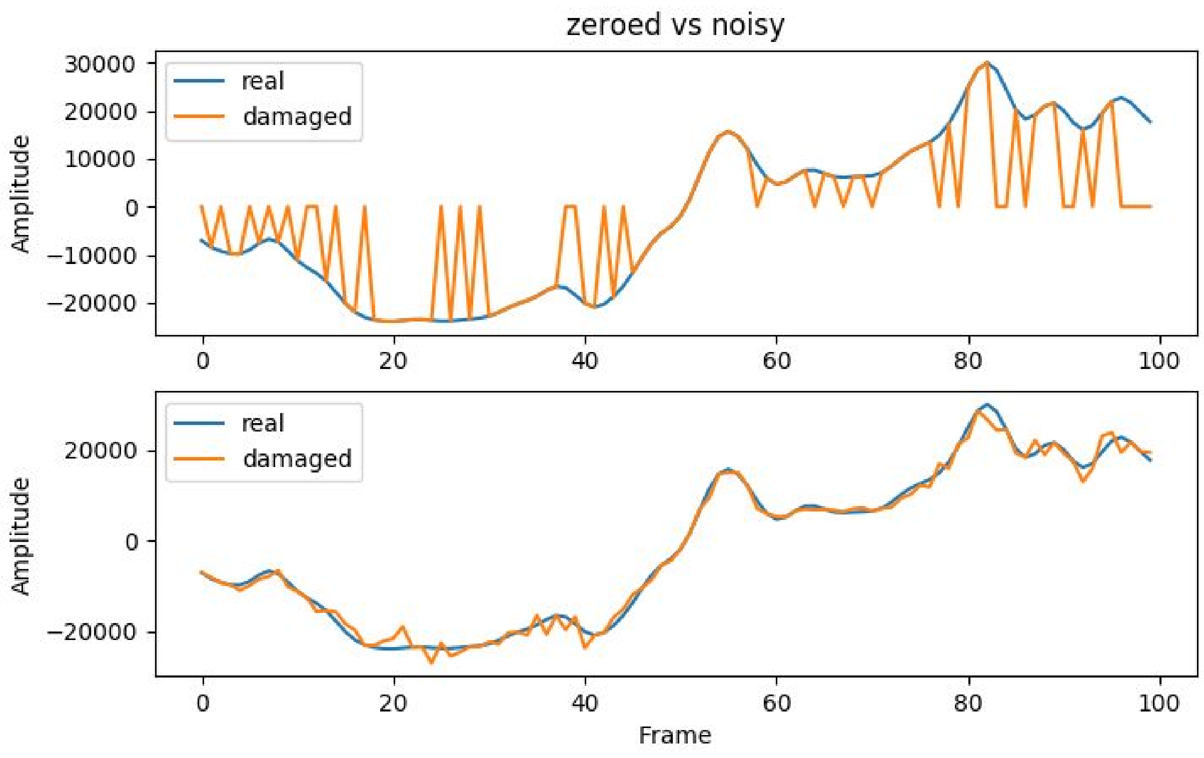


Figure 5: zeroed and noisy songs

Damage Recognition

We don't go in depth over damage recognition, but it's a truly important field, as for interpolation it's imprescindible to only have points in which it's possible to believe they belong to the function. Further in this paper we'll compare the behaviour of the restoration against false positives and false negatives, in which we found that it's way better to have false negatives than false positives, for the reason exposed previously. Here we propose a simple method which behaves badly if there are big blocks of damaged points, but that is not such a bad method if there are only some points damaged.

For this process, we use a damage parameter, which we propose as a statistic parameters that depends of the damage type. For our approach, we chose a damage parameter based on the mean of maximum difference between two consecutive points in the song and with that value, we compare the difference between the amplitudes of two points in the damaged song, and those that are greater than damage parameter are marked as points to needed to be repaired. This method is based on the fact that audio waves have a smooth behaviour most of its time. We wanted to do it making use of the slope, but the deltax would always be the same and so it's enough with the heights. Here's the code used, named after its designer.

```
def kevin(y, maxdelta):
    matches = [True] * len(y)
    laststable = y[0]

    for i in range(1, len(y)):
        delta = abs(float(laststable) - float(y[i]))

        if delta > maxdelta:
            matches[i] = False
        else:
            laststable = y[i]

    return matches
```

Later in the process we decided to emulate damage recognition and so we did it via comparing the original audiowave with the damaged audiowave and adding false positives and false negatives. The amount of those parameters would be as small as good the recognition algorithm would be. It's called cheat as it's necessary to compare with the original audiowave, which would be data we wouldn't have in reality.

```
def cheat(y0, y1, false_positives=0, false_negatives=0):
    matches = [i == j for (i, j) in zip(y0, y1)]

    for i in range(len(matches)):
        if matches[i]:
            if random.random() < false_negatives:
                matches[i] = False
        else:
            if random.random() < false_positives:
                matches[i] = True

    return matches
```

Song restoration

Songs are restored via interpolation, taking a group of points in which we have complete security about the song and guessing the ones in the middle; if we can't have security in the last or first points we have to use extrapolation [3], which doesn't give as good results as interpolation and can even give worse results than those of the initial values. What we have found is that as long as the interpolating points are good (are not too sparse and there's a minimal amount of false positives), the song can be restored with good results.

We analyze the results of restoring songs via interpolation following the next workflow:

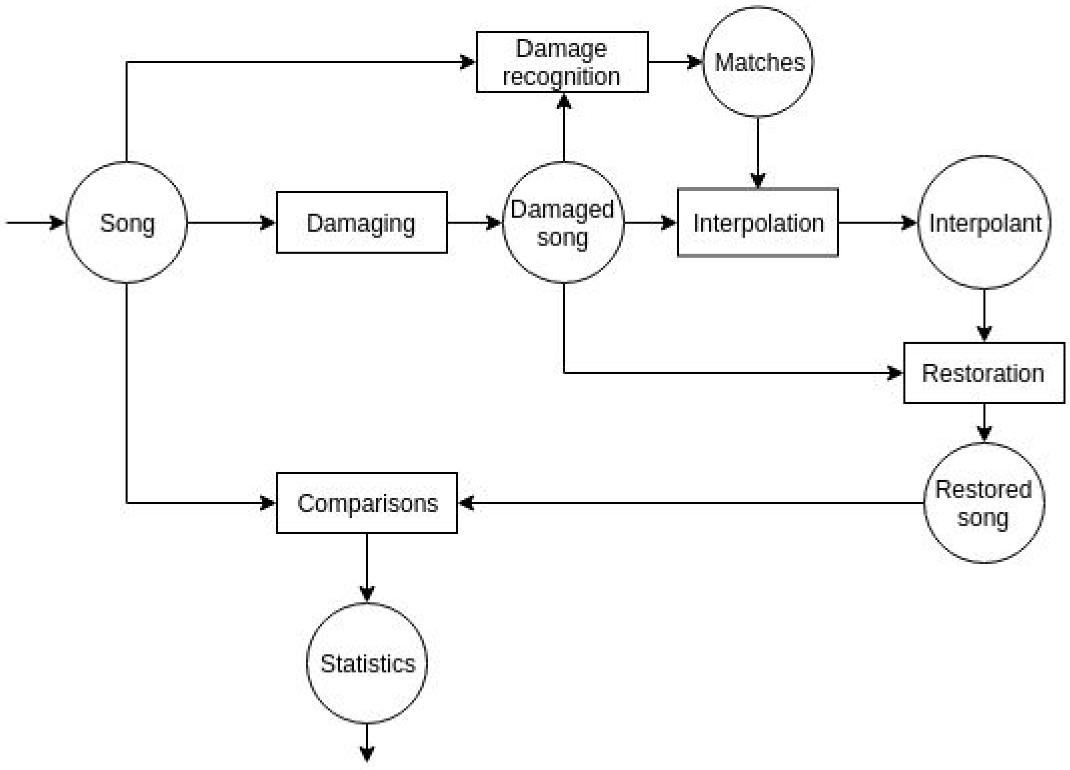


Figure 6: Wav restoration via interpolation workflow

We made use of Lagrange, Newton, linear and cubic interpolation for our investigation and these were the results graphically:

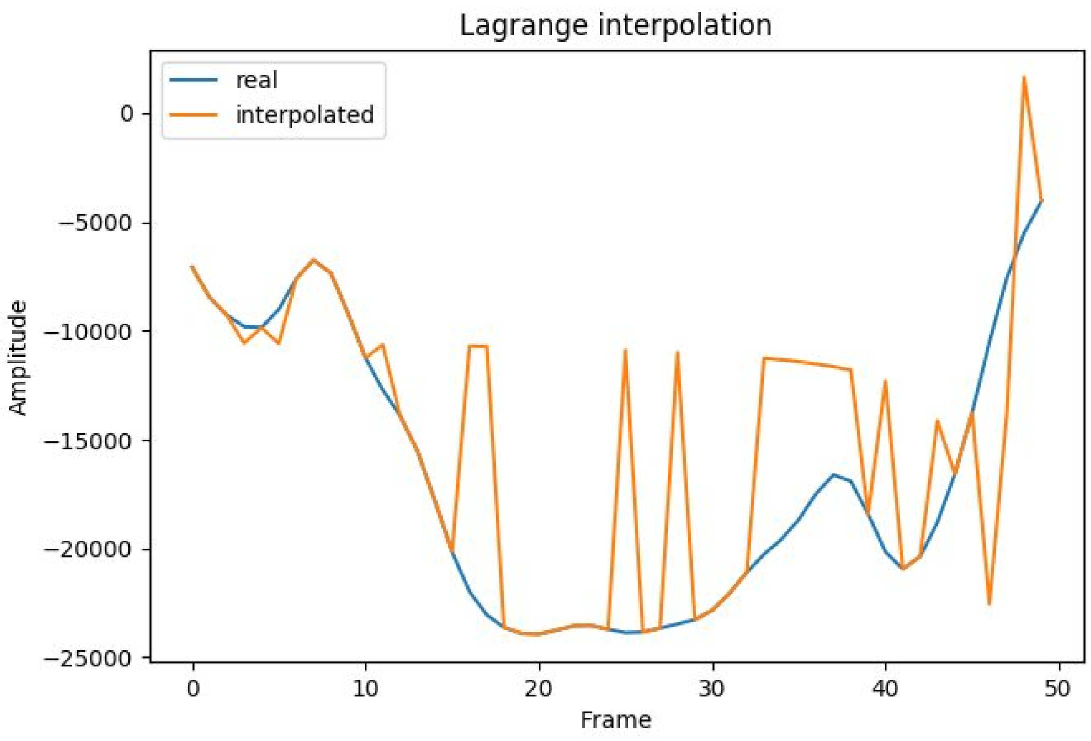


Figure 7: Results of reconstructing 50 points with Barycentric Lagrange interpolation.

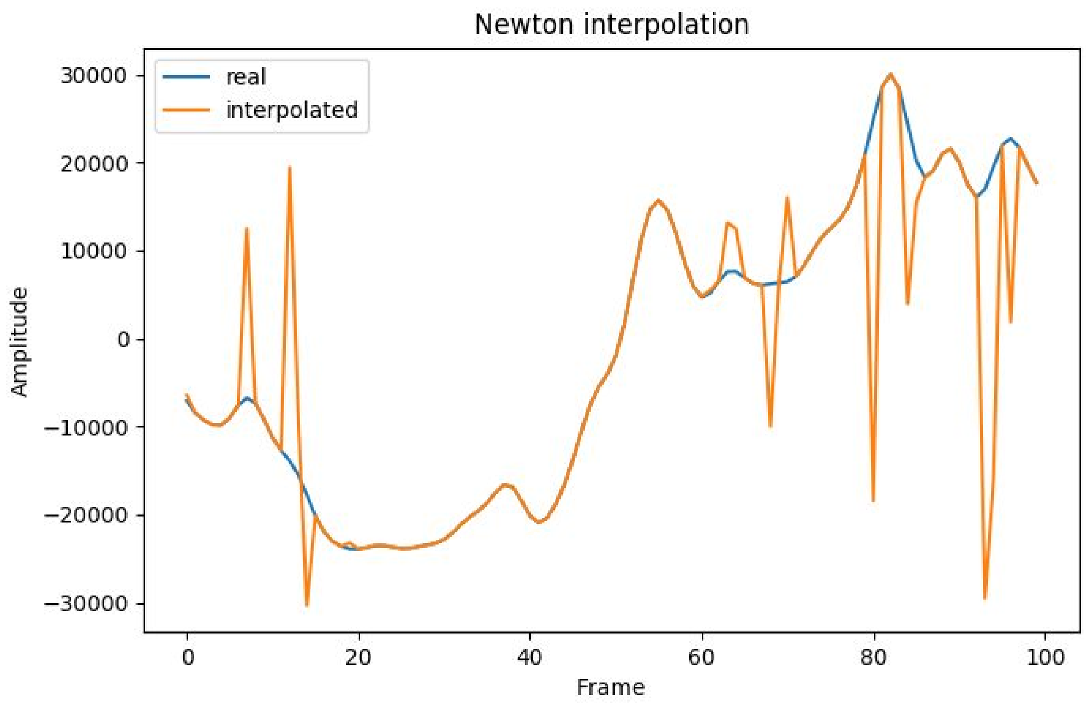


Figure 8: Results of reconstructing 100 points with Newton interpolation.

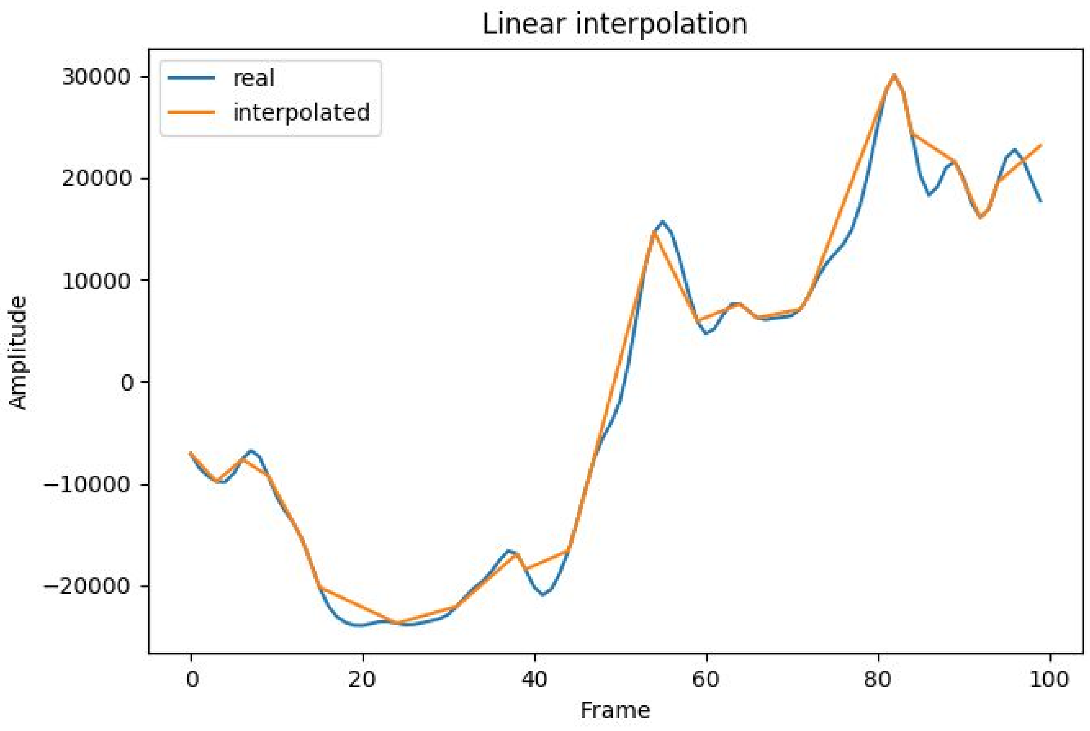


Figure 9: Results of reconstructing 100 points with Linear interpolation.

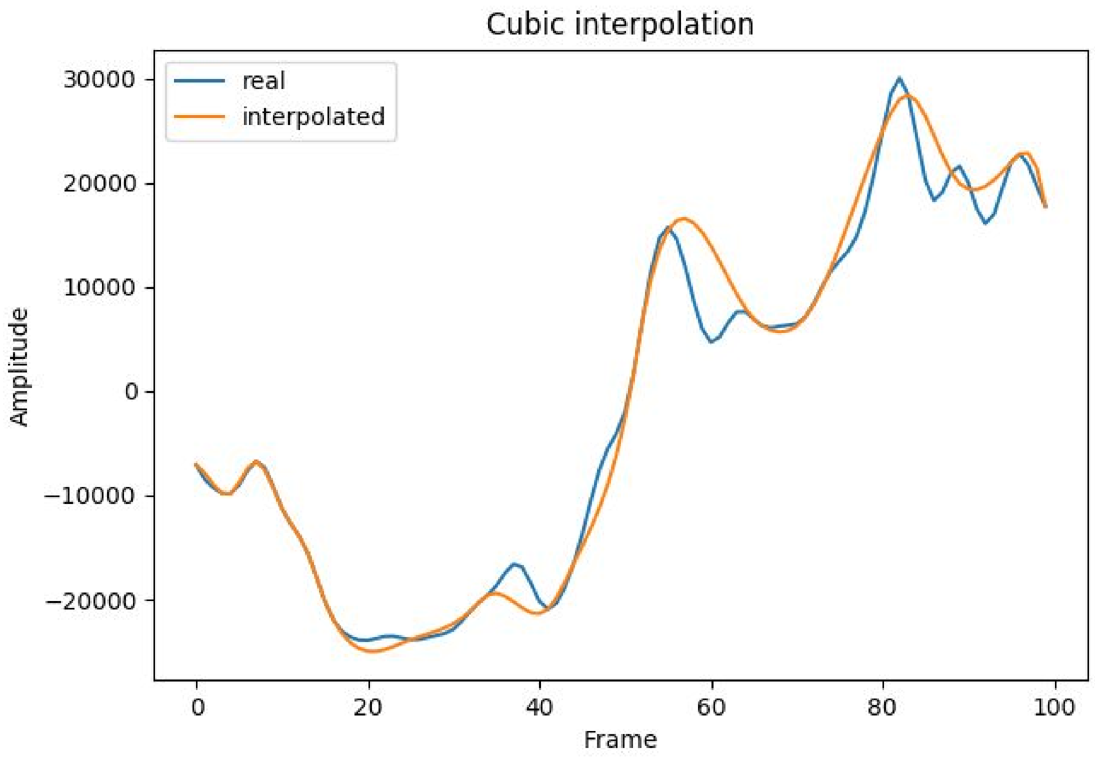


Figure 10: Results of reconstructing 100 points with cubic interpolation.

After those results we decided we should go further on the use of cubic and linear splines, so these are the ones used for interpolating complete songs as their computational complexity is O(n) and their propagation of error is the smallest, also because making use of only the nearest points for guessing makes more sense in this case. Our results can be condensed in the following tables:

Linear with zeros		**False positives**				
		0	0.05	0.1	0.2	0.3
False Negatives	0	135.86041	237.583604	334.625015	511.269714	670.420434
	0.05	155.293355	260.309487	360.874239	544.709645	709.365081
	0.1	179.401046	287.941129	387.7423	576.480734	750.520286
	0.2	234.040854	351.831753	460.104506	660.405308	842.093958
	0.3	305.38807	432.77936	556.574499	764.098212	949.703478

Table 1: Behaviour of the mean of absolute errors against the amount of false positives and false negatives for 1000000 points damaged with zerofill with a percent of 0.4 and blocks of size 1 (this is an enhancement of the damage method). Only unmatched points are taken into account.

Linear with	**False positives**

noise		0	0.05	0.1	0.2	0.3
False Negatives	0	329.697093	314.551198	306.770952	290.375657	283.522123
	0.05	361.813965	344.321265	329.136131	308.2646	299.452073
	0.1	397.07185	373.085778	356.877214	331.087377	317.204629
	0.2	472.443719	440.801883	415.349367	380.198499	357.226224
	0.3	573.0044	532.280284	491.035218	436.843381	405.245074

Table 2: Behaviour of the mean of absolute errors against the amount of false positives and false negatives for 1000000 points damaged with noiseadd with a percent of 0.6 and a rate of 0.3. Only unmatched points are taken into account.

Cubic with zeros		**False positives**				
		0	0.05	0.1	0.2	0.3
False Negatives	0	58.67985	226.628084	373.806013	607.371825	793.449291
	0.05	72.701371	252.496784	406.288676	654.002582	844.452562
	0.1	88.819745	281.880453	442.376317	703.627853	909.730519
	0.2	131.307795	354.859151	537.74678	828.675762	1039.335288
	0.3	189.959724	442.836211	655.288005	973.797695	1206.577178

Table 3: Behaviour of the mean of absolute errors against the amount of false positives and false negatives for 1000000 points damaged with zerofill with a percent of 0.4 and blocks of size 1 (this is an enhancement of the damage method). Only unmatched points are taken into account.

Cubic with noise		**False positives**				
		0	0.05	0.1	0.2	0.3
False Negatives	0	217.239692	230.731235	243.091301	261.208196	275.697011
	0.05	247.267077	255.870292	266.287951	279.110648	294.873462
	0.1	273.376688	285.609499	290.268593	300.671935	312.094459
	0.2	353.889467	355.155709	354.617193	354.806403	354.825858
	0.3	464.253711	449.112332	433.950218	418.695803	409.323418

Table 4: Behaviour of the mean of absolute errors against the amount of false positives and false negatives for 1000000 points damaged with noiseadd with a percent of 0.6 and a rate of 0.3. Only unmatched points are taken into account.

So we can conclude it's better for a recognition method to have false negatives instead of false positives and it can be worse if what we're talking about is noise and not real damage, which is a good payload because one is easier to recognize than the other one.

MIDI: Melody Reconstruction with Machine Learning

Group of points selection

With the purpose of reconstructing songs, we decided to use the .mid file extension. The midi file format was chosen because of the information that it contains. Instead of having a soundwave, it has the musical note and the time in which each one of these notes must be played. This information can be mapped into a data structure that contains times and frequencies, in other words, a group of points whose axis are time and frequency.

We decided to start our investigation with the information provided by these groups of points, so the implementation of the reconstruction algorithms will be applied on those. By this approach, we are not reconstructing a song with damaged soundwaves, but instead, we are reconstructing a song with damaged melody. The workflow of Midi is:

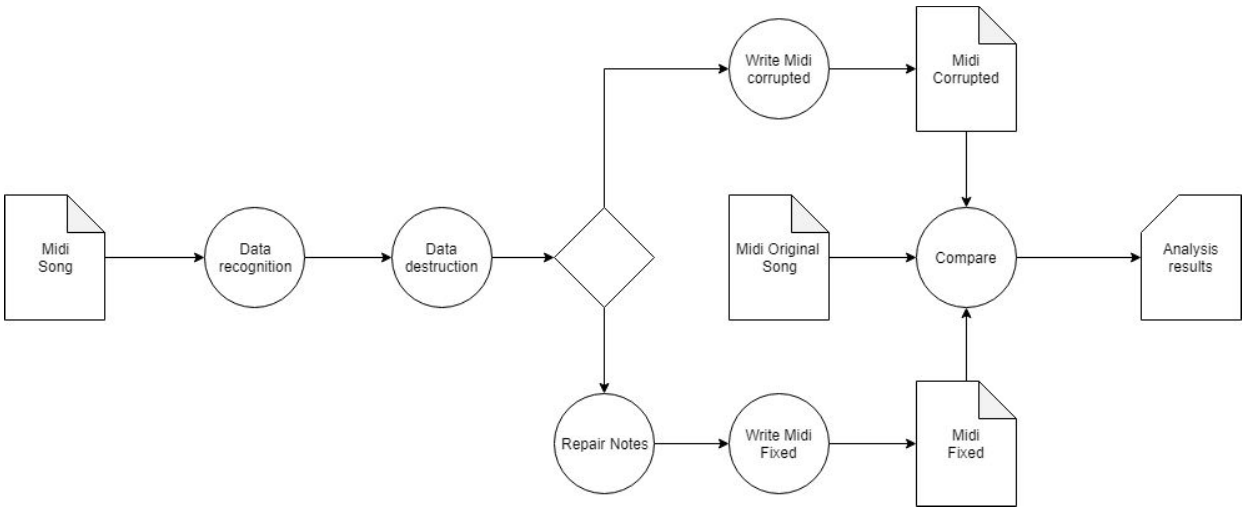


Figure 11: Workflow of the reconstruction of melodies via interpolation.

The Midi file is composed by Tracks, each track has a list of Messages, those Messages could be meta messages or notes messages. We focus only on notes messages; nevertheless we mostly worry about the meta messages when dealing with reconstruction of the midi.

The next code is used to split the data on a structure was easy to manage

```
def read_midi(mid):
    my_tracks = [] # It's a list, where each row has [meta_msg
,notes_msg ,msgs ]
```

```
    #Each row represents a Track
    for track in mid.tracks:
        meta_msg = []
        notes_msg = []
        msgs = []
        for msg in track:
            msgs.append(msg)
            if msg.type == 'note_on':
                notes_msg.append(msg)
            else:
                meta_msg.append(msg)

        my_tracks.append([meta_msg,notes_msg,msgs])

    return my_tracks
```

Damage Methods Explained

Once we got the data ready to be used we break some notes using the same approach as with WAV, which is; using a random we attempt to damage a percentage of the song, the next code represents the procedure

```
def ralph(notes,percent):
    #We used this to have track of the broken poss
    broken_pos = [False] * len(notes)

    for i,note in enumerate(notes):
        ran  = random.random()
        # The algorithm need a least n_steps to learn
        if(ran < percent) and (i > (n_steps * 2)):
            broken_pos[i] = True
            note.note = 0
    return notes,broken_pos
```

Here we used a list to have track of the broken parts, It's used to compare results at the end. At least we need a range of good notes to let the algorithm learn from it.

Melody restoration

Due to the problem approach (time series) it was decided to work with LSTM cells. LSTM are good when learning about sequences, that is possible due to the cell state and the information flow between each LSTM cell on the same layer.

A univariate model with an input layer of 15 nodes was built, where each node holds information about the series (for a single step onto the neural network it perceives as input a vector from time T to time T-15) That is, to predict the next one we used 15 time steps. This number is the result of running the same model changing this hyper parameter and checking which is the best (Random search for hyper parameter optimization). The same approach were used to know how many hidden layers are needed to have an optimal performance.

The hidden layers are fully connected between them, each layer is a 50 cell LSTM layer (random search results). Finally, the output layer is a layer of one neuron (meaning the next step in out time series).

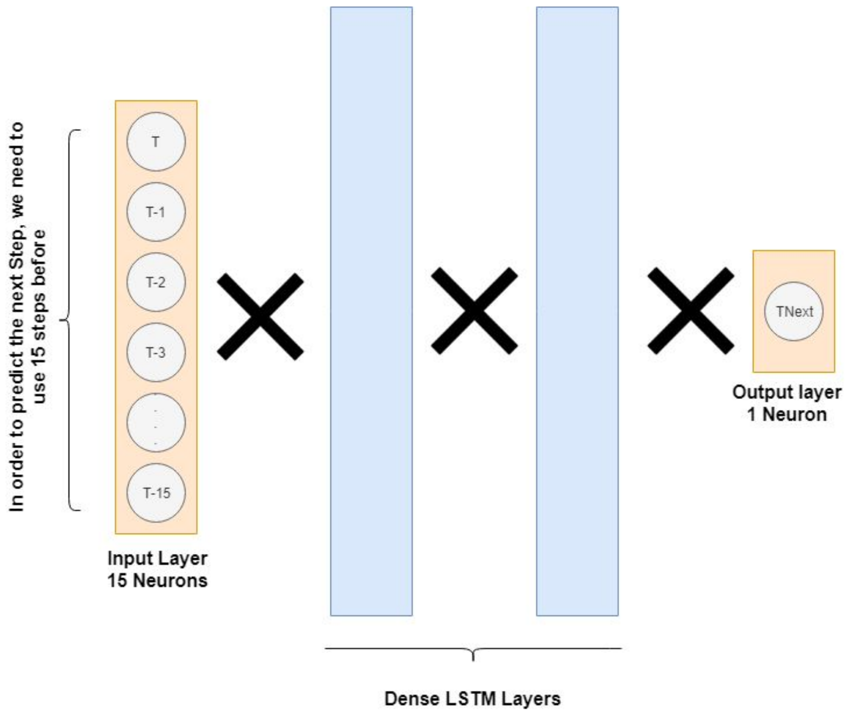
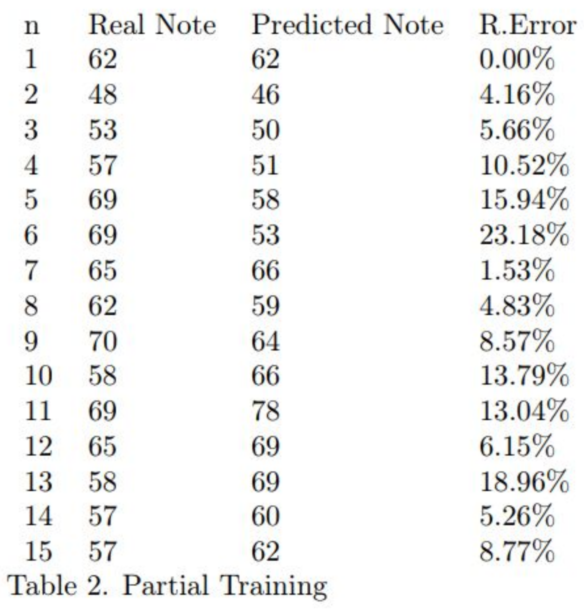


Figure 12: LSMT sketch.

```
def model_stacked_lstm(n_steps,n_features):
    global model
    model = Sequential()
    model.add(LSTM(50, activation='relu', return_sequences=True,
input_shape=(n_steps, n_features)))
    model.add(LSTM(25, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse')
```

The full code can be found in the attachments: A1

Using this approach, we corrupt 15 random notes. The results are:



n	Real Note	Predicted Note	R.Error
1	62	62	0.00%
2	48	46	4.16%
3	53	50	5.66%
4	57	51	10.52%
5	69	58	15.94%
6	69	53	23.18%
7	65	66	1.53%
8	62	59	4.83%
9	70	64	8.57%
10	58	66	13.79%
11	69	78	13.04%
12	65	69	6.15%
13	58	69	18.96%
14	57	60	5.26%
15	57	62	8.77%

Table 2. Partial Training

Table 5: Behaviour of the relative error of the predictions given by the LSMT.

Where the average relative error is 9.36%, Showing good result.

Conclusions

For audiowave restoration we found that the interpolation method that gave the best results (reconstruction closest to the original song) was the cubic interpolation as it minimizes the propagation of error and the complexity of rebuilding a long audiowave is reduced to O(n) instead of O(n^2), which would be the complexity with Lagrange or Newton, which can only work stably with at most 30 points.

In melody restoration we couldn't get as good results as in audiowave restoration, that's because melodies in this case are only instructions and so on we are trying to reconstruct instructions, which don't necessarily need to make sense as that's what the author of the mellody has to do. We're not reconstructing in this case, we're almost creating melodies. That's something a Machine Learning algorithm can afford, but it's not our investigation focus.

Globally, we can conclude that audiowave restoration is a way better approach because it's behaviour is more function-like and losing some points in it is not as troublesome as losing data in a midi file, where each point can represent much information about the song. We think machine learning could give us better results than interpolation in audiowave restoration if there're a lot of false positives as it avoids overfitting; however if we are plenty secure of the points we have and they're not too sparse, interpolation showed really good results.

Future Work

We can make a statistics study about the damage caused in songs by our damages process, to choose a better damage parameter of each of them. Besides, study real damages process and estimate a damage parameter to apply our method for real cases. We can use several songs and damage it, then compare them with the original and make a statistics study to get values like standard deviation and average to estimate the damage parameter.

Attachments

A1. Midi Program

```
from mido import Message, MidiFile, MidiTrack

# multivariate LSTM forecasting
from numpy import array
from numpy import hstack
import numpy as np
import random
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import RepeatVector
from tensorflow.keras.layers import TimeDistributed
import copy
import math
import time


def model_vanilla(n_steps,n_features):
    global model
    model = Sequential()
    model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
    model.add(LSTM(25, activation='relu' ))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse')

def model_stacked_lstm(n_steps,n_features):
    global model
    model = Sequential()
    model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps, n_features)))
```

```python
    model.add(LSTM(25, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse')

    return model

def train_batch(notes):
    x,y = split_sequence(notes, n_steps)
    x = x.reshape((x.shape[0],x.shape[1],n_features))

    model.fit(x,y,epochs=epochs,verbose=verbose)

def split_sequence(sequence, n_steps):
    X, y = list(), list()

    for i in range(len(sequence)):
        end_ix = i + n_steps

        if end_ix > (len(sequence) -1):
            break

        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)

    return array(X), array(y)

def ralph(notes,percent):
    broken_pos = [False] * len(notes)

    for i,note in enumerate(notes):
        ran  = random.random()

        if(ran < percent) and (i > (n_steps * 2)):
            broken_pos[i] = True
            note.note = 0


    return notes,broken_pos

def write_midi(mid,my_tracks,path):
    file = MidiFile(type=mid.type)
    file.ticks_per_beat = mid.ticks_per_beat
    for track in my_tracks:
        track_i = MidiTrack()
        #for msg in track[2]:
            #track_i.append(msg)

        for meta_msg in track[0]:
            track_i.append(meta_msg)
        for note in track[1]:
            track_i.append(note)

        file.tracks.append(track_i)

    file.save(path)
    print('wrote')

def read_midi(mid):
       my_tracks  =  []  #  It's  a  list,  where  each  row  has  [meta_msg
,notes_msg ,msgs ]
    #Each row represents a Track
    for track in mid.tracks:
        meta_msg = []
        notes_msg = []
        msgs = []
        for msg in track:
            msgs.append(msg)
            if msg.type == 'note_on':
                notes_msg.append(msg)
            else:
                meta_msg.append(msg)

        my_tracks.append([meta_msg,notes_msg,msgs])

    return my_tracks

def predict(sequece):
    x = sequece.reshape((1,n_steps,n_features))
    yhat = model.predict(x,verbose=verbose)
    return yhat

def reparador_felix_jr(tracks):
    global model
    meta_msgs, notes_msgs, flags = tracks
    model = model_stacked_lstm(n_steps,n_features)

    notes = []
    realn = []
    for msg in notes_msgs:
        notes.append(msg.note)

    notes = np.array(notes)
    max_i = 0
    for i in flags:
        if i:
            max_i = max_i + 1
    print (max_i)
    actual_i = 0
    for i, flag in enumerate(flags):
        if flag:
            start = i - (n_steps*2) if i > n_steps else 0
            train_batch(notes[start:i])
            note_predicted = predict(notes[i - n_steps:i])[0][0]
            note_predicted = 0 if note_predicted < 0 else note_predicted
               notes[i] = round(note_predicted) if note_predicted < 127
else 127
            print(actual_i, '/' , max_i)
            actual_i = actual_i + 1

            notes_msgs[i].note = notes[i]

    return tracks

model = None
n_steps = 15 # n notes used to predict n features
n_features = 1 # Only one track
epochs = 100 # n passes through the dataset
verbose = 0  #Show logs
damage_rate = 0.2 # Damage

def compare_midi(real,my_tracks,times):
    print('inside')

    for pos,track in enumerate(my_tracks):
        print("TRACK ", pos)
        p_meta_msgs, p_notes_msgs, p_flags = my_tracks[pos]
        r_meta_msgs, r_notes_msgs, r_flags = real[pos]
        acomulated_error = 0
        if len(p_notes_msgs) == 0:
            continue

        corrupted_counter = 0
        for j,p_msg_note in enumerate(p_notes_msgs):
            r_msg_note = r_notes_msgs[j]
            if p_flags[j]:
                corrupted_counter = corrupted_counter + 1
                                                       error_relativo    =
(math.fabs(r_msg_note.note-p_msg_note.note)/r_msg_note.note)*100
                          print('Real  ->',  r_msg_note.note,  '  Predicted
->',p_msg_note.note,'Error ->',error_relativo)
                acomulated_error += error_relativo

        print(acomulated_error,corrupted_counter)
        acomulated_error =  acomulated_error/corrupted_counter
        print('Error Relativo acomulado: ',acomulated_error)

        print('Time Training')
        time_acumulate = 0
        for i in range(1,len(times)):
            print('Time in track',i,' -> ',times[i])
            time_acumulate += times[i]

        print('Promedium ->',time_acumulate/(len(times)-1))



def main():
    global model, n_steps, n_features
    n_steps,n_features = 10,1
    diomio_number = '9'

    # Read the file
    mid = MidiFile('midi_partitures/happy.mid')

    my_tracks = read_midi(mid)
    real = copy.deepcopy(my_tracks)

    #Dañar
    for track in my_tracks:
        notes = track[1]
        track[1],track[2] = ralph(notes, damage_rate)

    #escribir
    path = mid.filename.split('.')
    path = path[0] + '_broken.' + path[1]
    write_midi(mid,my_tracks, path)

    model = model_stacked_lstm(n_steps,n_features)
    #Reparar

    times = []
    for pos,track in enumerate(my_tracks):
        print("TRACK ", pos)
        actual_time = time.time()
        my_tracks[pos] = reparador_felix_jr(track)
        delta_time = time.time()  - actual_time
        print('Delta time ->', delta_time)
        times.append(delta_time)





    # Write the song.
    path = mid.filename.split('.')
    path = path[0] + '_fixed.' + path[1]
    write_midi(mid,my_tracks, path)

    compare_midi(real,my_tracks,times)




if __name__ == "__main__":
    main()
```

References

[1] C.Z. Francisco, "Métodos Numéricos". *Medellín: Fondo Editorial Universidad EAFIT*, 2010.

[2] J.F. Epperson, "On the Runge example", *The American Mathematical Monthly*, 94(4), pp.329-341, 1987.

[3] C. Brezinski & M.R. Zaglia, "Extrapolation methods: theory and practice (Vol. 2)". *Elsevier*, 2013.

Bibliography

M. Lagrange, S. Marchand & J.B. Rault. “Long interpolation of audio signals using linear prediction in sinusoidal modeling”. *Journal of the Audio Engineering Society*, 53(10), pp.891-905, 2005.

J.P. Berrut & L.N. Trefethen, “Barycentric lagrange interpolation”, *SIAM review*, 46(3), pp.501-517, 2004.