May 2022

# SOUND SYNTHESIS WITH DEEP-AUTOENCODERS, DISCOVERING NEW SOUND DESCRIPTORS

*Supervised by Dr. Thomas Blumensath*

**Erik Valls Muñoz** **ID: 30405351**

Word Count: 8422 words

"This report is submitted in partial fulfilment of the requirements for the BEng Aeronautics and Astronautics, Faculty of Engineering and Physical Sciences, University of Southampton".

*I, Erik Valls declare that this thesis and the work presented in it are my own and has*

*been generated by me as the result of my own original research.*

*I confirm that:*

*1. This work was done wholly or mainly while in candidature for a degree at this University.*

*2. Where any part of this thesis has previously been submitted for any other qualification at this University or any other institution, this has been clearly stated.*

*3. Where I have consulted the published work of others, this is always clearly attributed.*

*4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.*

*5. I have acknowledged all main sources of help.*

*6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.*

*7. Either none of this work has been published before submission, or parts of this work have been published as: [please list references below]:*

# ABSTRACT

Traditional sound synthesis methods rely on very complicated parameters that revolve around the use of indefinite numbers of wave generators and envelopes, that make the process of translating ideas into sound quite a hard challenge. Several studies have reported the use of machine learning, in fact deep learning, for sound synthesis. With the purpose of providing with more straightforward ways of generating sound by simply describing them as they are. Using a convolutional neural network, in this work we try to find the very depths of how a large dataset of sound samples composed of different single notes played by different instruments is encoded, to find how the information regarding the timbre and the pitch is compressed in the most simplistic representation of such samples. It was also explored how different functions generate sounds when decompressed. Although no clear concise rules were managed to converge, quite a few indications of how the values inside the compressed representations affect the output sound were obtained by the analysis of the spectrograms.

# ACKNOWLEDGEMENTS

First to start off I would like to thank my research supervisor, Dr. Thomas Blumensath, I gratefully acknowledge the help provided and all the professional guidance and valuable support throughout the development of this project.

I am indebted grateful to Dr. Paul R. White, my personal academic tutor, for supporting throughout the whole degree, specially through part 1, and through such a difficult course given the circumstances.

I would like to thank my housemates Emma and Lily, for being all I needed during this year. I would not have been able to finish if it wasn't for you both. I would like to mention my former university fellow Norman, I could never express how much you meant during first year.

I very much like to thank my band mates from *The Break*, as being one of the most important aspirations to keep me going every day and giving me the opportunity to fully share my passion with them. Among others, I want to thank Nico, Moi and Martin for being such a personal reference to follow and for all the emotional support provided; and Jorge and Sara, I would have never finished this without you.

Last but not least, the most gratitude to my parents, and brother. To you I owe my life and world.

To all of you, thanks for inspiring me to become a better person every single day.

# CONTENTS

# 1 INTRODUCTION

Sound is a pressure wave that travels along a medium. The simplest sound is a pure tone, graphically described as a sine wave, but real-life sounds are infinitely more complex thanjust a sine wave. Complexity emanates from, firstly, the harmonic structure, closely related to the timbre; and secondly, the way sound evolves with time, for it is not a static event. For example, when we play a note on the piano it slowly fades out, and if we take a closer look, it is only the low harmonics that are preserved, and the high the ones that start to face sooner. The way the harmonic structure evolves with time is also part of the timbre. (White, 1994)

But what is timbre? When stating the word timbre, we are referring to the sum of sound characteristics that let us identify different sound sources, such as different voices or instruments; the reason why we can sometimes differentiate the caller just by hearing their voice. But how do sources sound different if they are generating the same pitch? Because sound is formed by a whole series of tones known as harmonics, the harmonic structure as mentioned before.

Starting from the basic sine wave, as we add more higher frequency sine waves to recreate the overtones, we can successfully reproduce an instrument timbre. This is known as additive synthesizing. In more technical terms, sine wave generators are configured differently to recreate the harmonic structure of an instrument. The use of wave generators to make music can be dated back to the late 19$^{th}$ century by the hand of instruments like the Telharmonium or the Theremin, and its rapid rise from then transformed music synthesis into an essential requirement in modern music. Although the start of sound modulation and sound synthesis was placed in the early 60s by the hand of Robert Moog (Lee, 2018). By modifying factors and parameters that control both the harmonic structure and the evolution of the sound as time flows when playing a note, the instrument timbre is defined.

It is no child game trying to reproduce a sound by recreating the timbre, and this might be frustrating when wanting to reproduce a very specific idea that cannot be directly related to the low-level parameters mentioned before. Sound synthesis gets more technical the deeper we dive into sound exploration techniques, which involve Low Frequency Oscillators (LFO), envelopes, etc. Synthesizer parameters increase exponentially as we get more specific with the sound we are trying to achieve when, for example, making music. The aim of this study is to evaluate in what possible way can we translate our imaginary description of sounds into the synthesizer or sound generator. Timbre is one of those tricky aspects that are so naturally written in our brains that deconstructing the timbre out of our own brain is a massively difficult puzzle.

Using convolutional networks, which are based on machine learning, it might be possible to use the data stored in compressed audio files to find out where characteristic information related to the sound is stored (this is pitch, volume, or timbre), which will allow for new descriptors and parameters to be discovered and used in sound synthesis.

In this work, a deep convolutional autoencoder is used with the aim of discovering where the sound properties are stored inside the hidden layers and subsequently searching for new sound descriptors that might arise from these hidden layers. For example, how muffled does a sound feel, or how metallic. To achieve this, a training dataset composed of 4000 audio files of different instruments and playing different notes will be used. Once training is complete, the encoder part is used to generate smaller dimensional data of the files and we compare them using different criteria in search of patterns or rules that are similar in the compressed data. This paper investigates a similar subject as Roche's work (Roche, et al., 2021), which uses supervised learning by applying labels to a known dataset, but focusing on undefined aspects of sound through unsupervised learning, in search for new sound descriptors, and furthermore, can we establish a relationship between sound properties and its compressed hidden representation?

This paper is organized into 5 sections. The first section gives a brief overview of the background information and settles the context of this study. The second section follows a description of the methodology used, along with the tools and programs that helped with the development of the project. The third section describes the results obtained, followed by the conclusions raised from them in the fourth section. Limitations and new methodology proposals are noted in the last section where how new methods and slight improvements could further complete this paper.

# 2 BACKGROUND

Traditional sound synthesis methods rely on using sine waves and noise, which electronically are defined as oscillators and noise generators. Noise is a randomly generated function that gives a harmonically rich frequency spectrum. Synthesizing can also be done by subtraction, which is very similar to what an artist would call sculpting, where we generate a noise wave, and we start subtracting frequency by using filters and envelopes. Normally, when synthesizing a simple sound, it is done by specifying low-level parameters (attack, release, decay…) that control an envelope over the sine waves generated at a

*Figure 1. A traditional envelope with attack, decay, sustain and release controls. [Ableton live 10]*

determined frequency (pitch) and a user specified velocity, also knowns as intensity or volume (Colonel, et al., 2018). Furthermore, in order to give more expression, synthesizers use auxiliary sine waves (Low frequency oscillator [LFO]) to control the main sine wave parameters, such as detuning or a tremolo effect (volume increase and decrease pulses that give a vibrant character). The envelopes can be transposed not only to the sine waves but also to the filters, as they are not explicit. This expands the possibilities, and the combinations are almost infinite. This also makes sound synthesis a much more complex method than just asking the program to recreate a saxophone, for example.

On the other hand, machine learning is a process in which computers learn and improve upon specific parameters and rules from an input data based on a user defined performance measure. This leads to algorithms that are used for a specific kind of task; when combining a large series of algorithms, we talk about a complex system that learns and improves simultaneously called Artificial Intelligence (AI) (Somogyi, 2021). The main difference with a conventional computer program resides in its ability to adapt and learn new rules based on new training data or experience. Where and how these changes in the internals of the program can also by user controlled and fixed as desired.

Three main machine learning categories can be distinguished: Supervised learning includes labels in the input data, and it's mostly used for classification or regression. If the labels are not present, it is recognised as unsupervised learning. The third type, reinforced learning, introduces user input decision making as new data to be learned, so the rules of the program are constructed by experience in different situations, for example, in self driving cars. (Somogyi, 2021)

There are many methods for machine learning, all based in the Feed Forward Network, where there is no type of feedback loop. This means data only moves through one direction and it is therefore the simplest type of artificial neural network. They are commonly composed of few layers, the basic building blocks of neural networks, but as soon as there are a significant number of layers it can be classified as Deep Learning. Deep Learning allows for a larger number of parameters to be learned from the data, and therefore are more suitable for modern applications such as face recognition or speech recognition (Bhattacharyya, 2020).

According to IBM, '*Deep learning attempts to mimic the human brain—albeit far from matching its ability—enabling systems to cluster data and make predictions with incredible accuracy.'* It is essentially defined as a neural network with three or more layers, which help with optimization and accuracy. Deep learning distinguishes itself from machine learning by the type of data and the methods used for learning. Machine learning uses labelled data to make predictions (supervised learning), so it requires some pre-processing to organize the data into

a structured format; meanwhile, deep learning eliminates some of the pro-processing involved with machine learning (unsupervised learning). (IBM Cloud Education, 2020). The opens up the liberty of the program to create its own rules.

In this project, the concept of the autoencoder is used. A type of unsupervised learning system composed of two symmetrical Convolutional Neural Networks (CNN): the encoder and the decoder. CNNs are a Deep learning algorithm that take input data and assign an importance to aspects in the data in order to differentiate from one another. They learn these filters with no user input and are analogous to the connectivity pattern in neurons. The main difference with FFNs (Feed Forward Networks) is that they can capture the spatial and temporal dependencies and therefore they can reduce the data without losing critical features for good prediction. In other words, it is possible to have similar scores if we flatten a matrix and run it through a FFN compared to a CNNs, but this is only true for extremely basic binary images. (Saha, 2018)

Imagine for an instance we have a 25x25 pixels black and white image. If we use an FFN, it would firstly flatten this image (converting it into a $25^2$ long vector) and then process it, reducing its dimension (from $25^2$ to $10^2$ and so on), as seen in Figure 2. If we use a CNN, the image will be reduced with its original dimension (from 25x25 to 10x10 to 5x5 and so on) through a series of layers. The comparison between the two neural networks would result in very similar performances, but if the image were in colour, the CNN would drastically outperform the FFN.

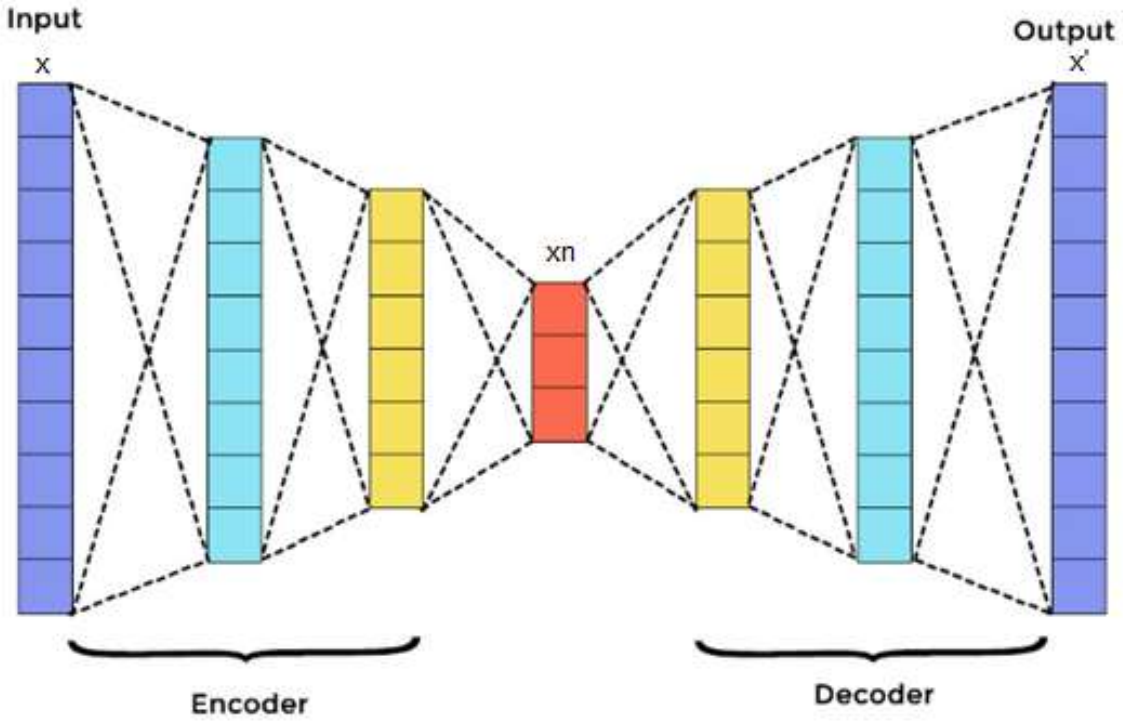



*Figure 2. FFN generic structure*

CNNs are defined as a multiple layer deep learning architecture where the output of the *n* layer is the input for the *n+1* layer, very widely used for compression purposes. The encoder is used for compression, shrinking the dimension of the input using discrete number of values; and the decoder expands the dimension to reconstruct the original input (Colonel, et al., 2018). All machine learning methods develop themselves upon a user defined performance measure, a loss function, which measures the difference between the output layer and input data for the autoencoder to tune its parameters and improve performance.

For example, a single layer encoder would map the input *x* into a hidden layer *y* using an activation function *f* which imposes non-linearity in the form followed by equation 1.

$$y = f(Wx + b)$$

***Eq. (1)***

Then the single layer decoder would map the hidden layer *y* into the output *x'* by using another activation function *f* following equation 2.

$$x' = f(W'y + b')$$

***Eq. (2)***

Where *W* and *b* are called weights, which are the variable parameters that the autoencoder tunes to minimize the loss function and reconstruct the input data. For a deep learning architecture, it shows the same functionality, where the encoder contains various layers using equation 1 for each one, mapping $x \rightarrow x_1 \rightarrow \cdots \rightarrow x_n$. Treating $x_n$ as *y* then the decoder maps $x_n \rightarrow x_{n+1} \rightarrow \cdots x'$.

Autoencoders can also be used for different purposes, depending on the loss function chosen. It directly controls the way the autoencoder behaves during training, if it seeks, for example, classification or reconstruction. But overall, they all work upon a specific input dataset, which can be in any form of shape.

Some of the first studies of machine learning in audio synthesis tasks were considered by Google's WaveNet Architecture using convolutional neural networks that generate raw audio waveforms (Oord, et al., 2016). It was then furtherly developed by another Google project called Magenta, which uses autoencoders to interpolate sound timbres between two instruments. The project was developed to a fully functional synthesizer called NSynth, and it is now available for the use of musicians (Engel, et al., 2017).

Magenta evolved to an *open-source research project exploring the role of machine learning as a tool in the creative process,* as stated un the website. (Magenta, s.f.). They offer a way

of posting recent projects relating machine learning, such as a very recent work called *Paint with Music*, which translates the drawings made on a canvas into musical notes performed by a choice instrument. This allows for new forms of expression as the only tool in your hands is how you stroke the pencil while painting. (Doury & Buttet, 2022)

This last two examples require very high-end computing power, but in the past 5 years there has been a rapid rise in the use of machine learning that require significantly less computer processing speed. Some preliminary work was carried out by Colonel, where a light weight deep autoencoder was used for sound synthesis (Colonel, et al., 2018). The autoencoder would compress and reconstruct magnitude short time Fourier transform frames of audio in order to control the waveform and develop new user approachable synthesizers.

These examples are focused on new sound synthesis, others are focused on the discovery of timbre control parameters. Various approaches have been proposed to further analyse this. In a major advance in 2021, the first investigations in timbre identification were studied in Roche's work, where labels for the dataset were obtained by listeners directly characterising sounds along eight perceptual dimensions. Then an autoencoder was trained using these labels addressed to each sound. The work succeeded to an extent to capture acoustic properties of these perceptual dimensions. Roche's work fully justifies the fact that information of this calibre is kept through the encoding of the sound files, but it is not yet known how and where it is kept. (Roche, et al., 2021). But how well would an autoencoder perform in an unsupervised environment when trying to encode the features that are labelled in Roche's work? Instead of giving already known labels and descriptions of the sound files, in this work we give free hand for the autoencoder to create its own differentiation.

# 3 METHODOLOGY

As mentioned before, we are going create a neural network that can process and reconstruct audio files in order to investigate the way general information from the sound data is encoded. We are going to need several tools and modules to achieve this, as well as a well organised dataset and some logic architecture for the neural network. The next step consists of training and then, once the performance criteria are achieved inside the neural network, testing and data acquisition will take place.

## 3.1 COMPUTATIONAL TOOLS

The software package used to develop this project was Python 3.6, with the help of Jupyter Notebooks. In order to construct a functional autoencoder a series of external modules are

needed (Rossum, et al., 1995). They all work between each other to smooth the workflow. The modules used are:

- NumPy is a basic package in all Python operations. Its main advantage is the ability to work with N-dimensional matrices as big as needed, known more commonly as Arrays; and offers a large variety of modules such as SciPy that are very useful for example, for signal processing. (Harris, et al., 2020)
- IPython is a powerful interactive shell that provides interactive data visualization and has a functionality to reproduce sounds in the Jupyter notebook itself. (Pérez & Granger, 2007)
- Matplotlib is another basic library for creating visualizations in Python, and it makes contents much easier to understand and compare when using plots. (Hunter, 2007)
- TensorFlow is an open-source end-to-end machine learning library for production and research. It offers a wide variety of beginner level APIs. The architecture of TensorFlow is divided into four functioning parts: data processor, model builder, training, and estimating the model performance based on a selection of different loss functions. It works with dimensional arrays. (TensorFlow.org, 2015)
- Librosa is a music and audio analysis Python package capable of translating raw audio files into NumPy arrays. TensorFlow autoencoders could simply work with time-series representations, which is the raw audio file, but it would require high computational power as the files would get very large. Librosa offers the ability to transform these files into multidimensional time-frequency representations so the autoencoder architecture can be kept much smaller. (McFee, et al., 2015)
- Keras is an open-source neural network library developed by François Chollet, fully functional on Python constructed on top of TensorFlow. It generally aids TensorFlow functionalities and makes much more user friendly. (Chollet, et al., 2015)
- Scipy is a fundamental algorithm tool for scientific computing with Python. It will allow us to perform Fast Fourier Transforms to further analyse sound properties. (Virtanen, et al., 2020)

All this software is free to use and available to download in every package's website.

## 3.2 LEARNING TASK DESCRIPTION

During the project, an intent to make the autoencoder learn low dimensional representations of musical audio will be made. Reducing the dimensional space will lead to variables in the hidden layers representation that might be related to sound characteristics and therefore can be investigated in the hunt of relevant key sound modifiers.

## 3.3 INPUTS

For the purpose of this project, a sufficiently large set of data was obtained from NSynth website, composed of 4 seconds long raw audio files. There are 4096 samples of monophonic 16kHz snippets of varying pitch, timbre, and velocities. (Engel, et al., 2017). They are unlabelled, and therefore it cannot be used for classification.

Data management was performed in Jupyter Notebooks. In this piece of work, we are not interested in sound duration, but in aspects of the sound such as the frequency and of course its timbre, so all the audio files were cropped to half a second each for training optimization, as it reduced by a factor of 10 the training time. Furthermore, the Librosa's Short Time Fourier Transformation was used to produce the spectrogram of every one of the audio files by using a NFFT (length of windowed signal) was set to 123, to ensure the uniformity of array dimensions. The hop length controls how much of the previous time frame intersects with the next one, a factor of 50% was chosen so that all data is weighted equally. Finally, a Hann window was used as it is a standard for sound processing. (Alvar M. Kabe, 2020)

This was done for all audio files and the data was presented in a large array of the shape 4096x64x64x2.

## 3.4 ARCHITECTURE

Keras library was used for the model construction once the data was ready. A deep-learning convolutional network was constructed, divided in two autoencoders. In between layers, a normalization layer was introduced. The first one was dedicated to transforming the input data into a small vector without sacrificing data integrity, and the second one reduces this vector to the minimum possible dimension preserving pitch information at least. It is expected that the quality of the sound would be strongly affected, but since it is not of the interest of this work, it is possible to ignore this loss to an extent.

### 3.4.1 Autoencoder 1

The encoder receives a 64x64x2 input array, it is then normalized and reduced by a factor of 2 along four 2D convolutional layers to a final dimension of 16x16x8. It was then flattened to a 2048 long one-dimensional vector. The compression ratio, or how much the data is compressed is merely decided by comparing the output to the input through trial and error. With this first autoencoder, the aim is to get the smallest encoded representation with highest reconstruction fidelity. Every layer reduced the dimensions by a half for simplicity, and it was found that the limit for this sound fidelity to be maintained was four layers, which reduced the dataset to a dimension of 16x16x8. The layers are also padded.

The decoder then uses the same architecture on reverse, transforming the 2048 vector into a 64x64x2 array as the original input was, using 2D transposing layers.

See appendix A for structural scheme

### 3.4.2 Autoencoder 2

The encoder ("shortener") receives the previous 2048 unidimensional vector and uses one dense layer for its reduction to a 256 long vector. In this case, we try to reduce even more the dimension without considering sound fidelity a priority, and it was also found by trial and error that the limit for the correct pitch to be reconstructed was one layer with a reduction factor of 8.

The decoder ("reshaper") uses the same method to decompress the shortened vector back to the original 2048 length.

See appendix B for structural scheme.

### 3.4.3 ReLU activation function

For the autoencoder weights to converge during training, the Rectified Linear Unit activation function was used (ReLU) for all layers but the last one of each decoding part of both autoencoders. (Nair & Hinton, 2010). It is formulated as seen in equation 3.

$$f(x) = \begin{Bmatrix} 0\,, x < 0 \\ x, x \geq 0 \end{Bmatrix}$$

***Eq. (3)***

This means that the function will output the input directly if it is positive, otherwise it will output zero. It is the default activation function in machine learning due to its easiness of training in models and if offers an overall better performance (Brownlee, 2020). There were tests performed using different functions like the *leaky ReLU* and the *sigmoid*, but they directly outputted either background hiss noise or very distorted representations when trying to reconstruct a test data sample. In fact, the problem with other functions like the sigmoid in this case and the hyperbolic tangent is that they cannot be used with multi-layer networks because of a vanishing gradient problem, which is overcome with the ReLU.

## 3.5 Training

The training was developed using a 20% validation split on the full 4096 dataset length, with a batch size of 20 along 1000 epochs. Adam method for stochastic gradient descent was implemented and the loss function was set to mean squared error. The autoencoder was trained on a NVIDIA GeForce GTX 1060.

For the second autoencoder, the same configuration was used but it was trained for 200 epochs, unlike the 1000 from the first autoencoder.

The autoencoders were then saved as individual files so the weights were maintained.

## 3.6 PERFORMANCE OVERVIEW

The first autoencoder achieved a loss of 0.3544 with a validation loss of 0.25. Testing the autoencoder and comparing the sound generated against the original, the result matches quite well. As mentioned before, we are not interested in the quality of the reconstruction but in the sound properties.

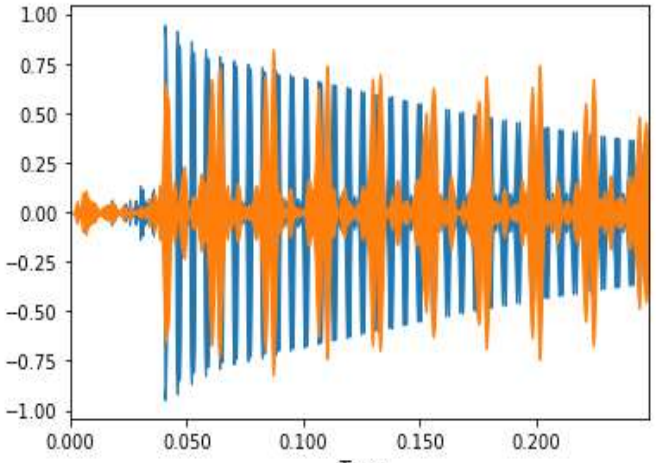



*Figure 3. Waveform comparison between original (blue) and reconstructed (orange) audio*

Figure 3 shows the waveform comparison, while figure 4 shows both the power-intensity spectrogram and the FFT, both extracted only using the first autoencoder.

As seen in the FFT, the harmonic structure remains almost unchanged, which means a very good performance for the aim of the first autoencoder. However, it is clear how the spectrogram and the waveform graph might show that the performance is not as good.

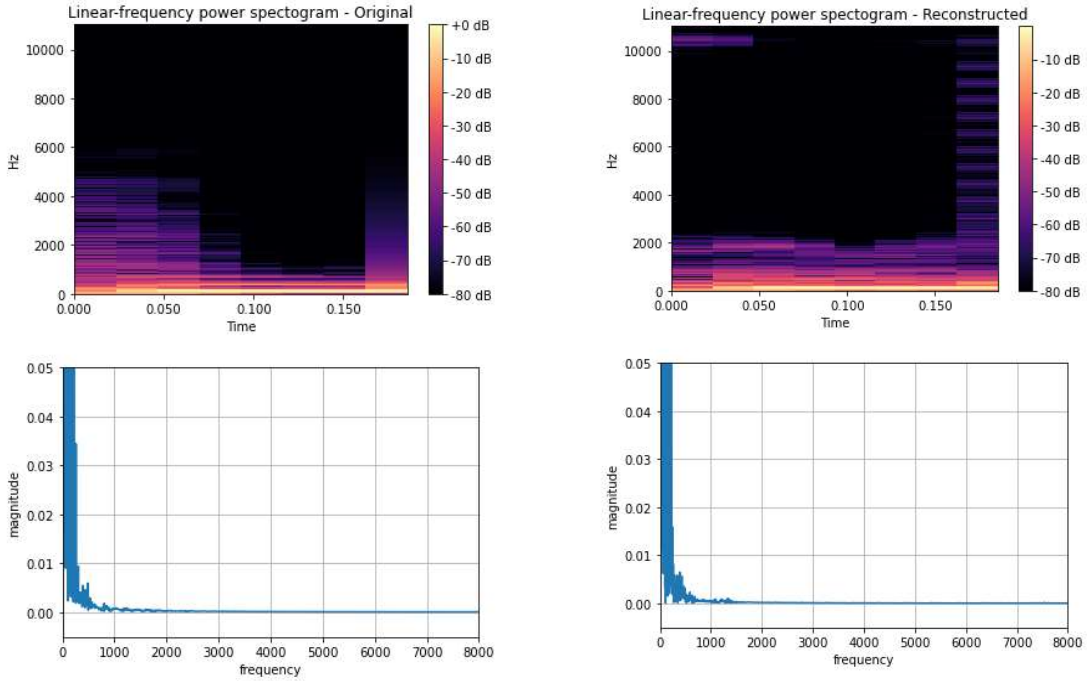



*Figure 4. Spectrograms for original (left) and reconstructed (right)*

The second autoencoder was trained for 200 epochs and returned a loss of 0.3027 and a validation loss of 0.5809. The dimensional reduction of this second autoencoder meant an increase in the loss, however, it still performed quite well in terms of reconstruction, keeping the pitch as explained in section 3.4.

Training times are in the order of an hour for the first autoencoder, and 10 minutes for the second one.

## 3.7 TESTING

Once both autoencoders were trained and ready to compress and decompress audio files, it was time for testing. There were two methods carried away during the testing process.

### 3.7.1 Direct comparison

The main objective is to look for similarities between the compressed vectors of samples from the dataset that present similar characteristics like pitch or timbre, by plotting each to identify what areas are common and which ones are different.

Firstly, samples from the dataset are selected and grouped by known characteristics (pitch, intensity and timbre, or instrument). They are processed by the two encoders and compressed into a 256- length vector. The grouped samples need to be of the same pitch or played by the same instrument, since we want to identify the areas in the vector that characterize that pitch or instrument.

This will be achieved by Principal Component Analysis (PCA). Generically, it is a method that allows to summarize the information content in large data tables. Implementing it to Python, it returns the main common areas between a batch of arrays, helping to identify correlations between data points (Sartorius, 2020). So, all instruments that play the same note in the dataset are arranged and their compressed vectorial representation is compared using PCA to find their similarities, known as principal components. The same process is done comparing all the pitch range of the same instrument. Before that, the compressed vector values were normalized between 0 and 1 dividing them by their highest value respectively. This was done for 19 different instruments available from the dataset and 5 different musical notes. The selection was merely arbitrary, but in most cases batches that had less than 15 samples were avoided.

### 3.7.2 Random function generation

The other method is to generate a 256 vector on a NumPy array and decompress it using the decoder. A sound is generated and listened to using IPython tools. Then characteristics of the sound produced were noted and this was repeated for many combinations of vectors using

known functions such as sine, cosine or linear. The vector plot gives no relevant information about the sound output, and therefore the power-frequency spectrogram was also analysed for a series of small variations of small functions.

The sine and cosine waves were tested for different amplitudes and natural frequencies, and the linear functions were changed in slope and sign, as well as the origin offset. All these factors contributed on changes in the spectrogram that were clearly visible. The spectrograms were generated in batches with the help of Jupyter notebooks by using for loops, so the spectrogram images can be cycled as if we were using a potentiometer to change a certain value, which was quite useful for identifying how the spectrograms evolves progressively with the parameters altered.

The Fast Fourier Transforms (FFT) of the output audio will also allow us to identify the harmonic structures of the sound along with the spectrogram and check whether different functions are related in terms of their frequency characteristics. Furthermore, it is a useful tool to track how the harmonic structure evolves with small parameter changes, in order to identify if there is a correlation between the vector values and the sound generated.

# 4 RESULTS

## 4.1 COMPARISON OF INSTRUMENTS USING PCA

The comparison as made for arbitrary notes and instruments. None of the batches gave enough relevant data to find any patterns or rules in the vector formation that can be related to any sound descriptors. The PCA method showed a variance ratio of less than 40% on all cases, and the principal components have very little strength to be considered relevant in between batches, as seen in all the graphs in the appendix, where the principal components are in the amplitude of 20% of what the compressed vector initially is, meaning there was not a single strong component that could define a batch of the same timbre or pitch.

However, when the mallet sound was analysed (Mallet acoustic 56), the 43 samples were plotted, and it was found that the vectors were much closer to 0 than observed in any other sample batches. Furthermore, there was a common spike at 75 that was also reflected in the PCA graph. Spikes like this were common to see in the PCAs, but they did not correlate when comparing them with the vector plotting graphs of the full batch.

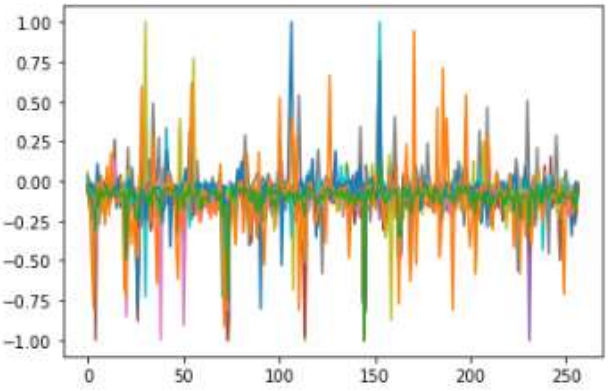



*Figure 5. Vector plotting of the Mallet acoustic 56 batch.*

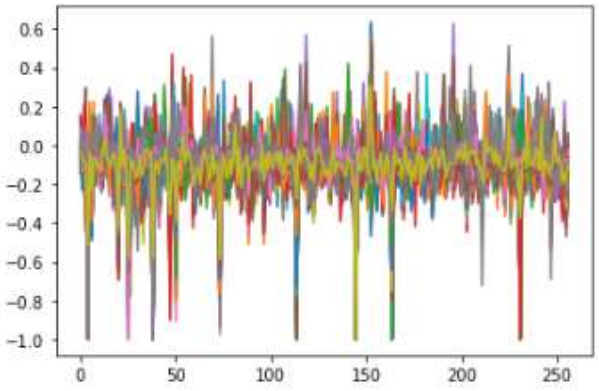



*Figure 6. Vector plotting of the Vocal Synthetic 3 batch.*

Figures 5 and 6 show the comparison of the vector plot between said timbre and an example one. It can be clearly seen that the Mallet batch is much less spread in terms of the location and number of maxima than the Vocal one.

This can be caused by percussive sounds not being strictly melodic sounds, but rhythmic, and as explained before, drums are based in white noise. This might also mean that the maxima, or even the overall amplitude of the vector, can be related to the melodic part of a sample, meaning the more the vector oscillates away from the x axis, the more pitch related information in contains. It might as well be related to the *decay* of the sound: Mallet sounds refer to what we would normally hear when playing a tom on a drumkit, which of course might have a much shorter decay than a voice sound.

## 4.2 EXPLORATION

The exploration was made for sine, cosine, linear and random functions. Following:

$$f(x) = exploration\ function$$

Where X is an array of 256 numbers from 0 to 1. Giving each point in the vector and then running it through the decoder to get an audio reconstruction. Samples of the spectrogram were selected arbitrarily trying to show the evolution through a phase. Phases are distinguished in bullet points.

### 4.2.1 POSITIVE SINE WAVES

They give a very high-frequency non-sense hiss for equation 4 for A being any number.

$$f(x) = A * \sin(x)$$

***Eq. (4)***

At this point, $f(x) = \sin(w * x)$ was explored for different values of w. The same output was perceived for the values of $w = (2,3)$, but an interesting bubbly low frequency sound was the output for $f(x) = \sin(4x)$. This function starts inducing negative values on the final vector, since until then, the function had not enough frequency to develop them.

This function was also explored for $w = (5,8,12,20,50,60,200,1000,2000,11001000)$ as arbitrary values, but there was no correlation between either the spectrograms or the sound that the vector produced, since they all produced 2 or 3 pulses sound that vary unreasonably in frequency.

When combining both steps and using equation 5, it was noticed from the sounds generated that the critical point for a bubble sound was at $f(x) = 4 * \sin(4.5x)$ some other values of *w* were used for an amplitude of 4.

$$f(x) = A * sin(w * x)$$

***Eq. (5)***

The amplitude does not change any aspect besides the overdrive or distortion of the output sound (harshness). The frequency (w) swifts the characteristic of the sounds with no clear patterns observed in neither the spectrograms nor the sounds. However, when frequency is close to 4, it was observed that the starting timeframe when the characteristic bubble sound starts if directly related to where the function crossed the X axis.

### 4.2.2 NEGATIVE SINE WAVES

The first exploration was for the values seen in equation 6.

$f(x) = -\sin(w * x)$ with $w = (1,2,3,4,6,8,10,20,30,50,200,300,1000,2000)$.

***Eq. (6)***

The output was formed by a low-pitched hum noise with a high frequency pulse at the end of the timeframe that was directly related to the slope of the function becoming positive. This was only true for $w = (2,3,4)$, at which this last made the sound much middle frequency focused. The rest of the values give no reasonable rules and they kind of feel very random besides they all were composed by 2-3 pulses of bubbly type sounds.

When adding an amplitude multiplying equation 6, a value of 4 was selected arbitrarily. Overall, the sounds became much more “metallic” and harsher, as happened for the positive sine waves. An amplitude of 20 and 100 was also carried for testing and just as the positive function it seemed to control the harshness and distortion of the output.

Amplitude has been demonstrated to also change the harshness of the sound. With low frequency functions, the positive functions generated very high frequency hiss sounds meanwhile the negative functions generated just the opposite. This can introduce that the areas of the spectrogram frequency bands where the sound is generated is related to the positive or negative values of the compressed vector. It is unsure how this rule is applied for now.

Progressive changes in the frequency of the sine and cosine functions were made and the spectrograms were studied. Tests were carried for an amplitude of 3, in a range of w between 0 and 20, in equal periods.

The results show that:

- From w = 0 to 4, only a high frequency hiss is shown in the spectrogram in figure 7.

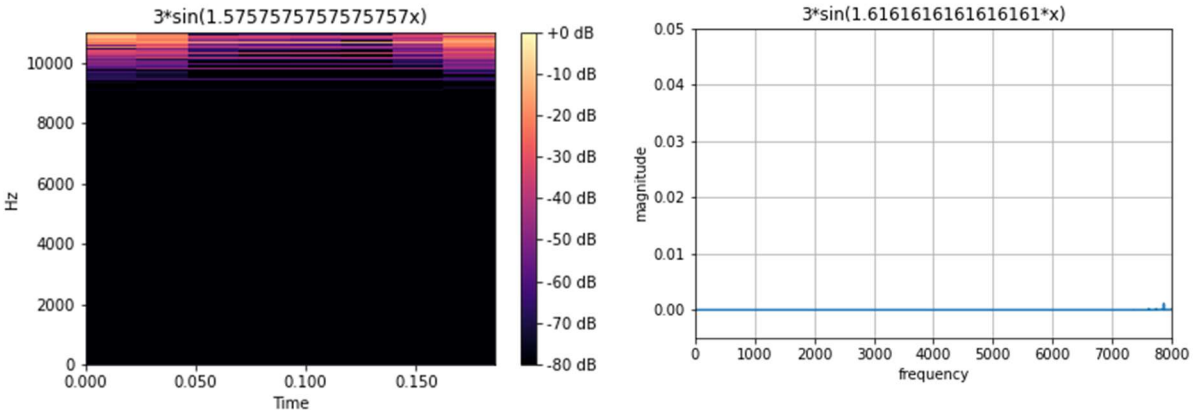



*Figure 7. Spectrogram at w =1.57*

- As soon as w surpasses the value of 4, the spectrogram shifts to the lower frequencies, and it escalates the frequency spectrum progressively until w = 5.25. seen in both examples of figure 8. The FFT shows how the same harmonic structure is only magnified as we progressively increase w.

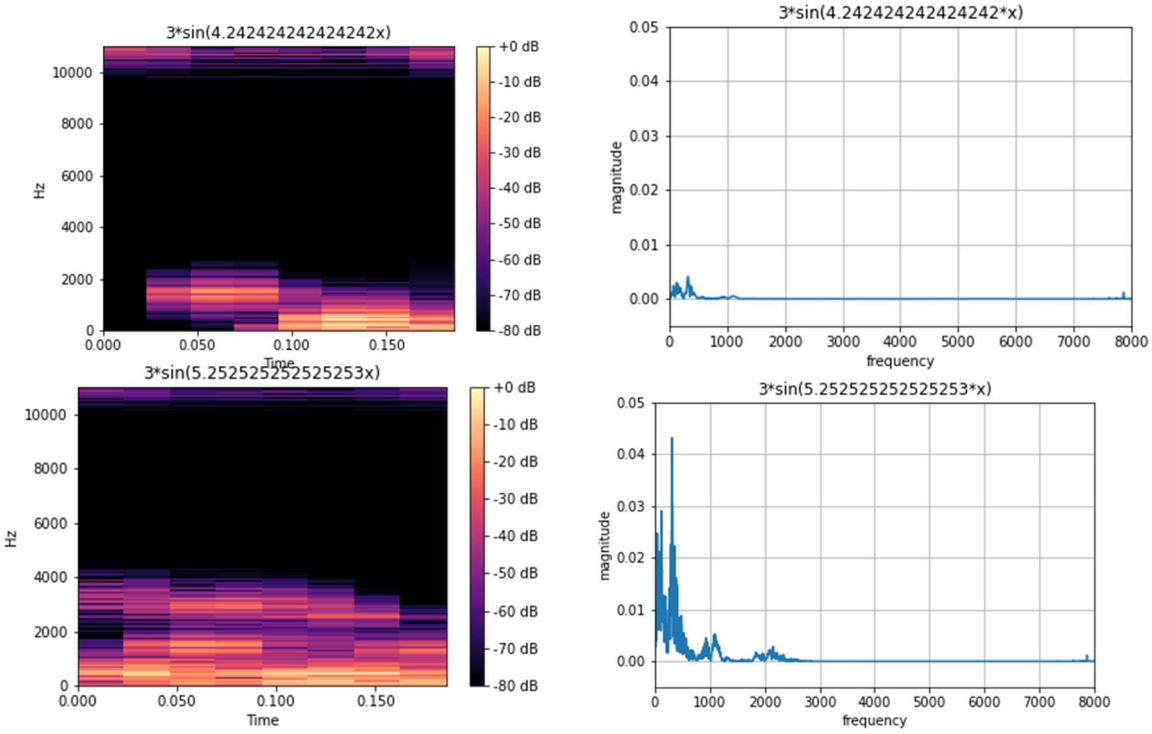
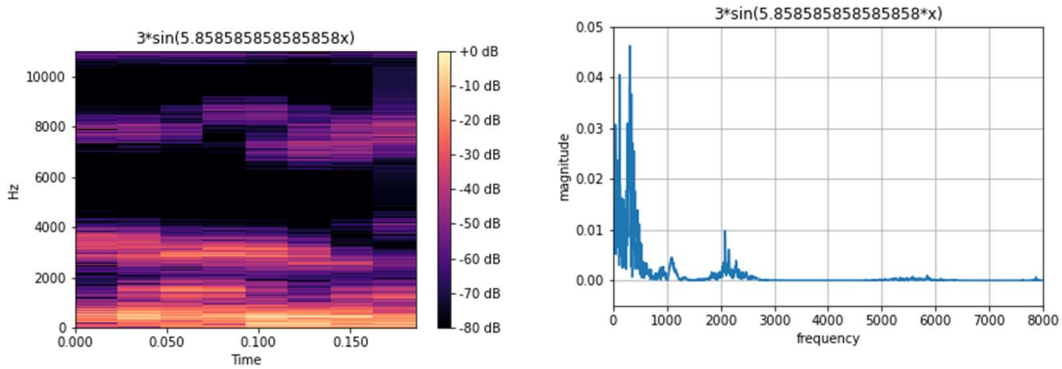



*Figure 8. Spectrogram at w = 4.24 and 5.25*

- At this stage it starts developing further areas and it leaves empty spots in the spectrogram, appreciated in example 1 of figure 9. Progressively, these areas start to fulfil and then they black out when the value of w = 10, seen in example 2 of figure 9, where 2 big full-frequency spectrum vertical bands are at both start and the end of the timeframe. It is also appreciated that from one example to another, the intensity of the spikes is greatly reduced, and some spikes like the one at 2kHz are lost.



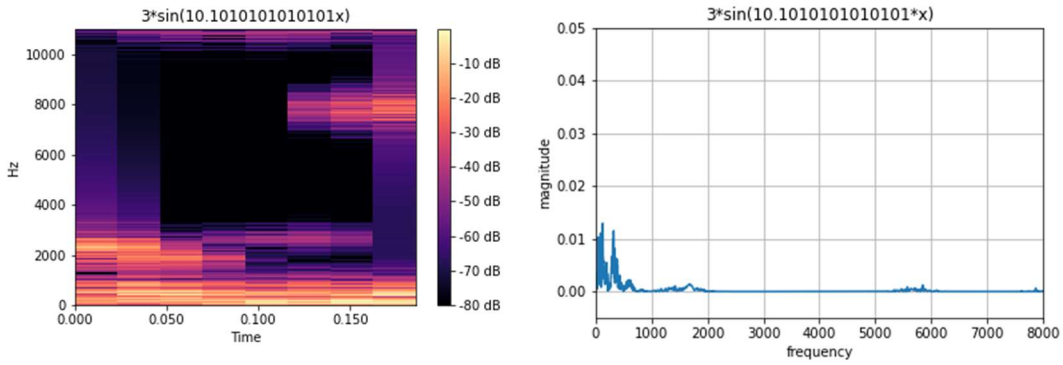



*Figure 9. Spectrogram at w =5.85 and 1.10*

- After that and up until w=20, the areas start to shift randomly without much context, as seen in figure 10. FFT shows a higher activity on the high frequencies area.

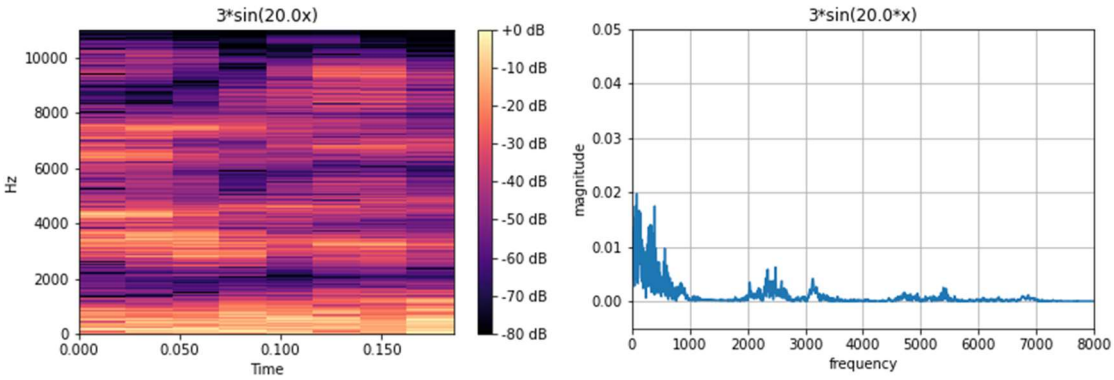



*Figure 10. Spectrogram at w = 20*

If we take the same method for the negative sine wave, the spectrogram quickly shows the full frequency coverage along the progressive change, as appreciated in the evolution through figure 11.

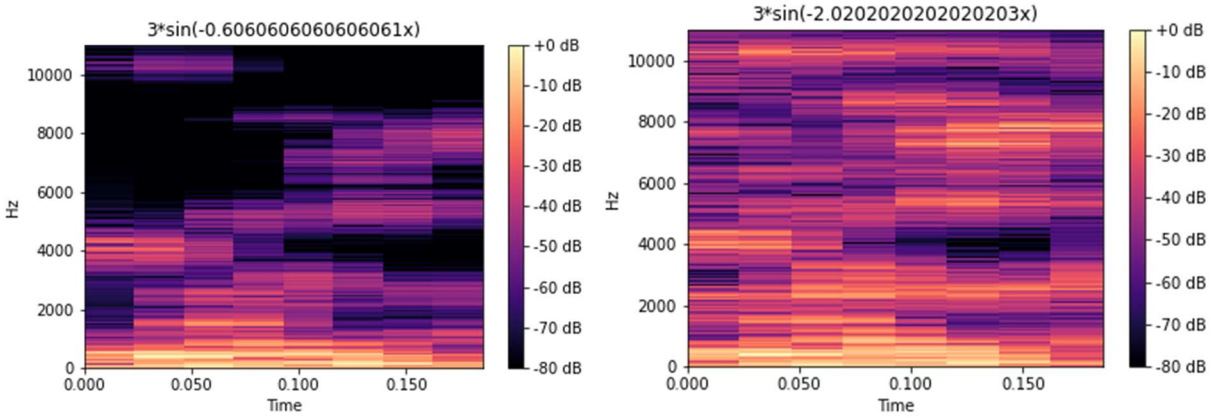



*Figure 11. Spectrograms at w=-0.6 and -2.02*

In relation to the FFT spectrograms, they show findings also visible in the linear-frequency power ones. Harmonic structures are kept slightly along batches of functions, but they also are shifted slightly between the whole sine wave coverage, for example. In comparison between the positive and negative frequency sine waves, the harmonic structure is changed drastically, but the overall behaviour and progression through the values is kept similarly.

Listening to some audio files, there are hues that are introduced progressively to the sound as we progressively travel along the variable coefficient (for example, when varying the frequency of the sine function from 4 to 5, a hue is introduced, and it is only perceivable when listening to the decompressed audio file). These hues are clearly visible for example, comparing figure 8 and 9, where the spike at 2kHz grows; and looking at the 5.5kHz band in figure 9, we can see that there are some tiny clues of sound that are not present in figure 8. These tiny clues are augmented in figure 10, where the 2kHz spike mentioned before has travelled slightly to the left.

This suggests that in a small sine wave frequency increase, the vector values at any x coordinate, change slightly, as the hues also change slightly. This means that there is a chance that certain x coordinates of the compressed vector control the hues and shift them around.

### 4.2.3 COSINE WAVES

Since cosine waves are 90 degrees offset sine waves, their properties are very similar, and they generate the same kind of sounds overall with no generic rules besides the ones mentioned above. However, the harmonic structures and development areas of the spectrograms are different from the ones appreciated in the results coming from the sine waves. Along the whole parameter coverage, harmonic structures are kept along the cosine wave frequency changes while hues are introduced and deleted around.

### 4.2.4 LINEAR FUNCTIONS

Once the variance in the sine functions was observed to be progressive developments of the spectrogram areas, the same method was then used to investigate other linear functions.

$$f(x) = m * x$$

***Eq. (7)***

The general output for equation 7 with varying *m* produces a very high-pitched whining and it has no control over the sound whatsoever. However, as soon as the offset is introduced using equation 8, some changes are observed.

$$f(x) = mx + n$$

***Eq. (8)***

The spectrograms were generated for a slope *m* of 2, with 100 different values of *n* between 0 and 3, separated equally. The upper limit was selected because the value of 3 was observed to be a threshold for oversaturated sounds. While kept under 3, the sounds did not saturate and therefore were more useful in terms of observation.

Results show that only the functions that crossed the x axis gave significant results, that is: positive slope with negative origin, and negative slope with positive origin. For the first one, it was observed that in the spectrograms the following pattern:

- For values of n between 0 and 0.5, a large band in the 10kHz and a high hiss sound (figure 12). The only thing perceivable in the FFT is a tiny spike at almost 8kHz.

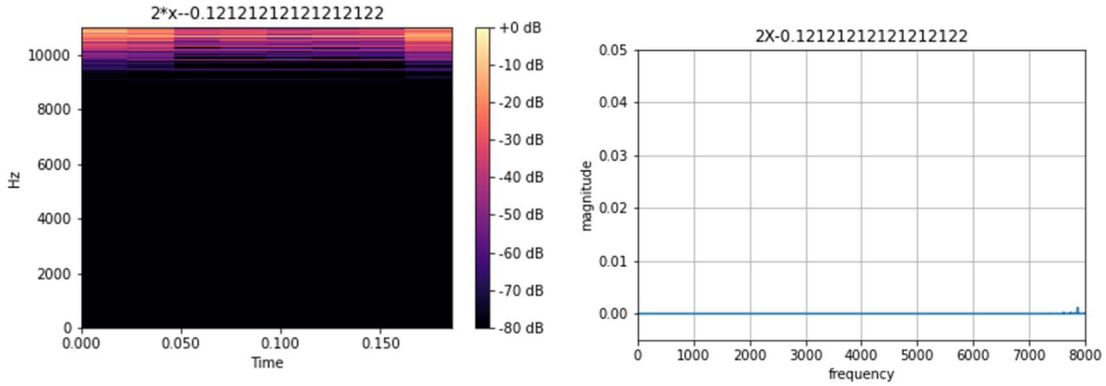



*Figure 12. Spectrogram at n = -0.12*

- Starting at n = 0.5, the band shifted from 10KHz to 100Hz as if it was displaced progressively looking at figure 13. The spikes seen in figure 12 shifted as well to the 100Hz band.

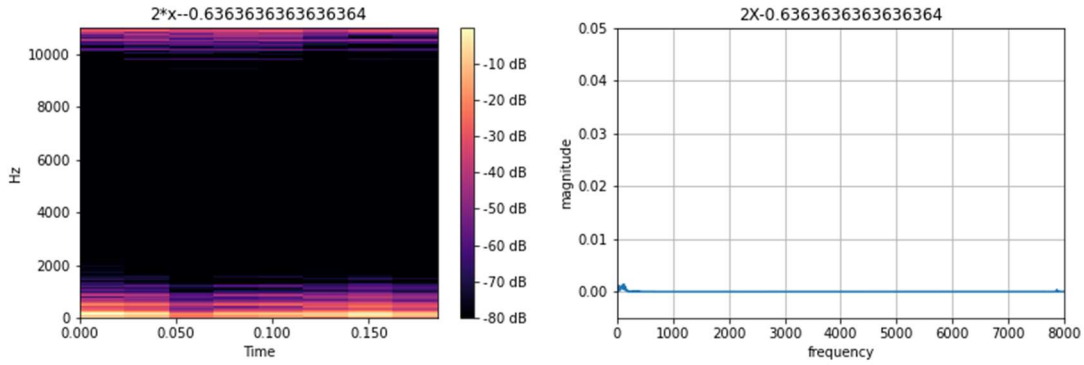



*Figure 13. Spectrogram at n = -0.63*

- In figure 14, at n=0.7, the 10kHz band disappeared leaving the low frequency band by itself, which started to develop by areas moving up in the frequency scale. These areas did not change as n was increased, but they presented higher power progressively. In the FFTs, we can see how the harmonics start to increase in intensity, but they are preserved as we increase n. The spike around 800Hz becomes clearer.

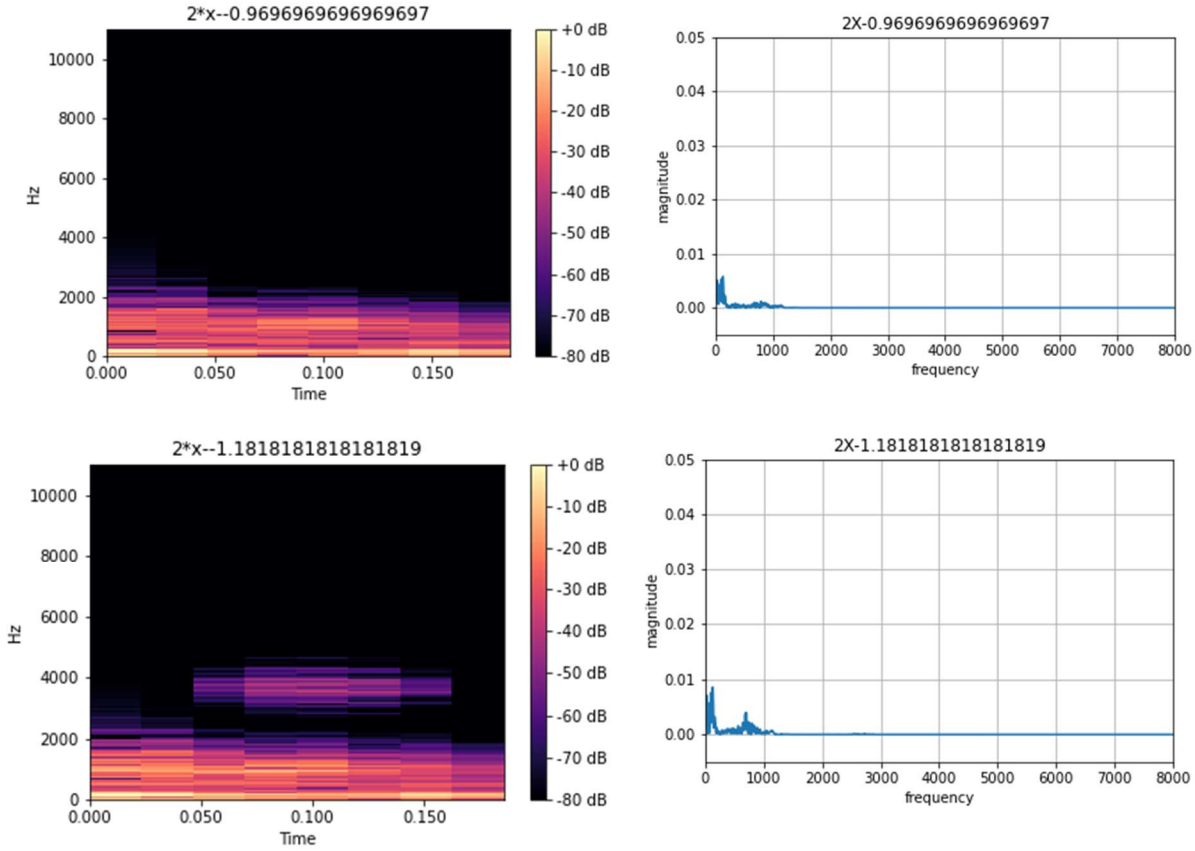
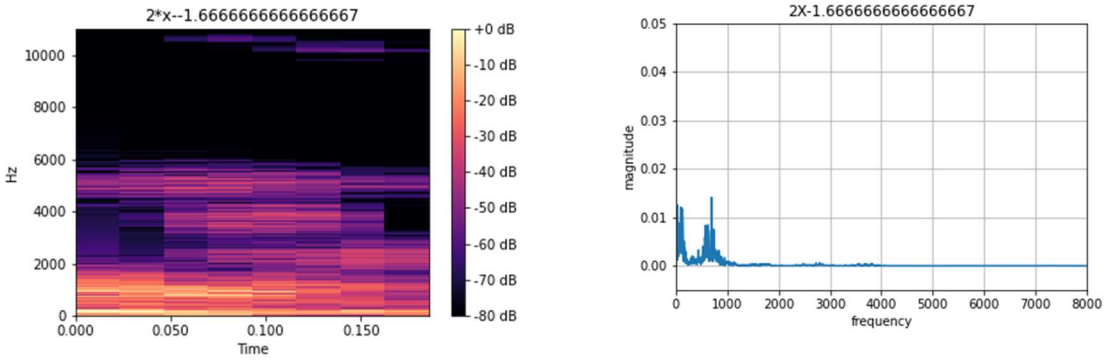



*Figure 14. Spectrogram at n = -0.96 and -1.18*

- At n= 1.63, a small band located at 11kHz started developing and kept on filling the upper 8kHz area as n was increased until the value of 3, as appreciated in both examples of figure 15.



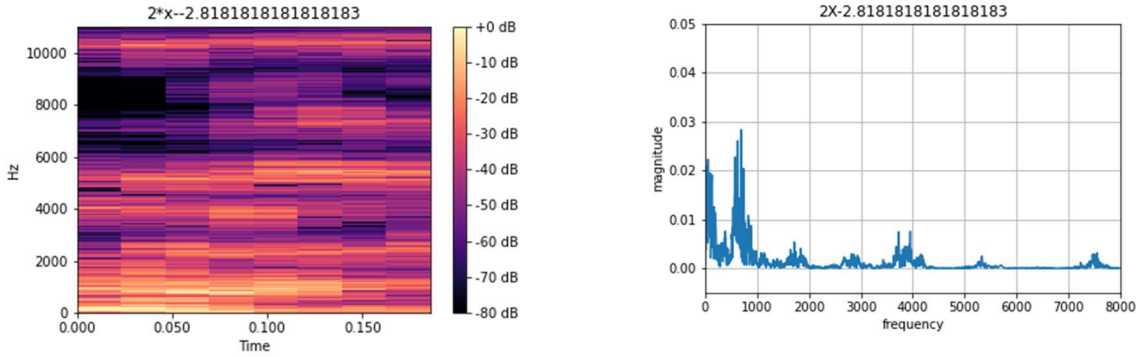



*Figure 15. Spectrogram at n = -1.66 and -2.81*

This behaviour was also imprinted on the FFT: the two spikes visible at figure 14 have now evolved to be the main characteristic of the harmonic structure, while more spikes corresponding to the hues have been introduced. At n=-1.66, we can start to appreciate a bit so tiny visible indices of new hues, becoming perfectly clear at n=-2.81.

For the second function, given by equation 8 with negative slope but positive origin, the opposite behaviour was observed:

- A few different areas are developed starting at n = 0.03, and the progressively disappear as the spectrogram grows to the low frequency bands. At n = 1, the last area is almost disappeared, and all the developed areas are compressed below 2kHz. As seen in figure 16.

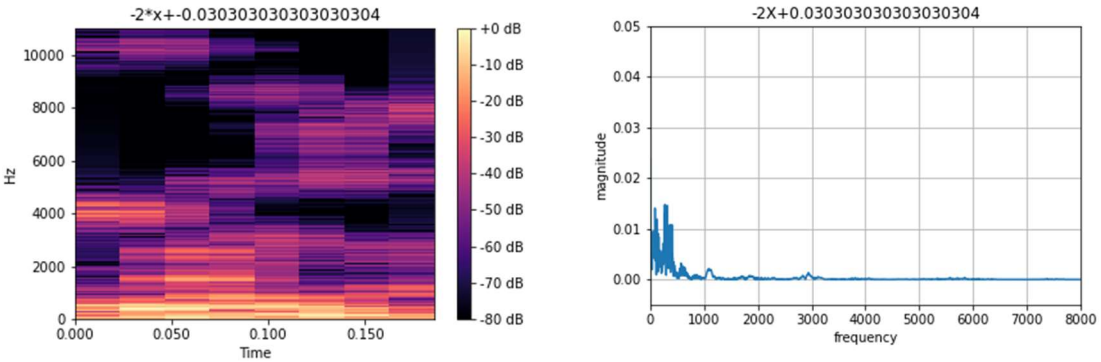



*Figure 16. Spectrogram at n =0.03*

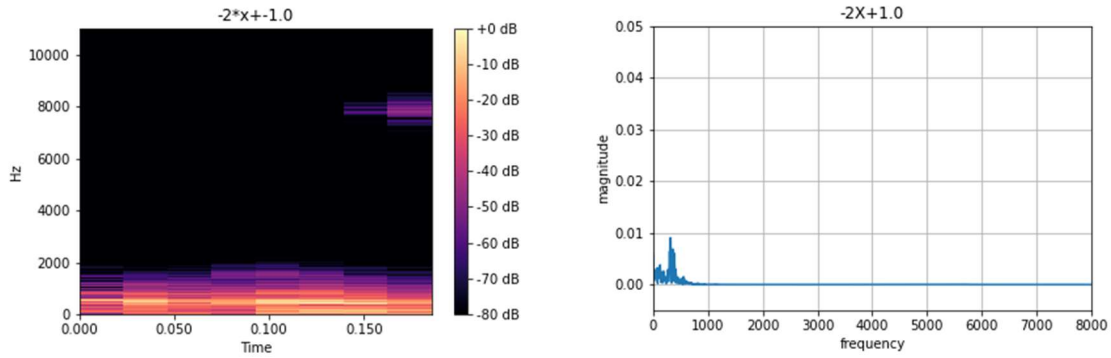



*Figure 17. Spectrogram at n = 1*

The same behaviour is seen in the FFT. The harmonic structure remains equal, and as we progress through the n values, the spikes' intensity decreases proportionally leaving the most intense spikes in figure 17 to be the only visible ones in figure 16.

- From here, starting at n = 1.21 (figure 18), the band shifts to the 10kHz area as if it was displaced progressively, just like happened with the above function. At n = 1.54, the lower frequency band has completely disappeared, leaving high frequency hiss sound all along.

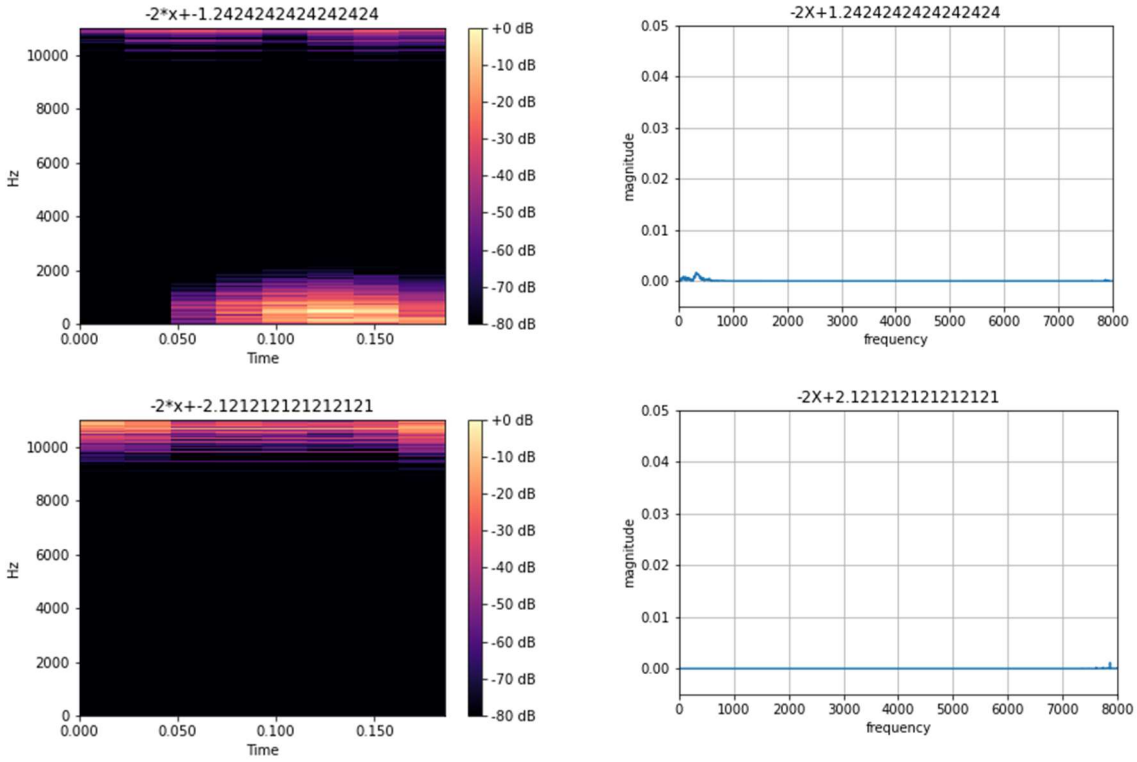



*Figure 16. Spectrogram at n =1.24 and 2.12*

The same observation is seen in the FFT, appreciated in both figures 17 and 18. With the same properties as the first linear function. The spikes located at 500Hz at *n=-1.24* are shifted to almost 8kHz as soon as n=-2.12.

### 4.2.5 STRAIGHT (NO SLOPE) LINEAR FUNCTIONS

The same procedure was used for functions of the form:

$f(x) = G$ where *G* is the gain, which ranged from 0 to 3.

The results showed two different variants.

The functions that lay in the positive area of the y axis output a high frequency hiss sound that does not change its value no matter the gain value selected.

For negative gains they developed the same way as the linear functions with different development areas observed in the spectrogram when compared to the linear. The harmonic structure is kept as the gain is increased as the intensity is also increased. Looking at figure 19, we can see how from one example to another the two small spikes at below 1kHz transform in the two predominant spikes along with several new hues, located at different frequencies as the ones we can observe for example in figure 15.

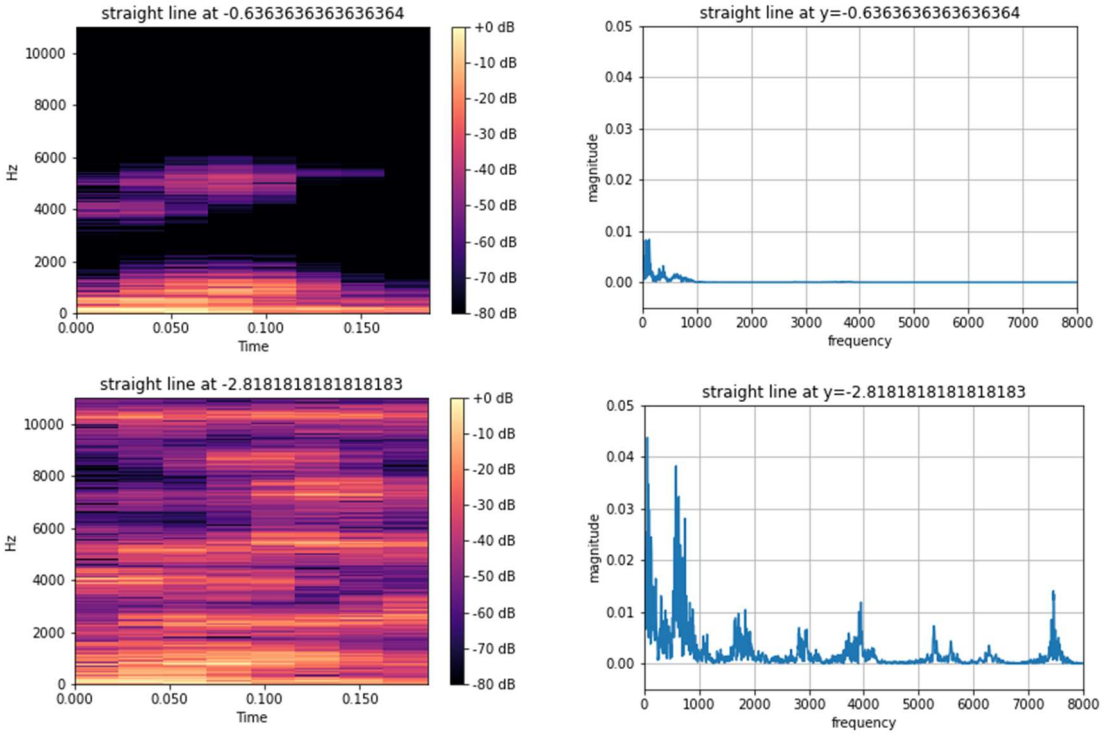



*Figure 17. Spectrogram at f = -0.63 and -2.81*

### 4.2.6 RANDOM GENERATOR

For this case, a random generator NumPy function *np.rand* was used to generate a 256-vector composed by random numbers between 0 and 1. This vector was then offset in the y axis and the spectrograms were studied.

The offset was placed between 0 and -3. This way when the offset was 0, the vector lay completely in the positive side, whilst when it was -3, it lay completely in the negative side. Positive offset values gave no useful sound or spectrograms, just like with positive straight lines.

- Up until an offset of -1.06, the spectrogram is developed in the high frequency with a small high intensity area in the 1kHz range, as seen in figure 20.

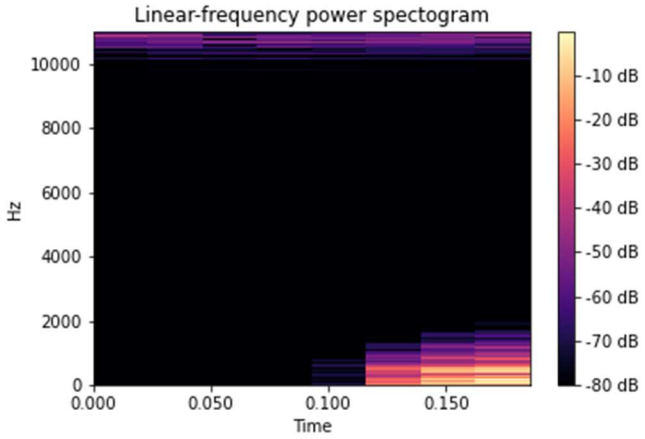



*Figure 18. Spectrogram at offset = -1.06*

- Then, looking at figure 21, it shifts to area development just like the linear functions

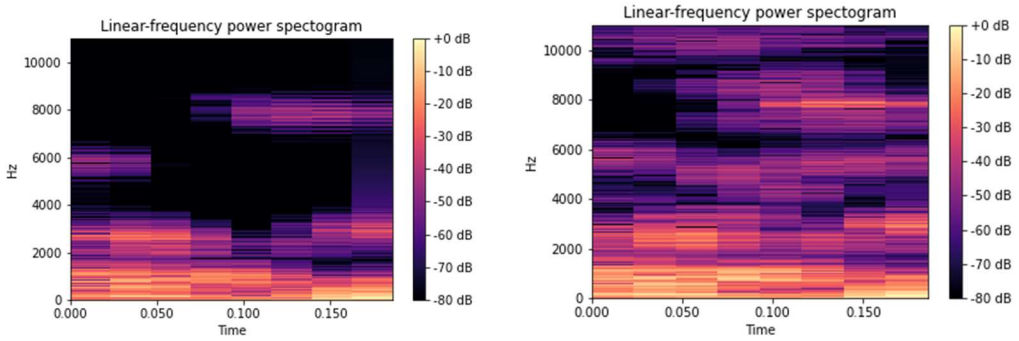



*Figure 19. Spectrogram at offset = -1.63 and -2.56*

The spectrogram was composed in the same way as linear functions, the bigger the offset value the more certain areas in the spectrogram were developed.

### 4.2.7 SINGLE VALUE CHANGE

Finally, one last test was performed by setting a straight line along the 0 and changing one of the vector values to +2 and -2. This delivered different spectrograms that changed when the values were shifted but no significant patterns were observed.

For example, a function with negative point change at 56 would generate a vector that looked like this:

$$[0\ ,\dots,0\ ,-2\ ,0\ ,\dots\ ,0]\ \ for\ [x_1,\dots,x_{55},x_{56},x_{57},\dots,x_{256}]$$

The positive changes delivered sounds that varied very slightly between each other, as well as the negative values. None of the spectrograms seemed to show higher frequencies than 2kHz, but they differed in the duration. There was no proportional correlation between the

position of the changed value and the duration of the sound, but it surely was the only function that place silences in the spectrograms, as seen in figures 21 and 22.

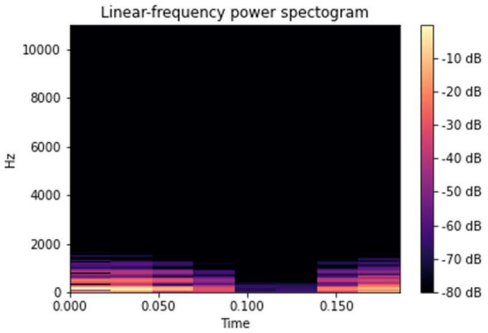



*Figure 21. point change at coordinate 56.*

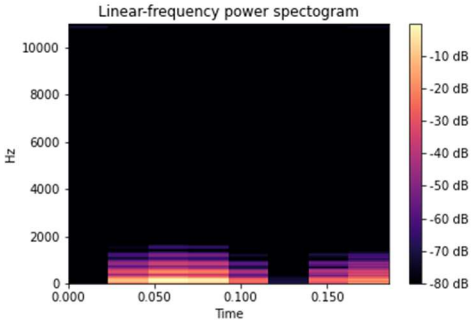



*Figure 20. point change at coordinate 98*

Throughout the whole section hues are mentioned as slight characteristics of a sound that become apparent at certain periods of determined parameters of a function. These are related to the areas of development that can be seen in a spectrogram.

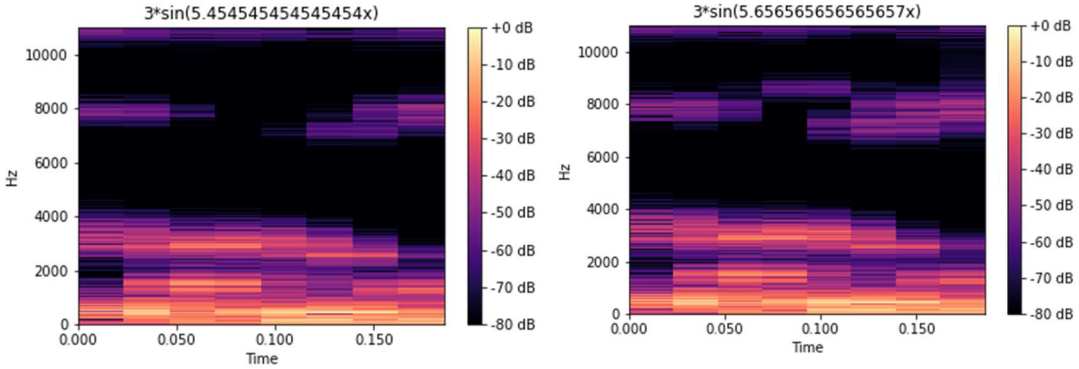



*Figure 22. Sine wave spectrograms for frequencies 5.45 and 5.65 in amplitude 3.*

Comparing the spectrograms of figure 24 of two functions with very similar parameters, we can see that not only the present hues at 8kHz and 3kHz are more developed in intensity between one another, but new hues are created as new areas are created inside the

spectrogram. Now these hues are completely different for the negative equivalent of the function used, as seen in figure 25.

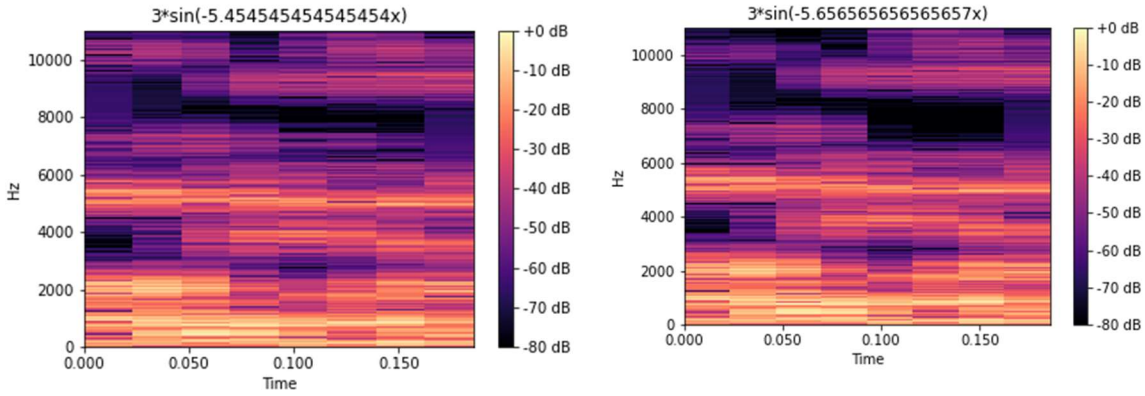



*Figure 23. Sine wave spectrograms for frequencies -5.45 and -5.65 in amplitude 3.*

If we fix our attention to the area around 10kHz, the transformation between example 1 and example 2 is simple a decrease in intensity, inversely to the positive frequency in figure 24.

However, the area at 8kHz appreciated in figure 24 is no longer present in figure 25, even though the parameters are the same but changed in sign. This can also be checked in appendix E through the FFT graphs by looking at the harmonic structure, which is the same for both functions in figure 24, but it is different to the functions in figure 25.

# 5 CONCLUSION

To date, very few significant rules have been discovered. However, it has been found that useful sound is only generated from a certain gain value and vectors must have negative values to generate sound. Otherwise, they only output a high-pitched sound with no significance whatsoever.

For very small parameter changes, the harmonic structure, closely related to the timbre, is also kept for almost all the range of parameters for the same type of function. So, the structure for the sine waves at changes of frequency of less than 0.5 is kept and the only applicable change is seen in the intensity of the frequency spikes, which means it might be related to sound intensity or harmonic richness. However, this harmonic structure changed from one function to another. Furthermore, the concept of hues was introduced, although it was only perceivable and describable when listening to the reconstructed audio. Hues are introduced in small parameter frames, and they correspond to the spectrogram areas that become visible as the parameters are progressively changed. However, no rules were discovered for these as well.

On the other hand, no pitch relations between the vectors and the outputs were found, yet the autoencoder could reconstruct this characteristic, as it seems that this relation is much more complicated than just a simple value change in the vector. When looking at the PCA results we have no strong significant information other than the percussive timbres are less likely to differ in the location of the vector spikes between one sample to another, as appreciated in figure 5 and 6. The relations of the compressed values with the sound seem to be indirect. Much better and sophisticated tests must be carried out in order to arrive at some significant rules and find relevant sound descriptors.

It makes sense to conclude, that the function generation method had some rules since the values are not random between small parameters changes, but they follow the same harmonic structures. So, we can affirm that there is a logical relation, but we were unable to define it.

# 6 LIMITATIONS AND FURTHER IMPROVEMENTS

This research has tended to focus on finding relations between the compressed hidden representations of audio files and their audible characteristics. While no significant relations have been discovered, there is still considerable uncertainty that the approach followed is the right one for the task. For future references, there have been other works where the use of variational autoencoders (VAE) have helped in the search for significant results, such as Roche's (Roche, et al., 2021). A variational autoencoder is defined as an autoencoder with a regularised training phase that prevents overfitting so that a generative process is enabled (Rocca, 2019). In addition to that, different architectures can also be explored with different compression ratios, so better accuracies can be achieved during training, and a better optimization to the specific task is set, since the approach taken in the project is quite a standard one. This also including the use of bigger compressed vectors for less reconstruction loss, especially for the exploration method, which will allow, perhaps, the introduction or clarification of the hues, or even a completely different set of rules than the ones discovered on the path of this work.

Furthermore, there is no way to acoustically analyse the spectrogram areas that are developed with small parameter changes in the exploration functions. By comparing them, we can only affirm that they introduce different hues for different exploration functions, as the areas are developed at different time stamps and frequency bands. But we are unable to know what they sound like, nor the reaction of the ears and the descriptors or adjectives that can be attributed to such hues. An implementation of a sound generating tool or program that acts as a synthesizer, just as seen in Colonel's work, might also help identify characteristics of the function generated sounds. Instead of looping through the spectrograms and FFTs, hearing

the generated sound might show hidden characteristics that are not identifiable by looking at the graphs. (Colonel, et al., 2018). Of course, listening to the batch of reconstructed audio files is the best way to properly describe the changes as we travel along the batch. This will help identify the mentioned hues much better, which is directly related to the main objective of this work: discovering new sound descriptors. The procedure to achieve this, though, requires a much higher knowledge of programming, and therefore would need to be performed and followed by much more experience specialist that would make the idea be useful and intuitive. In fact, the initial aim and title of this paper “discovering new sound descriptors” are very much related to the hues and how they affect the timbre of the output sound.

Of course, this project has been fully developed in a mid-range powered computer, and for that a more powerful computer will not only reduce the training times but improve the overall performance since it will take less time to try other configurations and network types through trial and error. Furthermore, during the programming process of the autoencoder, some research on Keras Tuner was made and tried to get implemented into the project. This module offers an easy way to fine tune the number of layers and compression ratios of the encoder and decoder in order to get maximum performance for a limited set of parameters given by the user. (TensorFlow, n.d.). This was quickly withdrawn since the amount of time to process and optimize was in the order of 20 times higher than the time taken to train the actual autoencoder.

One last improvement, as seen in Roche’s work as well, human input might also be a useful tool when discovering new descriptors (Roche, et al., 2021). Labelling the dataset with a set of characteristics might also open a possibility on finding similarities in the vectors of samples with the similar labels. For example, if a certain batch of samples sound “metallic”, there might be similarities in their compressed representations. These labels can also be transduced to pitch, instrument, and loudness values in order to map these values to the encoded vectors. The mapping of the values would also open a new window in following where these three descriptors are hidden inside the latent representation of each compressed audio file. Now this also involves the use of several different autoencoders and might be tricky to perform without proper formation in these aspects but relating these labels to the hues discovered might create some specified relations between the timbre and the position of the spectrogram developed areas.

# REFERENCES


Alvar M. Kabe, B. H. S., 2020. Chapter 5 - Analysis of continuous and discrete time signals. En: B. H. S. Alvar M. Kabe, ed. *Structural Dynamics Fundamentals and Advanced Applications.* s.l.:Academic Press, pp. 271-427.

Bhattacharyya, S., 2020. *Deep Learning Volume 7.* Berlin/Boston: De Gruyter.

Brownlee, J., 2020. A Gentle Introduction to the Rectified Linear Unit (ReLU). *Machine Learning mastery.*

Chollet, Franccois & others, a., 2015. *Keras.* [Online]
Available at: https://keras.io/
[Last accessed: 10 January 2022].

Colonel, J., Curro, C. & Keene, S., 2018. *AUTOENCODING NEURAL NETWORKS AS MUSICAL AUDIO SYNTHESIZERS.* Aveiro, Portugal, Digital Audio Effects.

Doury, S. & Buttet, C., 2022. Paint with Music. *magenta blog posts.*

Engel, J. y otros, 2017. *Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders.* s.l.:s.n.

Hunter, J., 2007. Matplotlib: A 2D graphics environment. En: *Computing in Science \& Engineering.* s.l.:IEEE COMPUTER SOC, pp. 90-95.

IBM Cloud Education, 2020. *Deep Learning.* [Online]
Available at: https://www.ibm.com/cloud/learn/deep-learning#:~:text=Deep%20learning%20is%20a%20subset,from%20large%20amounts%20of%20data.
[Last accessed: 16 March 2022].

Lee, S., 2018. *'This is the early history of the synthesizer'.* [Online]
Available at: https://www.redbull.com/gb-en/electronic-music-early-history-of-the-synth
[Last accessed: 20 February 2022].

Magenta, s.f. *Make Music and Art Using Machine Learning.* [Online]
Available at: https://magenta.tensorflow.org/
[Last accessed: 12 March 2022].

McFee, y otros, 2015. *librosa: Audio and music signal analysis in Python..* s.l., s.n.

Nair, V. & Hinton, G. H., 2010. Rectified linear units improve restricted boltzmann machines. En: *Proceedings of the 27th international conference on machine learning.* s.l.:ICML-10, pp. 807-814.

numpy.org, 2020. Array programming with Numpy. En: *Nature.* s.l.:Springer Science and Business Media LLC, pp. 357-362.

Oord, A. v. d. y otros, 2016. *WAVENET: A GENERATIVE MODEL FOR RAW AUDIO,* London: s.n.

Pérez, F. & Granger, B. E., 2007. IPython: a System for Interactive Scientific Computing. En: *Computing in Science and Engineering.* s.l.:IEEE Computer Society , pp. 21-29.

Rocca, J., 2019. Understanding Variational Autoencoders (VAEs). *Towards Data Science*, 24 September.

Roche, F. y otros, 2021. Make That Sound More Metallic: Towards a Perceptually Relevant Control of the Timbre of Synthesizer Sounds Using a Variational Autoencoder. En: *Transactions of the International Society for Music Information Retrieval.* s.l.:s.n., pp. 52-66.

Rossum, V., Jr, G. a. D. & L, F., 1995. *Python reference manual.* Amsterdam: Centrum voor Wiskunde en Informatica Amsterdam.

Saha, S., 2018. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. *Towards data science*, 15 December, pp. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

Sartorius, 2020. What Is Principal Component Analysis (PCA) and How It Is Used?. *Science Snipperts Blog*, 18 August.

Somogyi, Z., 2021. *The Application of Artificial Intelligence.* 1st ed. Antwerp, Belgium: Springer, Cham.

TensorFlow.org, 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,* s.l.: s.n.

TensorFlow, s.f. *Introduction to the Keras Tuner.* [Online]
Available at: https://www.tensorflow.org/tutorials/keras/keras_tuner#overview
[Last accessed: 20 April 2022].

Virtanen, P. y otros, 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. En: *Nature Methods.* s.l.:s.n., pp. 261-272.

White, P., 1994. Sound Synthesis: Part 1. *Sound on Sound,* Issue https://www.soundonsound.com/techniques/sound-synthesis-part-1.

# 7 APPENDICES

All Jupyter files and notebooks are available at:

https://github.com/pantartas/IP

## 7.1 APPENDIX A. AUTOENCODER 1 ARCHITECTURE

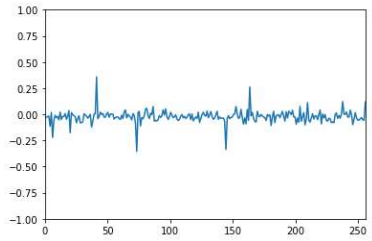
```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
original audio (InputLayer)  [(None, 64, 64, 2)]       0
_________________________________________________________________
batch_normalization (BatchNo (None, 64, 64, 2)         8
_________________________________________________________________
conv2d (Conv2D)              (None, 64, 64, 4)         76
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 4)         148
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 4)         16
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 8)         296
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 8)         584
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
=================================================================
Total params: 1,128
Trainable params: 1,116
Non-trainable params: 12
_________________________________________________________________
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoded audio (InputLayer)   [(None, 2048)]            0
_________________________________________________________________
reshape (Reshape)            (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 16, 16, 8)         584
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 32, 32, 8)         584
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 8)         32
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 32, 32, 4)         292
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 64, 64, 4)         148
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 64, 64, 2)         74
=================================================================
Total params: 1,714
Trainable params: 1,698
Non-trainable params: 16
```

## 7.2 APPENDIX B. AUTOENCODER 2 ARCHITECTURE

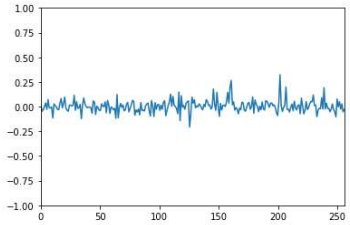
```
Model: "shortener"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoded vector (InputLayer)  [(None, 2048)]            0
_________________________________________________________________
dense_2 (Dense)              (None, 256)               524544
=================================================================
Total params: 524,544
Trainable params: 524,544
Non-trainable params: 0
_________________________________________________________________
Model: "reshaper"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
shortened vector (InputLayer [(None, 256)]             0
_________________________________________________________________
dense_3 (Dense)              (None, 2048)              526336
_________________________________________________________________
dense_4 (Dense)              (None, 2048)              4196352
=================================================================
Total params: 4,722,688
Trainable params: 4,722,688
Non-trainable params: 0
```

## 7.3 APPENDIX C. PCA RESULTS

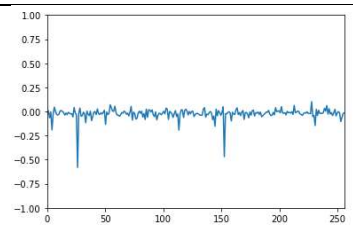
### 7.3.1 Instrument batches



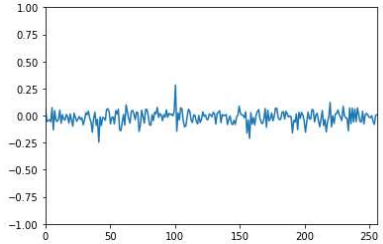
**Electronic bass 18 (59)**
**0.195**



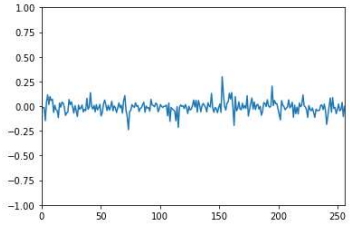
**Electronic bass 25 (53)**
**0.195**



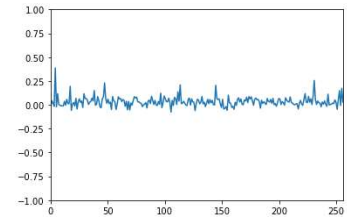
**Synthetic bass 9 (119)**
**0.175**



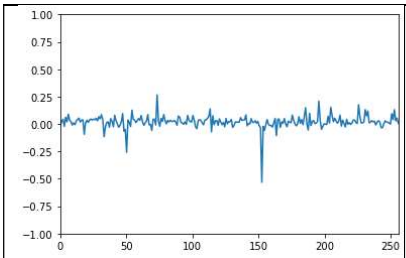
**Synthetic bass 134 (94)**
**0.252**



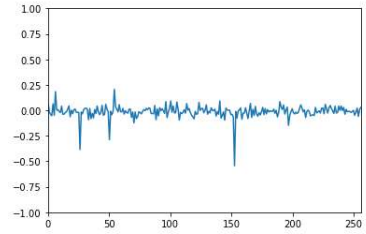
**Brass acoustic 16 (39)**
**0.1885**



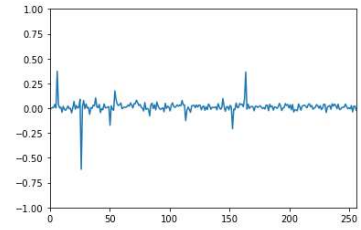
**Brass acoustic 46 (72)**
**0.401**



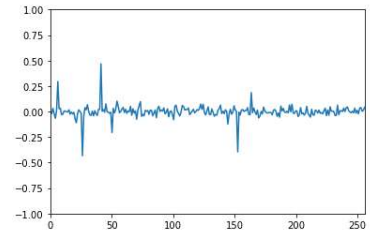
**Flute acoustic 2 (55)**
**0.208**



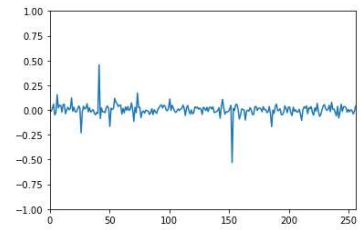
**Flute synthetic 0 (125)**
**0.2481**



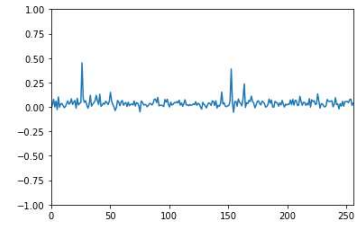
**Guitar acoustic 10 (111)**
**0.1648**



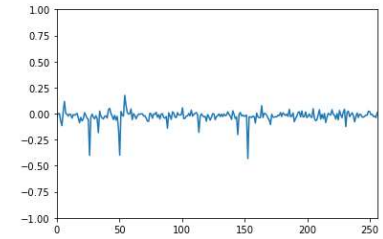
**Guitar electronic 22 (74)**
**0.1788**



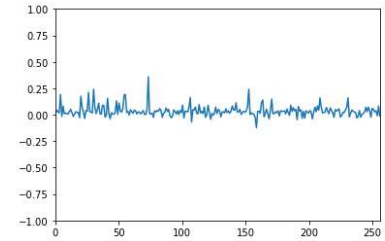
**Keyboard acoustic 4 (119)**
**0.2256**



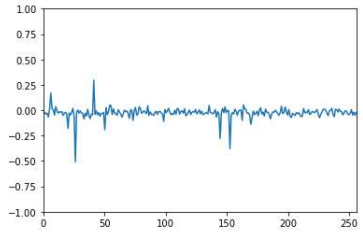
**Keyboard electronic 1 (96)**
**0.1825**



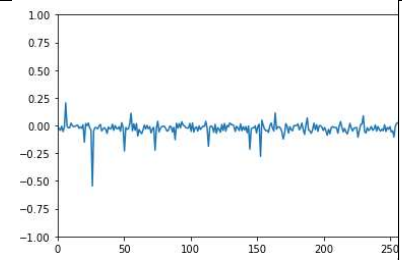
**Keyboard synthetic 0 (108)**
**0.1634**



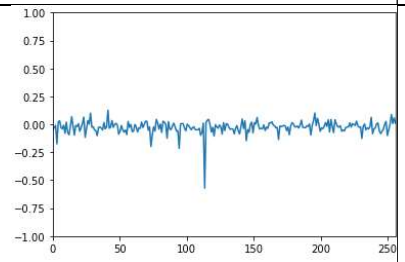
**Mallet acoustic 56 (43)**
**0.2584**



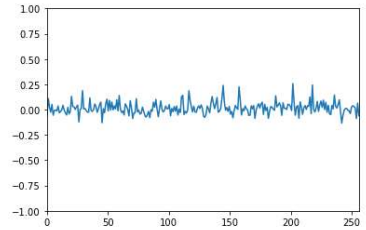
**Mallet acoustic 62 (108)**
**0.203**



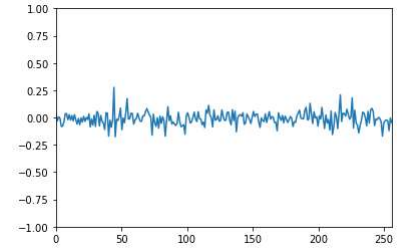
**Organ electronic 1 (72)**
**0.2323**



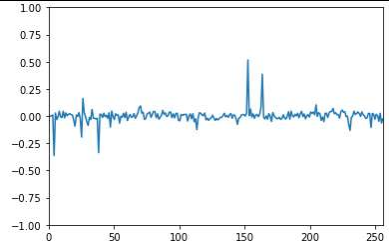
**Reed acoustic 18 (47)**
**0.2067**



**String acoustic (46)**
**0.1708**

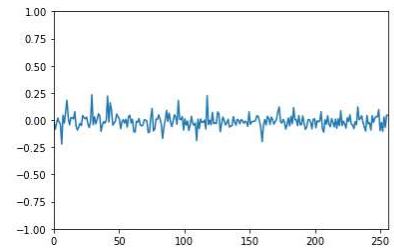


**Vocal acoustic 0 (22)**
**0.2629**

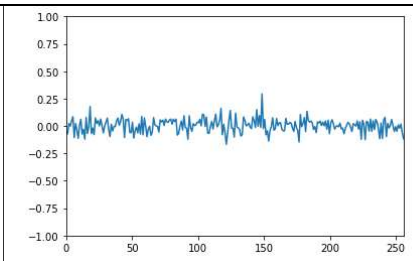


**Vocal synthetic 3 (119)**
**0.1901**

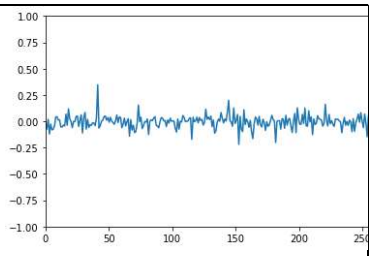
### 7.3.2 Note batches



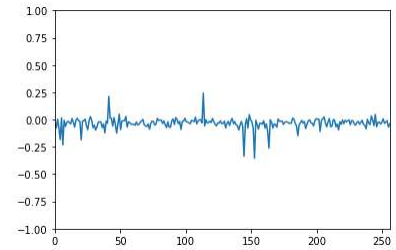
**Note 40 (69)**
**0.205**



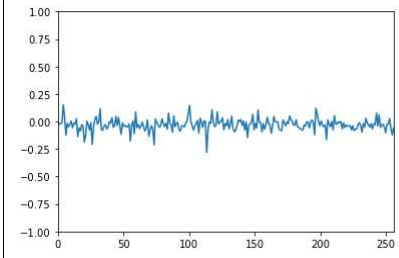
**Note 50 (55)**
**0.206**



**Note 70 (49)**
**0.2450**



**Note 90 (45)**
**0.1936**



**Note 100 (25)**
**0.2969**

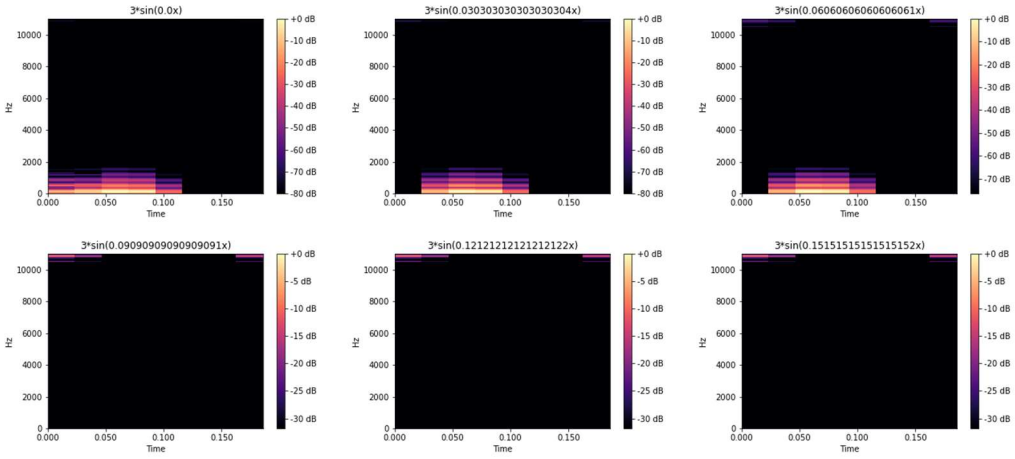
## 7.4 APPENDIX D. FREQUENCY-POWER LINEAR SPECTROGRAMS

Only the most relevant results are included. More results are available at

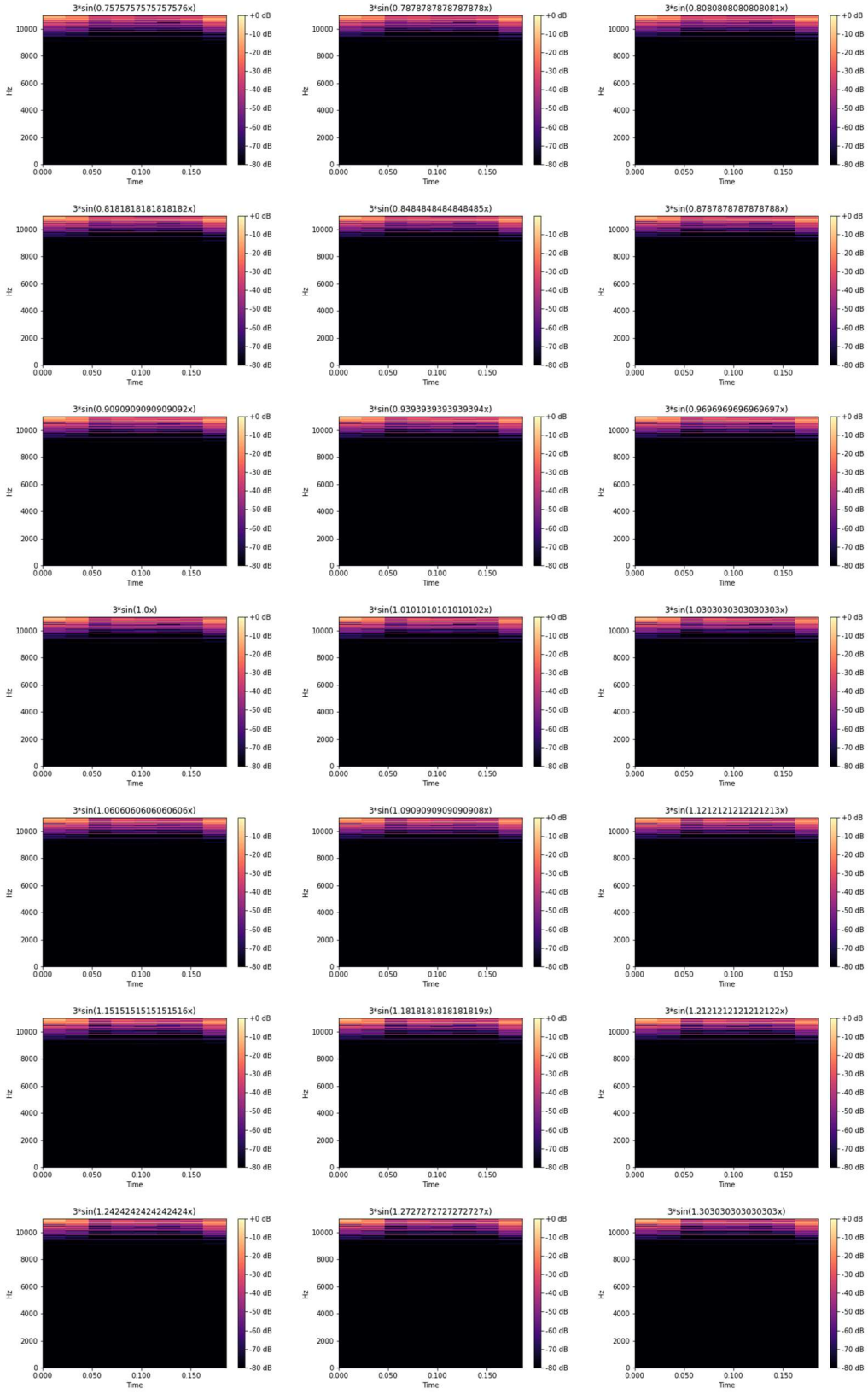
https://sotonac-my.sharepoint.com/:u:/g/personal/evm1u18_soton_ac_uk/EQe3ErnmfdpOqCEOLJPfUw8B3OjrJb-YzEj5dCb7Q8Tqrg?e=jNB4Xz

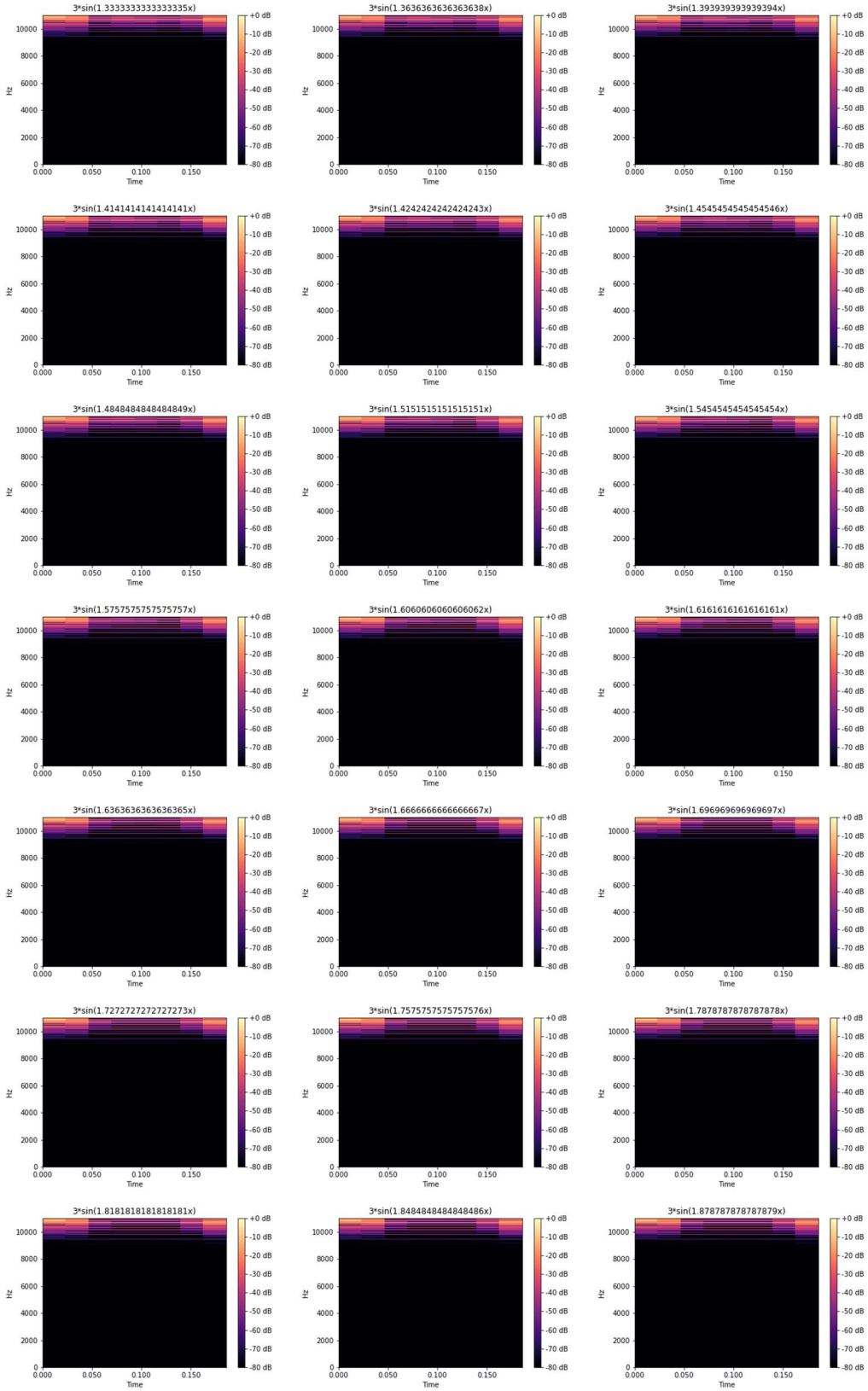
### 7.4.1 Sine/cosine functions

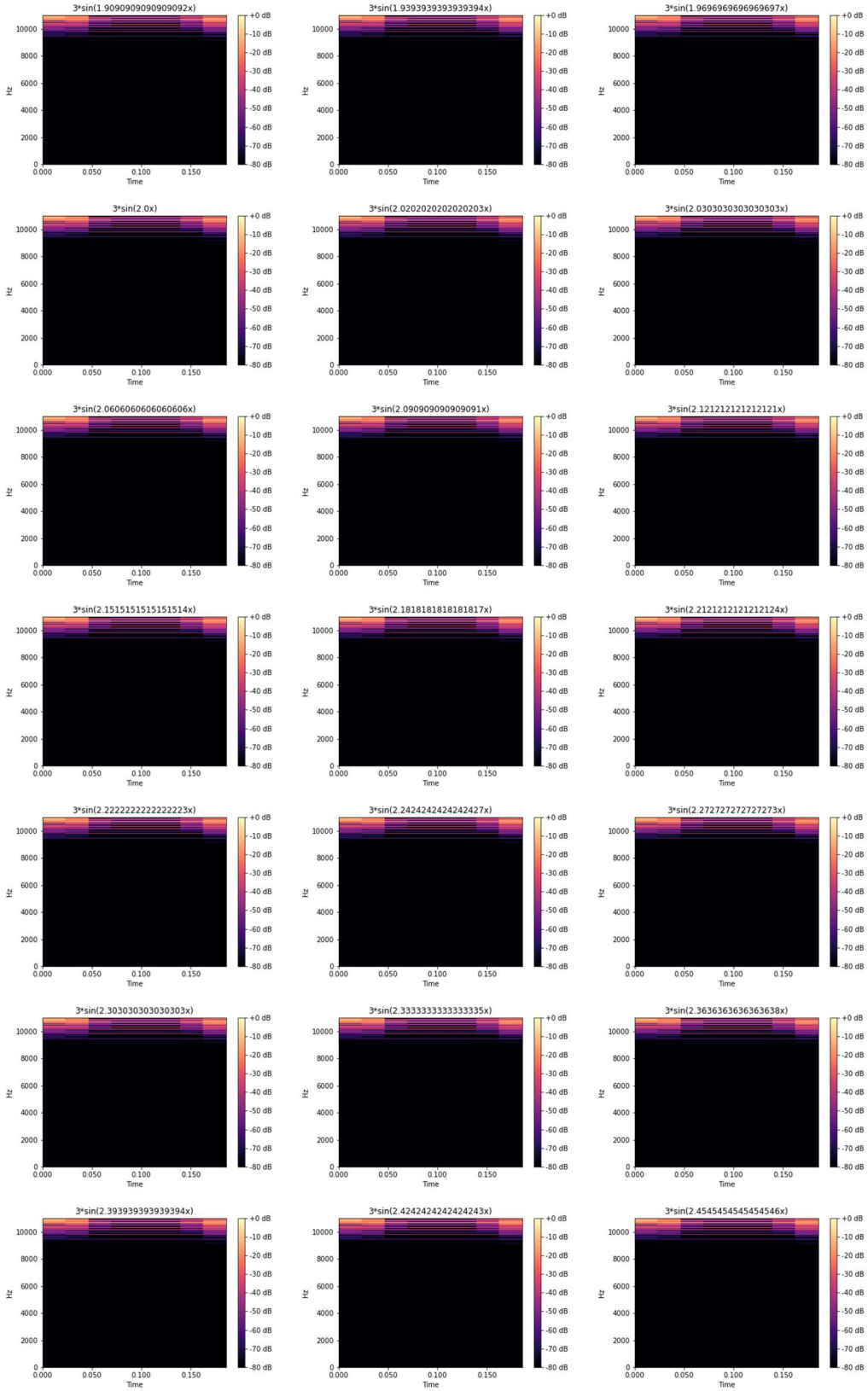
#### *7.4.1.1 F(x) = 3*sin(w*x) for w= [0,20] in steps of 0.3*



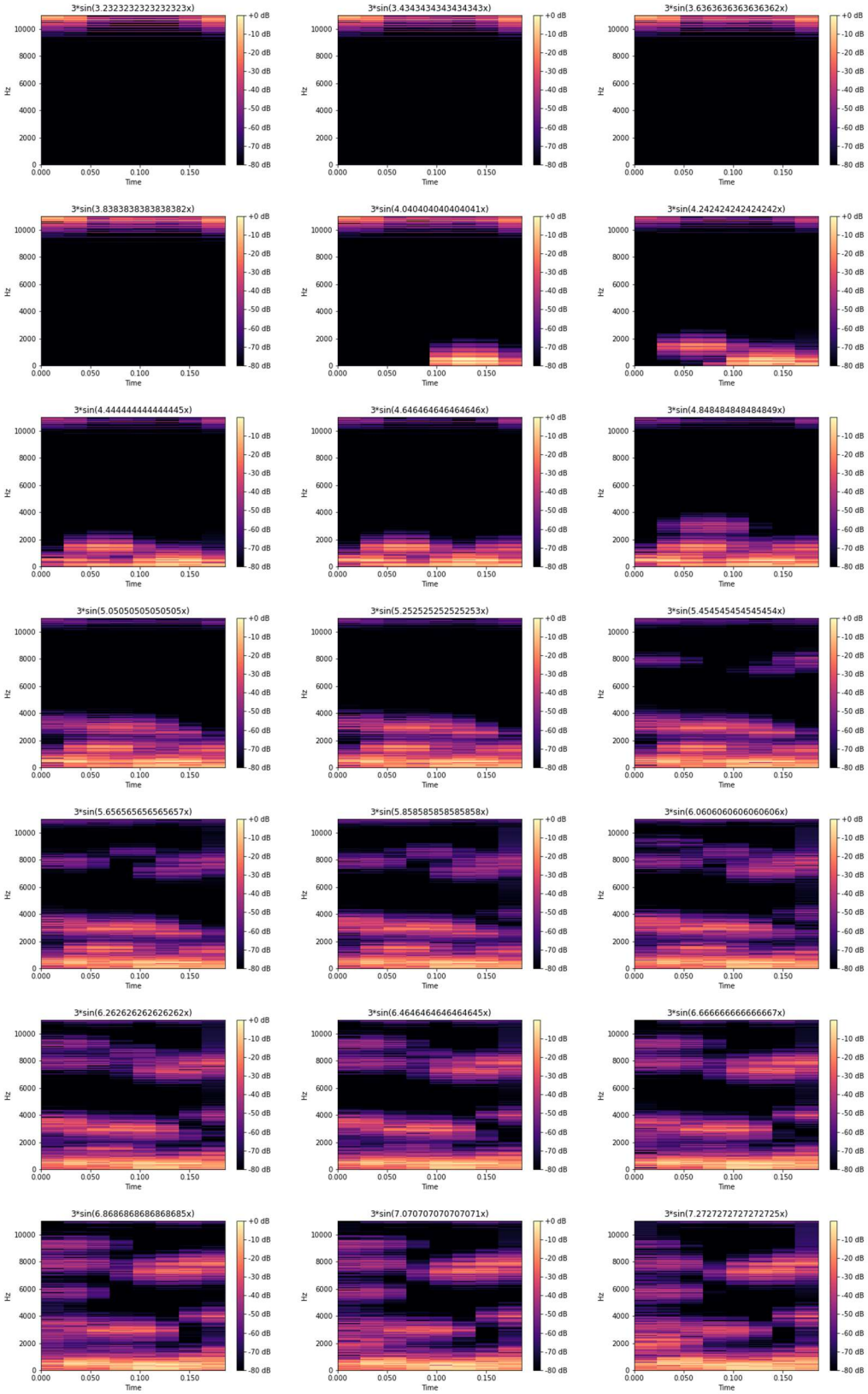

3*sin(0.18181818181818182x)
3*sin(0.20202020202020202x)
3*sin(0.21212121212121213x)
3*sin(0.24242424242424243x)
3*sin(0.2727272727272727x)
3*sin(0.30303030303030304x)
3*sin(0.33333333333333337x)
3*sin(0.3636363636363635x)
3*sin(0.3939393939393939x)
3*sin(0.40404040404040403x)
3*sin(0.42424242424242425x)
3*sin(0.4545454545454546x)
3*sin(0.48484848484848486x)
3*sin(0.5151515151515151x)
3*sin(0.5454545454545454x)
3*sin(0.5757575757575758x)
3*sin(0.6060606060606061x)
3*sin(0.6363636363636364x)
3*sin(0.6666666666666667x)
3*sin(0.696969696969697x)
3*sin(0.7272727272727273x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

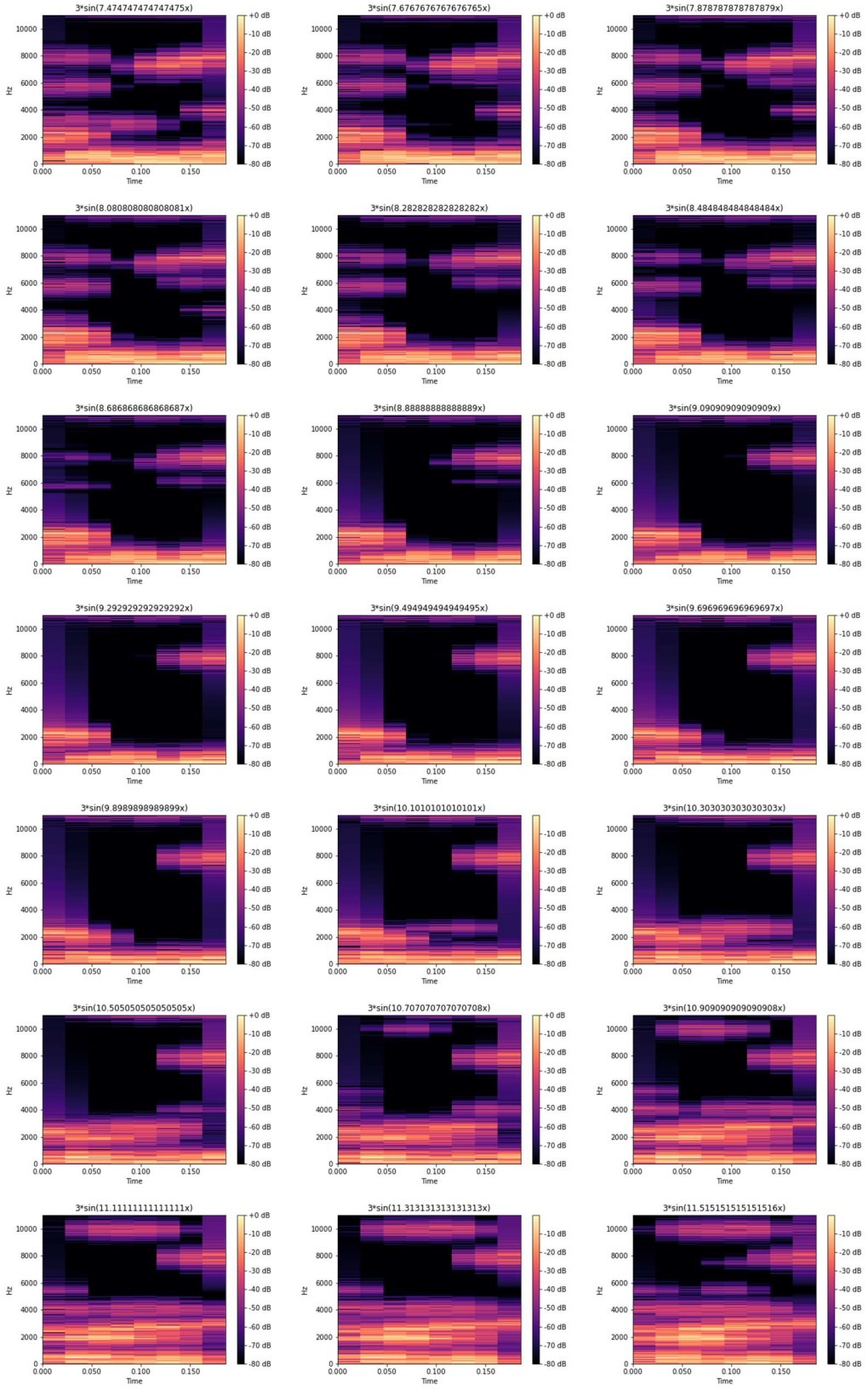

3*sin(0.7575757575757576x)
3*sin(0.7878787878787878x)
3*sin(0.80808080808080 81x)
3*sin(0.8181818181818182x)
3*sin(0.8484848484848485x)
3*sin(0.8787878787878788x)
3*sin(0.9090909090909092x)
3*sin(0.9393939393939394x)
3*sin(0.9696969696969697x)
3*sin(1.0x)
3*sin(1.0101010101010102x)
3*sin(1.0303030303030303x)
3*sin(1.0606060606060606x)
3*sin(1.0909090909090908x)
3*sin(1.1212121212121213x)
3*sin(1.1515151515151516x)
3*sin(1.1818181818181819x)
3*sin(1.2121212121212122x)
3*sin(1.2424242424242424x)
3*sin(1.2727272727272727x)
3*sin(1.303030303030303x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

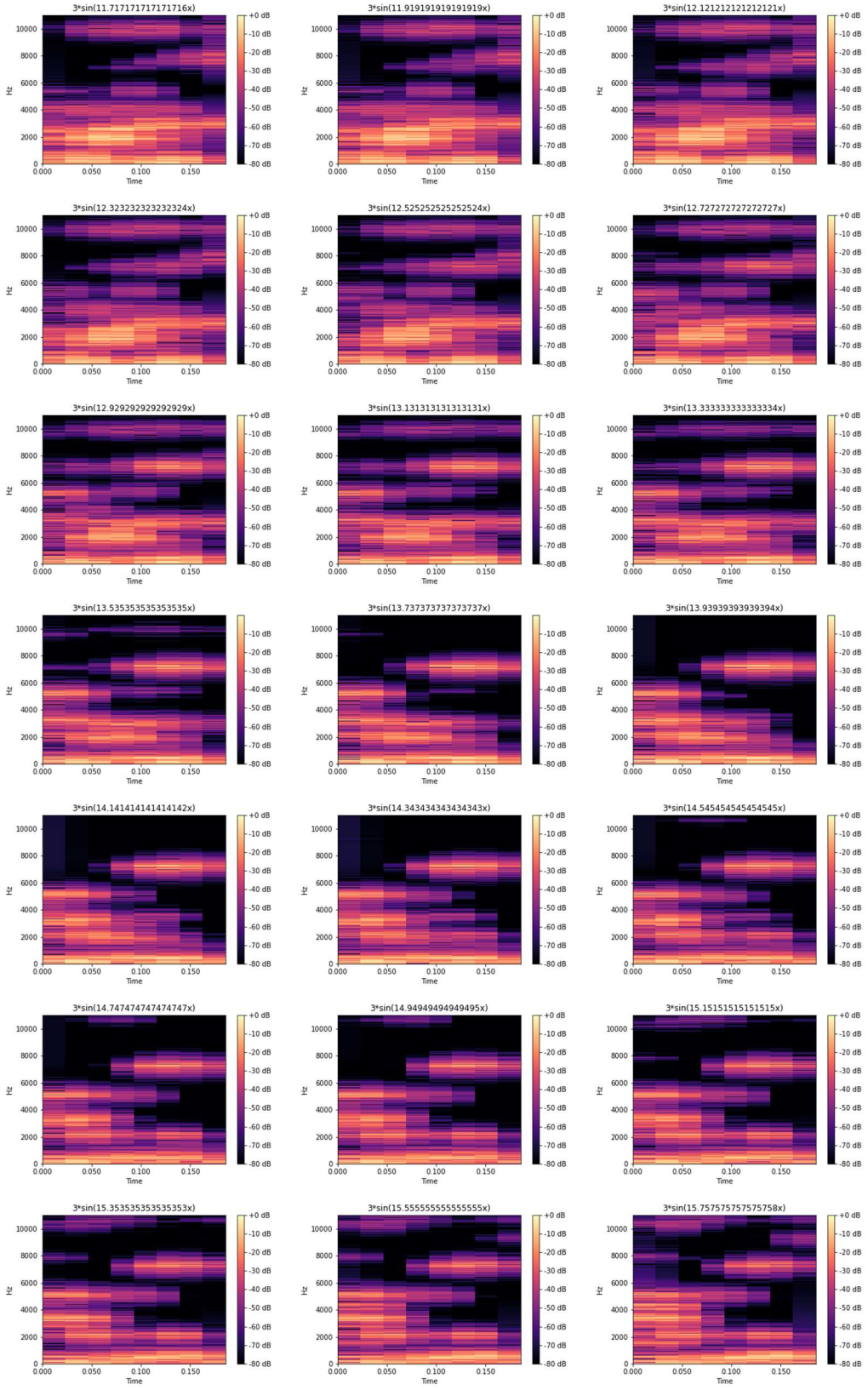

3*sin(1.3333333333333335x)
3*sin(1.3636363636363638x)
3*sin(1.393939393939394x)
3*sin(1.4141414141414141x)
3*sin(1.4242424242424243x)
3*sin(1.4545454545454546x)
3*sin(1.4848484848484849x)
3*sin(1.5151515151515151x)
3*sin(1.5454545454545454x)
3*sin(1.5757575757575757x)
3*sin(1.6060606060606062x)
3*sin(1.6161616161616161x)
3*sin(1.6363636363636365x)
3*sin(1.6666666666666667x)
3*sin(1.696969696969697x)
3*sin(1.7272727272727273x)
3*sin(1.7575757575757576x)
3*sin(1.7878787878787878x)
3*sin(1.8181818181818181x)
3*sin(1.8484848484848486x)
3*sin(1.878787878787879x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

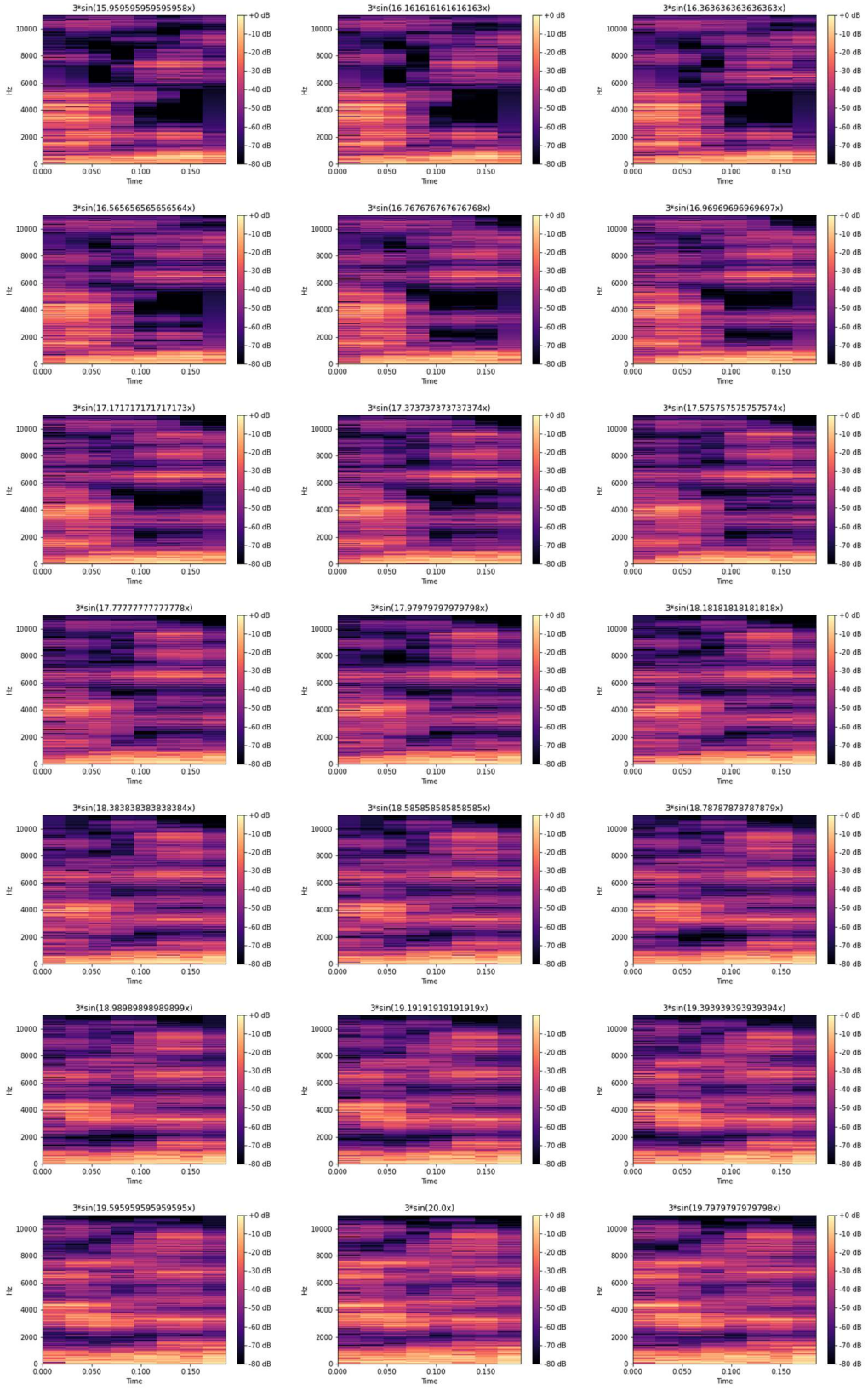

3*sin(1.9090909090909092x)
3*sin(1.9393939393939394x)
3*sin(1.9696969696969697x)
3*sin(2.0x)
3*sin(2.0202020202020203x)
3*sin(2.0303030303030303x)
3*sin(2.0606060606060606x)
3*sin(2.090909090909091x)
3*sin(2.121212121212121x)
3*sin(2.1515151515151514x)
3*sin(2.1818181818181817x)
3*sin(2.2121212121212124x)
3*sin(2.2222222222222223x)
3*sin(2.2424242424242427x)
3*sin(2.272727272727273x)
3*sin(2.303030303030303x)
3*sin(2.3333333333333335x)
3*sin(2.3636363636363638x)
3*sin(2.393939393939394x)
3*sin(2.4242424242424243x)
3*sin(2.4545454545454546x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


3*sin(2.4848484848484485x)
3*sin(2.515151515151515x)
3*sin(2.5454545454545454x)
3*sin(2.5757575757575757x)
3*sin(2.606060606060606x)
3*sin(2.6262626262626263x)
3*sin(2.6363636363636362x)
3*sin(2.666666666666667x)
3*sin(2.6969696969696972x)
3*sin(2.7272727272727275x)
3*sin(2.75757575757575758x)
3*sin(2.787878787878788x)
3*sin(2.8181818181818183x)
3*sin(2.8282828282828283x)
3*sin(2.8484848484848486x)
3*sin(2.878787878787879x)
3*sin(2.909090909090909x)
3*sin(2.9393939393939394x)
3*sin(2.9696969696969697x)
3*sin(3.0x)
3*sin(3.0303030303030303x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

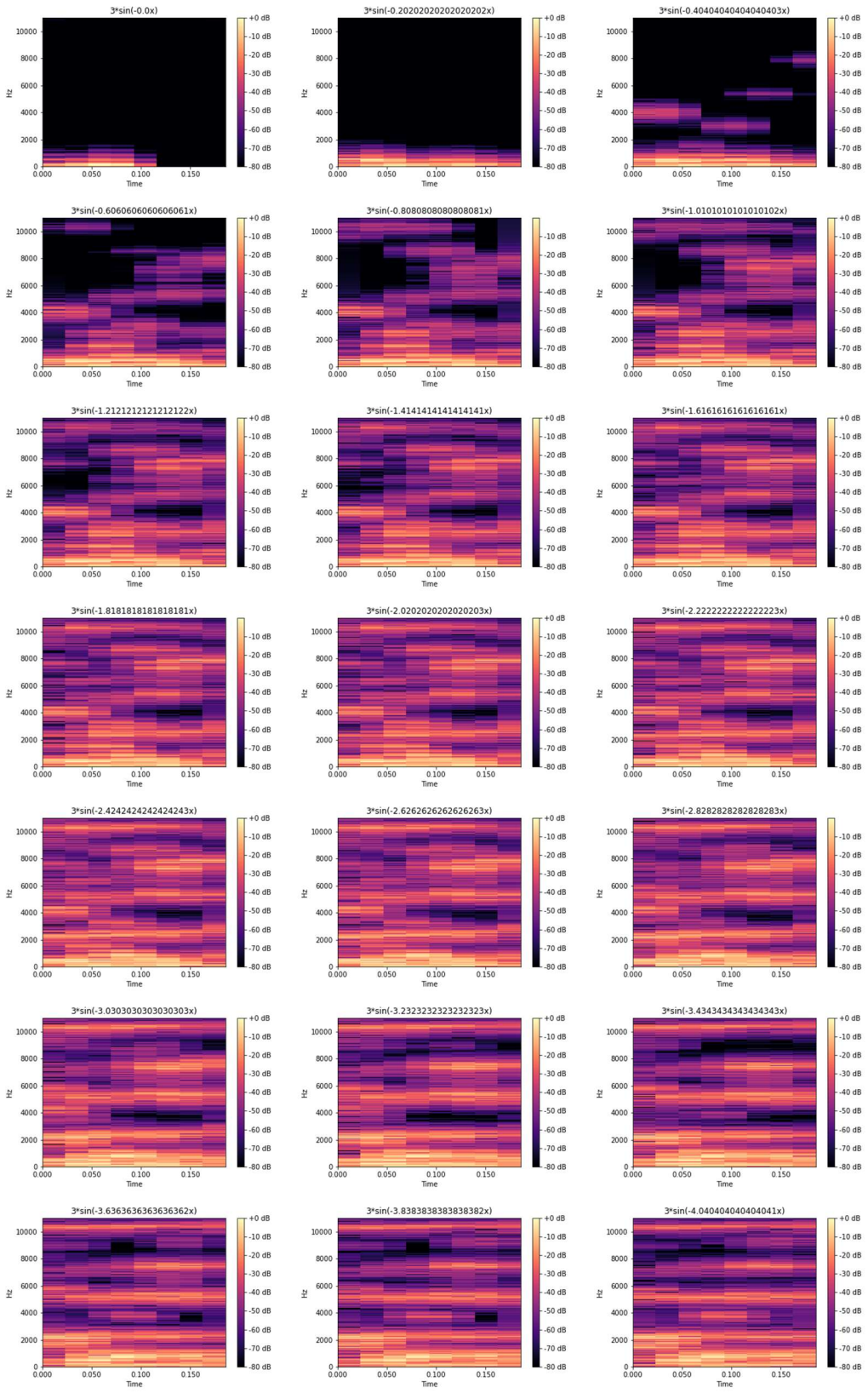

3*sin(3.2323232323232323x)
3*sin(3.4343434343434343x)
3*sin(3.6363636363636362x)
3*sin(3.8383838383838382x)
3*sin(4.040404040404041x)
3*sin(4.2424242424242424x)
3*sin(4.444444444444445x)
3*sin(4.646464646464646x)
3*sin(4.848484848484849x)
3*sin(5.05050505050505x)
3*sin(5.252525252525253x)
3*sin(5.454545454545454x)
3*sin(5.656565656565657x)
3*sin(5.858585858585858x)
3*sin(6.0606060606060606x)
3*sin(6.262626262626262x)
3*sin(6.4646464646464645x)
3*sin(6.666666666666667x)
3*sin(6.8686868686868685x)
3*sin(7.070707070707071x)
3*sin(7.2727272727272725x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

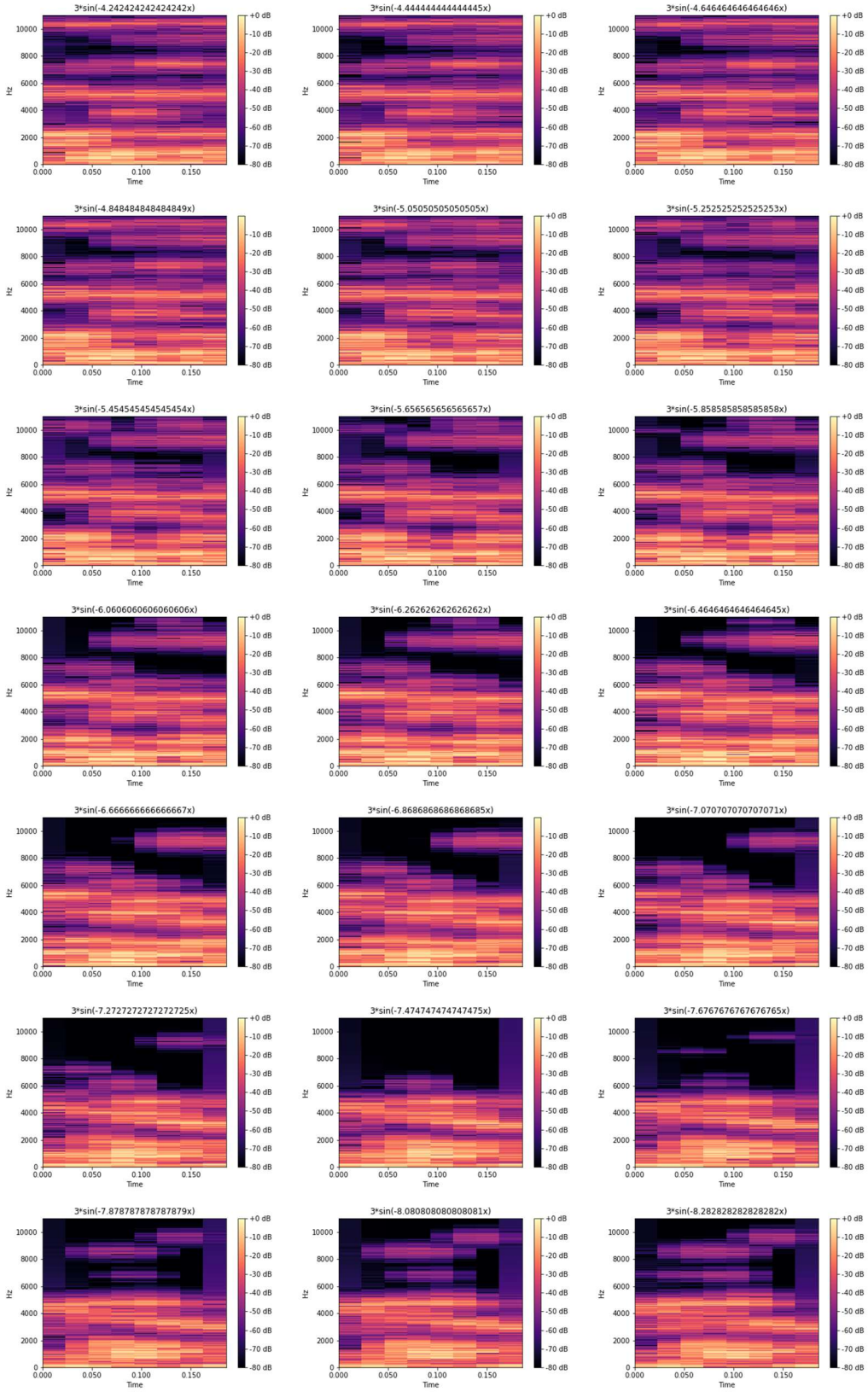

3*sin(7.474747474747475x)
3*sin(7.6767676767676765x)
3*sin(7.878787878787879x)
3*sin(8.080808080808081x)
3*sin(8.282828282828282x)
3*sin(8.484848484848484x)
3*sin(8.686868686868687x)
3*sin(8.88888888888889x)
3*sin(9.09090909090909x)
3*sin(9.292929292929292x)
3*sin(9.494949494949495x)
3*sin(9.696969696969697x)
3*sin(9.8989898989899x)
3*sin(10.1010101010101x)
3*sin(10.303030303030303x)
3*sin(10.505050505050505x)
3*sin(10.707070707070708x)
3*sin(10.909090909090908x)
3*sin(11.111111111111111x)
3*sin(11.313131313131313x)
3*sin(11.515151515151516x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

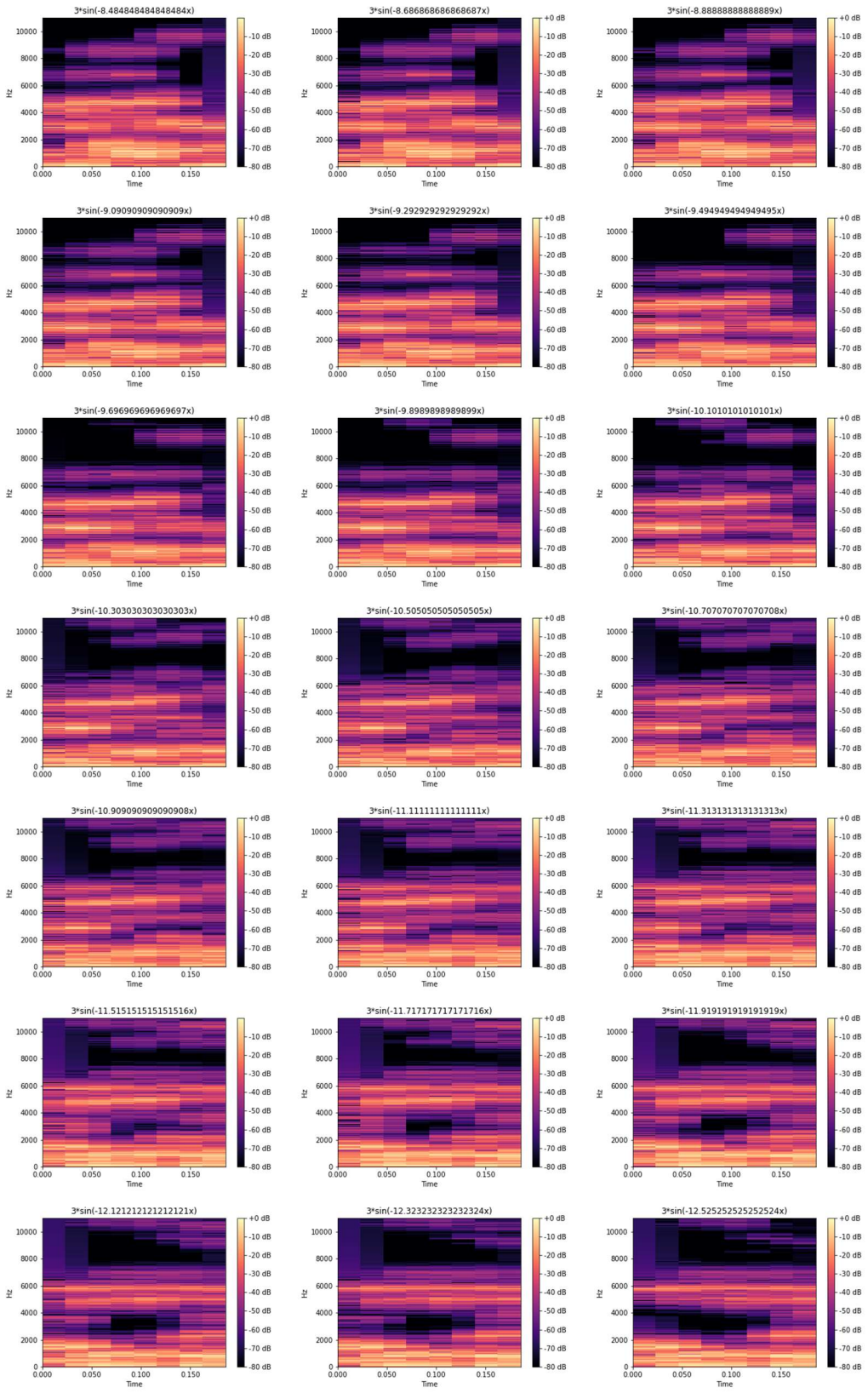

3*sin(11.717171717171716x)
3*sin(11.919191919191919x)
3*sin(12.121212121212121x)
3*sin(12.323232323232324x)
3*sin(12.525252525252524x)
3*sin(12.727272727272727x)
3*sin(12.929292929292929x)
3*sin(13.131313131313131x)
3*sin(13.333333333333334x)
3*sin(13.535353535353535x)
3*sin(13.737373737373737x)
3*sin(13.939393939393940x)
3*sin(14.141414141414142x)
3*sin(14.343434343434343x)
3*sin(14.545454545454545x)
3*sin(14.747474747474747x)
3*sin(14.949494949494950x)
3*sin(15.151515151515151x)
3*sin(15.353535353535353x)
3*sin(15.555555555555555x)
3*sin(15.757575757575758x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

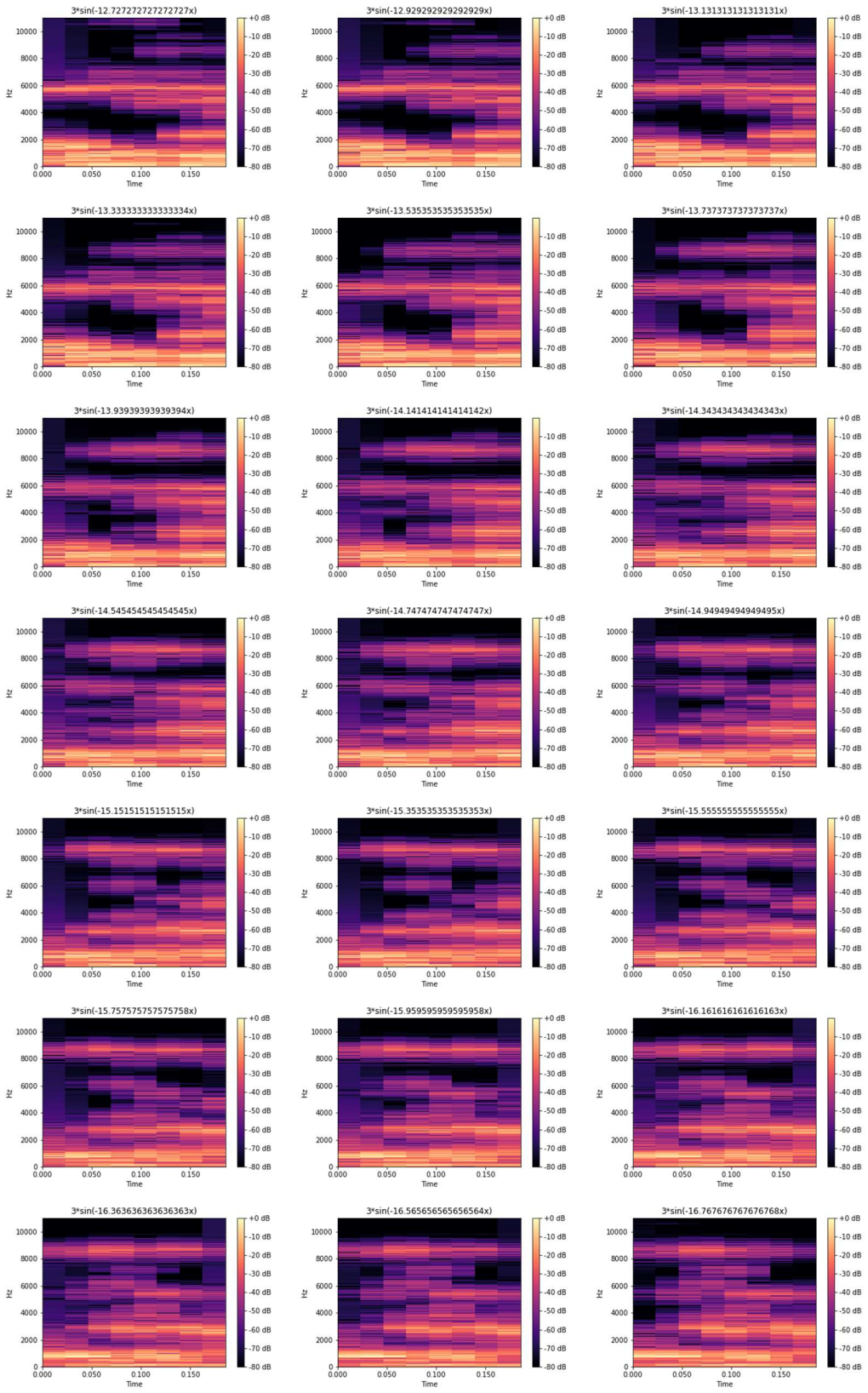


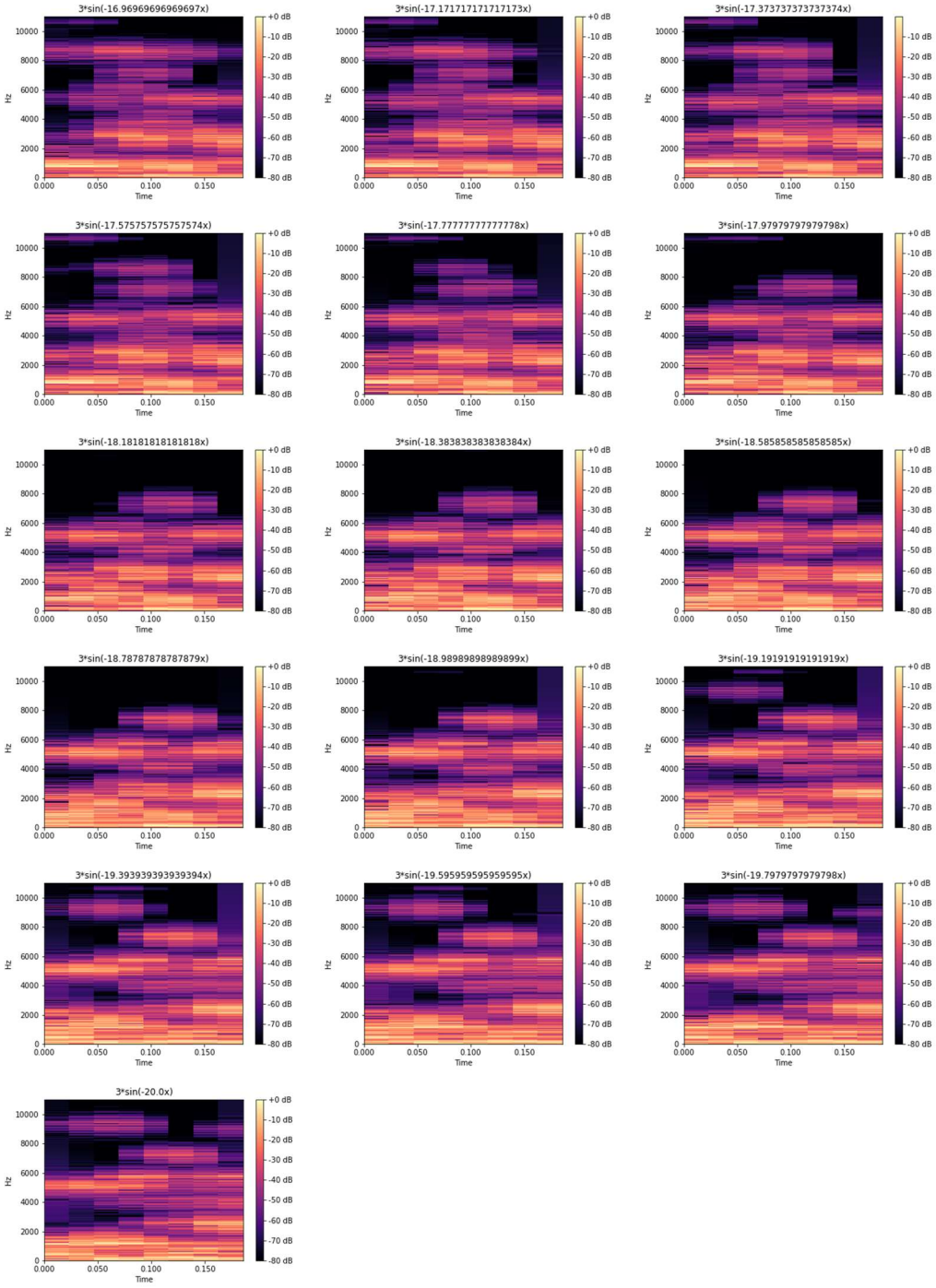

### *7.4.1.2 F(x) = -3*sin(w*x) for w = [0,20] in steps of 0.3*


3*sin(-0.0x)
3*sin(-0.20202020202020202x)
3*sin(-0.40404040404040403x)
3*sin(-0.6060606060606061x)
3*sin(-0.8080808080808081x)
3*sin(-1.0101010101010102x)
3*sin(-1.2121212121212122x)
3*sin(-1.4141414141414141x)
3*sin(-1.6161616161616161x)
3*sin(-1.8181818181818181x)
3*sin(-2.0202020202020203x)
3*sin(-2.2222222222222223x)
3*sin(-2.4242424242424243x)
3*sin(-2.6262626262626263x)
3*sin(-2.8282828282828283x)
3*sin(-3.0303030303030303x)
3*sin(-3.2323232323232323x)
3*sin(-3.4343434343434343x)
3*sin(-3.6363636363636362x)
3*sin(-3.8383838383838382x)
3*sin(-4.040404040404041x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


3*sin(-4.242424242424242x)
3*sin(-4.444444444444445x)
3*sin(-4.646464646464646x)
3*sin(-4.848484848484849x)
3*sin(-5.05050505050505x)
3*sin(-5.252525252525253x)
3*sin(-5.454545454545454x)
3*sin(-5.656565656565657x)
3*sin(-5.858585858585858x)
3*sin(-6.0606060606060606x)
3*sin(-6.262626262626262x)
3*sin(-6.4646464646464645x)
3*sin(-6.666666666666667x)
3*sin(-6.8686868686868685x)
3*sin(-7.070707070707071x)
3*sin(-7.2727272727272725x)
3*sin(-7.474747474747475x)
3*sin(-7.6767676767676765x)
3*sin(-7.878787878787879x)
3*sin(-8.080808080808081x)
3*sin(-8.282828282828282x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

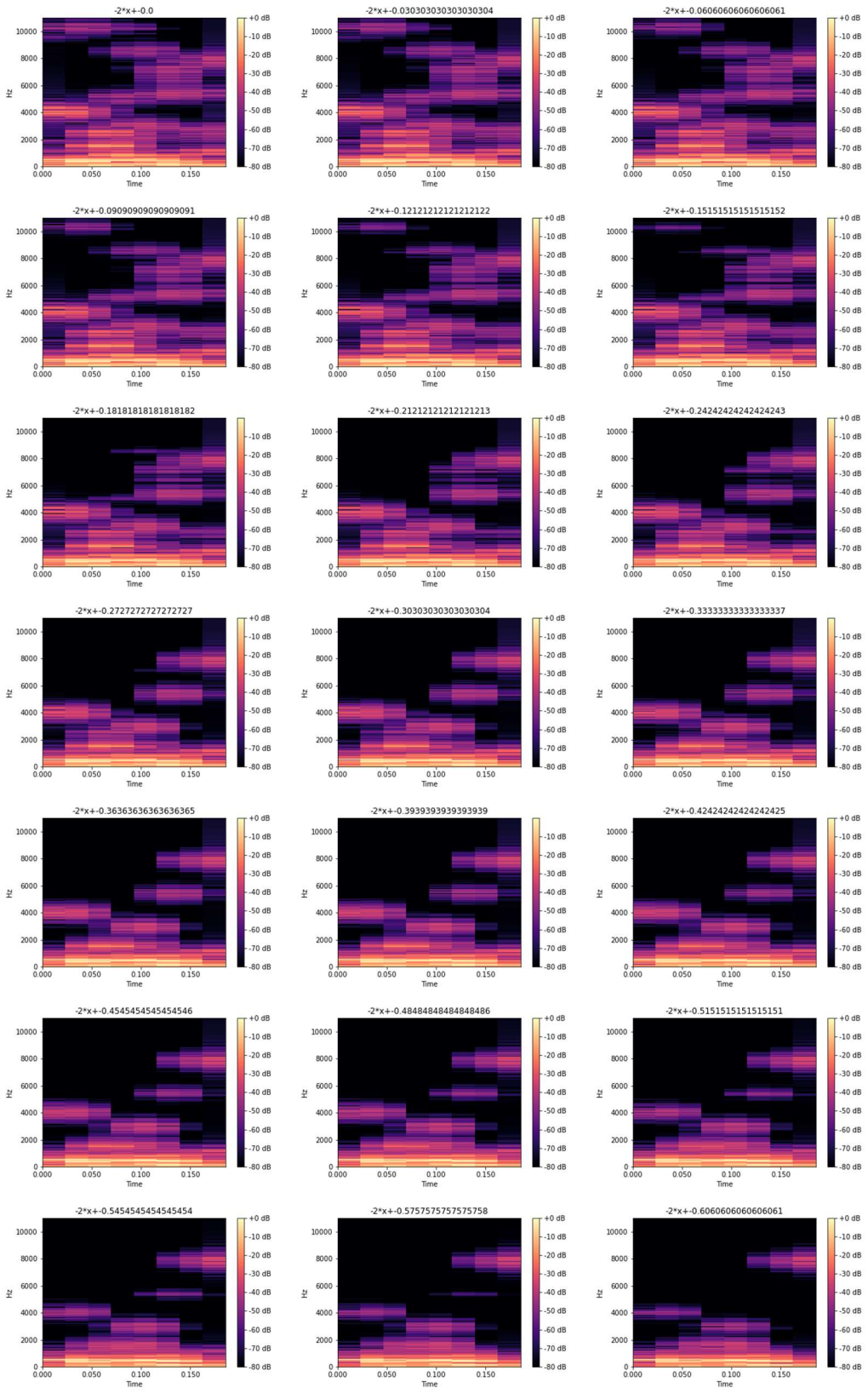

3*sin(-8.4848484848484x)
3*sin(-8.6868686868687x)
3*sin(-8.888888888888889x)
3*sin(-9.090909090909x)
3*sin(-9.292929292929292x)
3*sin(-9.494949494949495x)
3*sin(-9.696969696969697x)
3*sin(-9.8989898989899x)
3*sin(-10.1010101010101x)
3*sin(-10.303030303030303x)
3*sin(-10.505050505050505x)
3*sin(-10.707070707070708x)
3*sin(-10.909090909090908x)
3*sin(-11.111111111111111x)
3*sin(-11.313131313131313x)
3*sin(-11.515151515151516x)
3*sin(-11.717171717171716x)
3*sin(-11.919191919191919x)
3*sin(-12.121212121212121x)
3*sin(-12.323232323232324x)
3*sin(-12.525252525252524x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

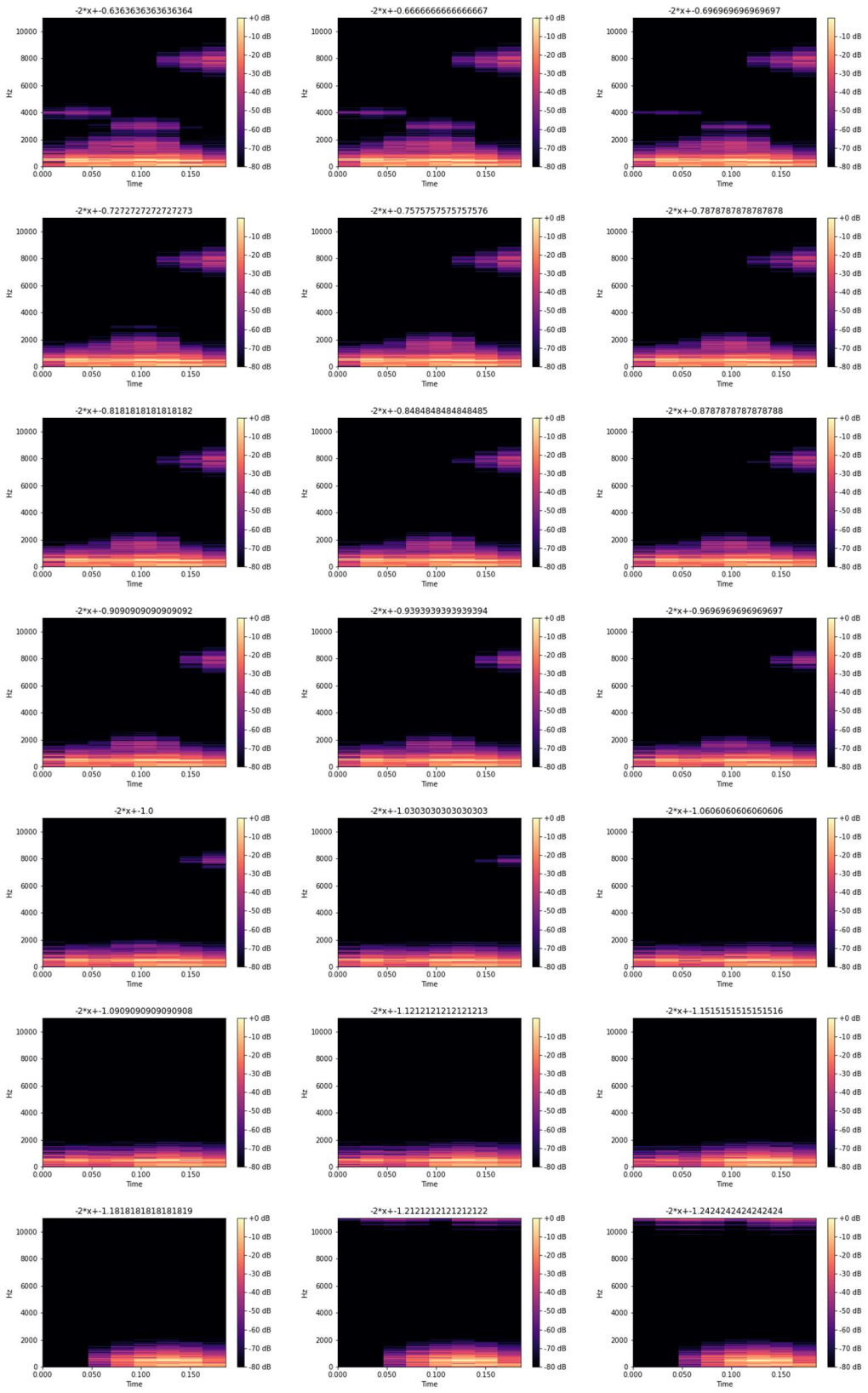

3*sin(-12.727272727272727x)
3*sin(-12.929292929292929x)
3*sin(-13.1313131313131x)
3*sin(-13.333333333333334x)
3*sin(-13.535353535353535x)
3*sin(-13.737373737373737x)
3*sin(-13.93939393939394x)
3*sin(-14.141414141414142x)
3*sin(-14.343434343434343x)
3*sin(-14.545454545454545x)
3*sin(-14.747474747474747x)
3*sin(-14.94949494949495x)
3*sin(-15.151515151515151x)
3*sin(-15.353535353535353x)
3*sin(-15.5555555555555555x)
3*sin(-15.757575757575758x)
3*sin(-15.959595959595958x)
3*sin(-16.161616161616163x)
3*sin(-16.363636363636363x)
3*sin(-16.565656565656564x)
3*sin(-16.767676767676768x)
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

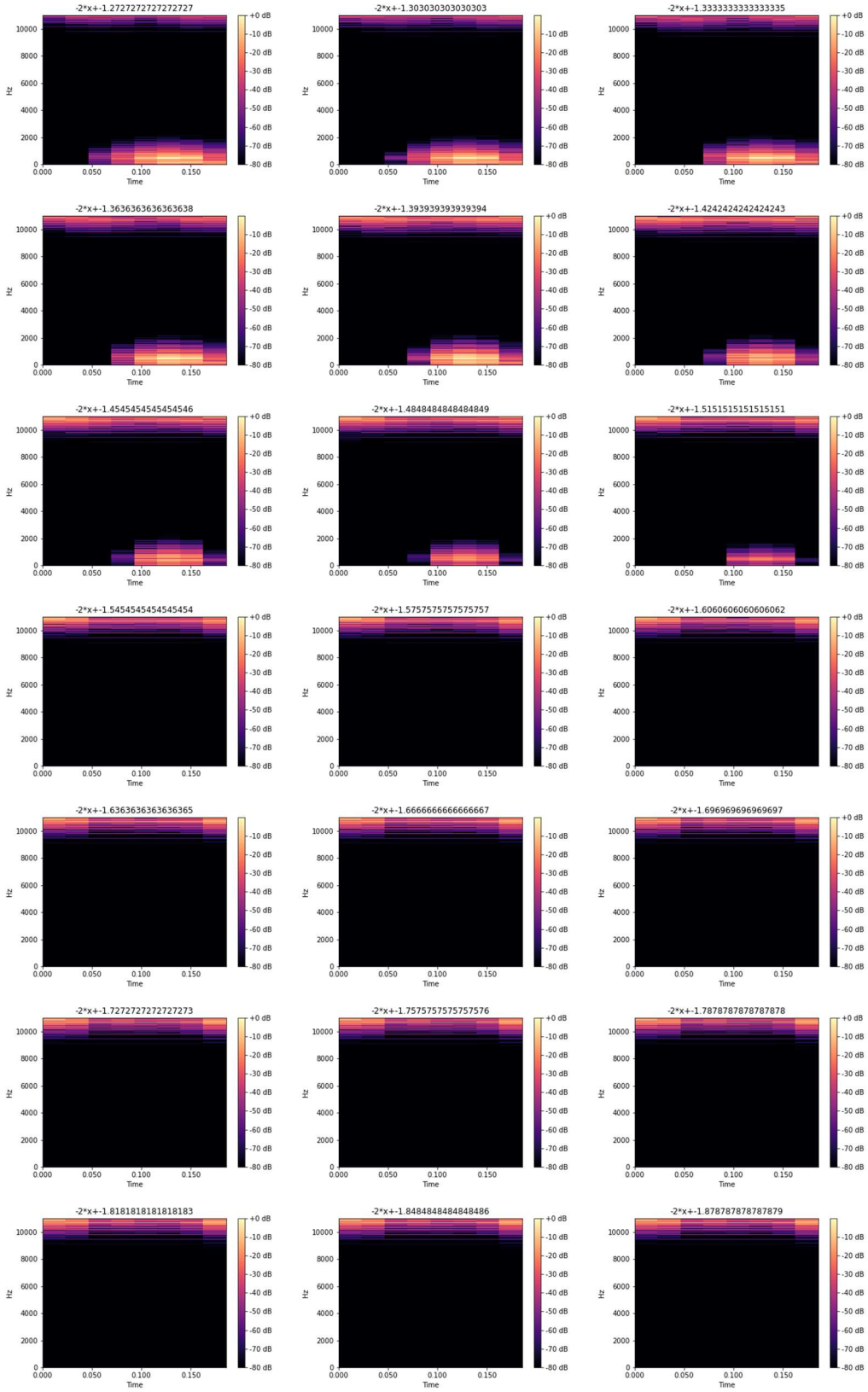



## 7.4.2 Linear functions

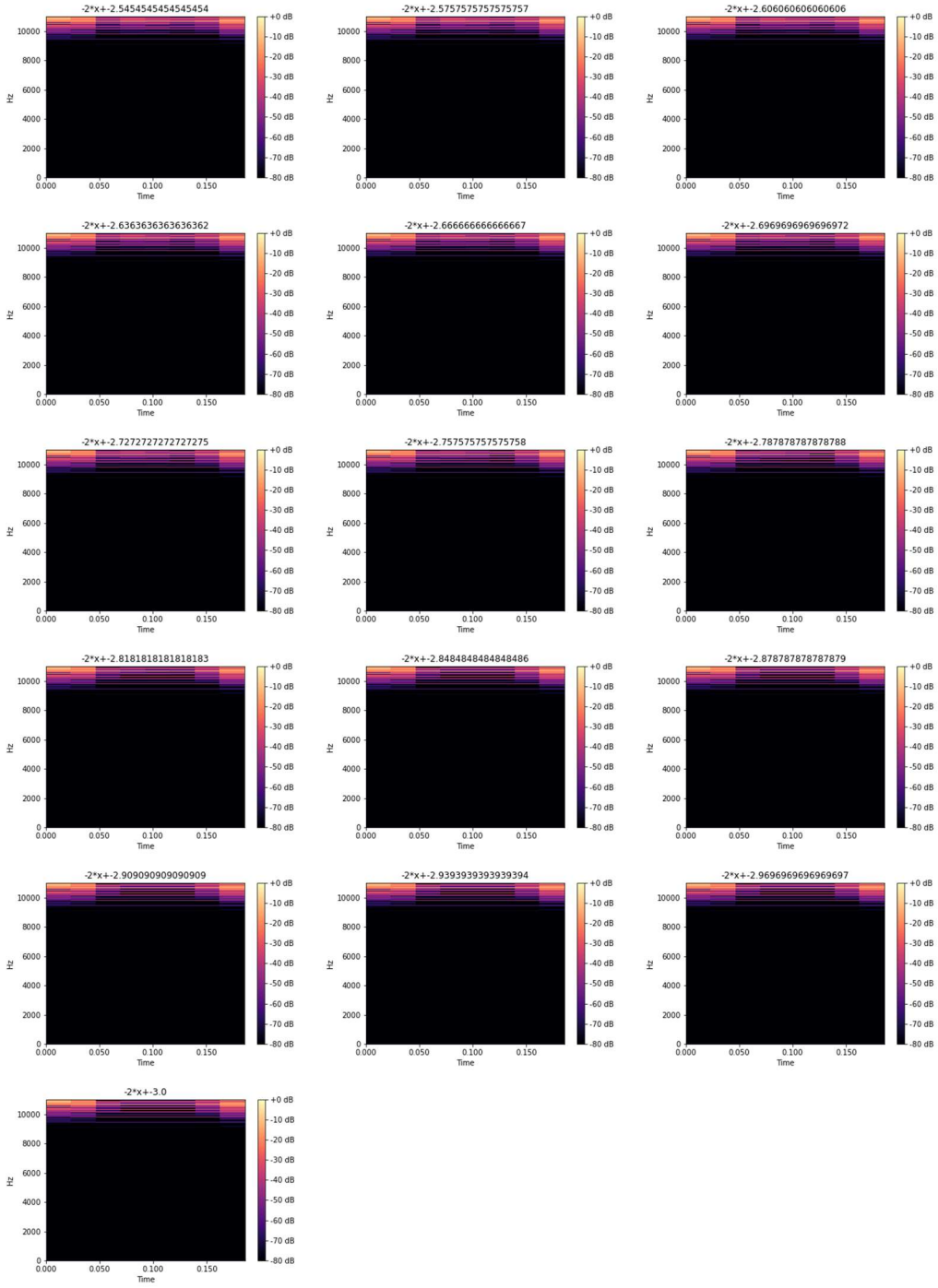
### *7.4.2.1 F(x) = -2x + m for m = [0,3] in steps of 0.03*


-2*x+-0.0
-2*x+-0.030303030303030304
-2*x+-0.06060606060606061
-2*x+-0.09090909090909091
-2*x+-0.12121212121212122
-2*x+-0.15151515151515152
-2*x+-0.18181818181818182
-2*x+-0.21212121212121213
-2*x+-0.24242424242424243
-2*x+-0.2727272727272727
-2*x+-0.30303030303030304
-2*x+-0.33333333333333337
-2*x+-0.36363636363636365
-2*x+-0.3939393939393939
-2*x+-0.42424242424242425
-2*x+-0.4545454545454546
-2*x+-0.48484848484848486
-2*x+-0.5151515151515151
-2*x+-0.5454545454545454
-2*x+-0.5757575757575758
-2*x+-0.6060606060606061
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


-2*x+-0.6363636363636364
-2*x+-0.6666666666666667
-2*x+-0.6969696969696697
-2*x+-0.7272727272727273
-2*x+-0.7575757575757576
-2*x+-0.7878787878787878
-2*x+-0.8181818181818182
-2*x+-0.8484848484848485
-2*x+-0.8787878787878788
-2*x+-0.9090909090909092
-2*x+-0.9393939393939394
-2*x+-0.9696969696969697
-2*x+-1.0
-2*x+-1.0303030303030303
-2*x+-1.0606060606060606
-2*x+-1.0909090909090908
-2*x+-1.1212121212121213
-2*x+-1.1515151515151516
-2*x+-1.1818181818181819
-2*x+-1.2121212121212122
-2*x+-1.2424242424242424
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

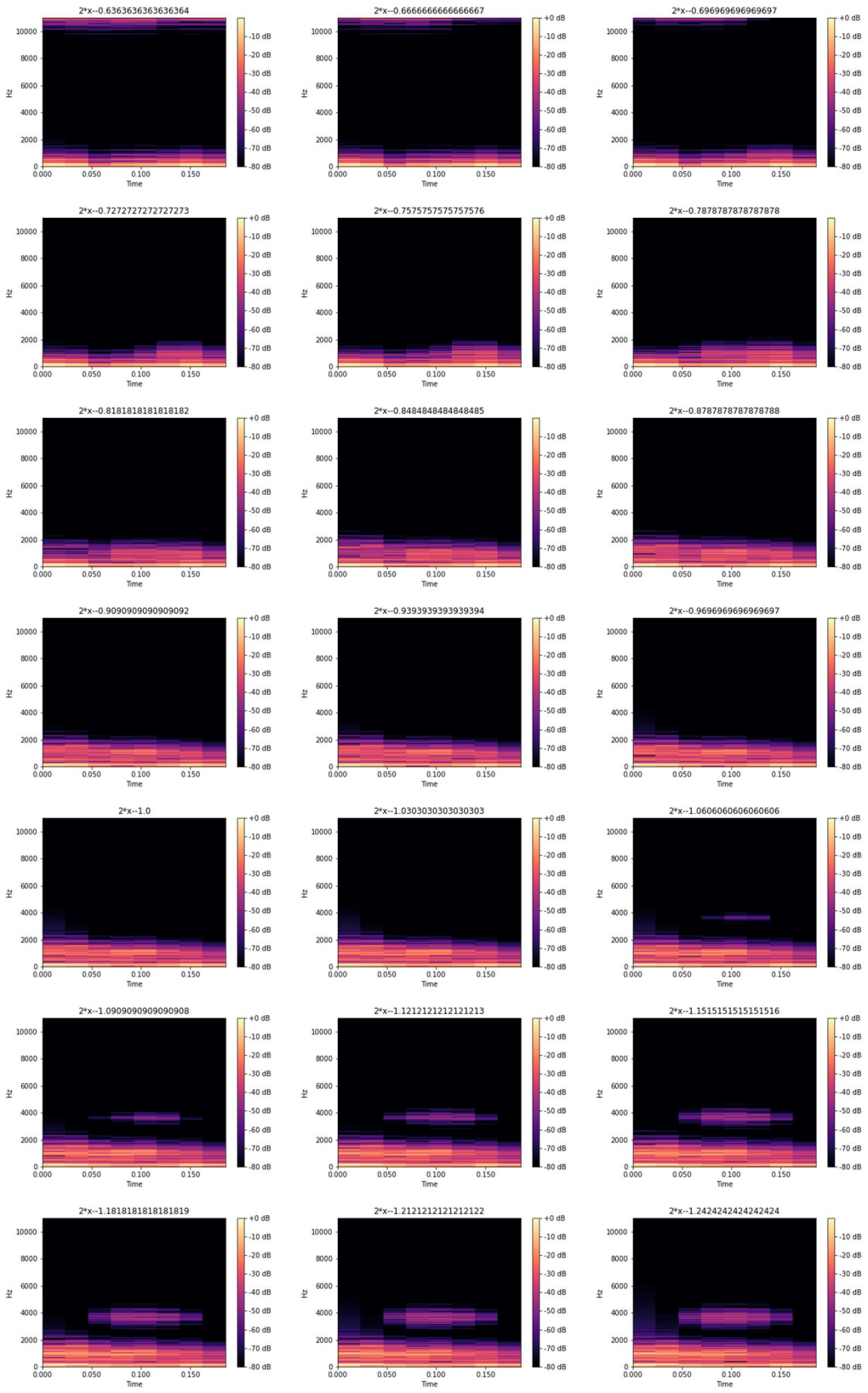

-2*x+-1.2727272727272727
-2*x+-1.303030303030303
-2*x+-1.3333333333333335
-2*x+-1.3636363636363638
-2*x+-1.393939393939394
-2*x+-1.4242424242424243
-2*x+-1.4545454545454546
-2*x+-1.4848484848484849
-2*x+-1.5151515151515151
-2*x+-1.5454545454545454
-2*x+-1.5757575757575757
-2*x+-1.6060606060606062
-2*x+-1.6363636363636365
-2*x+-1.6666666666666667
-2*x+-1.696969696969697
-2*x+-1.7272727272727273
-2*x+-1.7575757575757576
-2*x+-1.7878787878787878
-2*x+-1.8181818181818183
-2*x+-1.8484848484848486
-2*x+-1.878787878787879
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

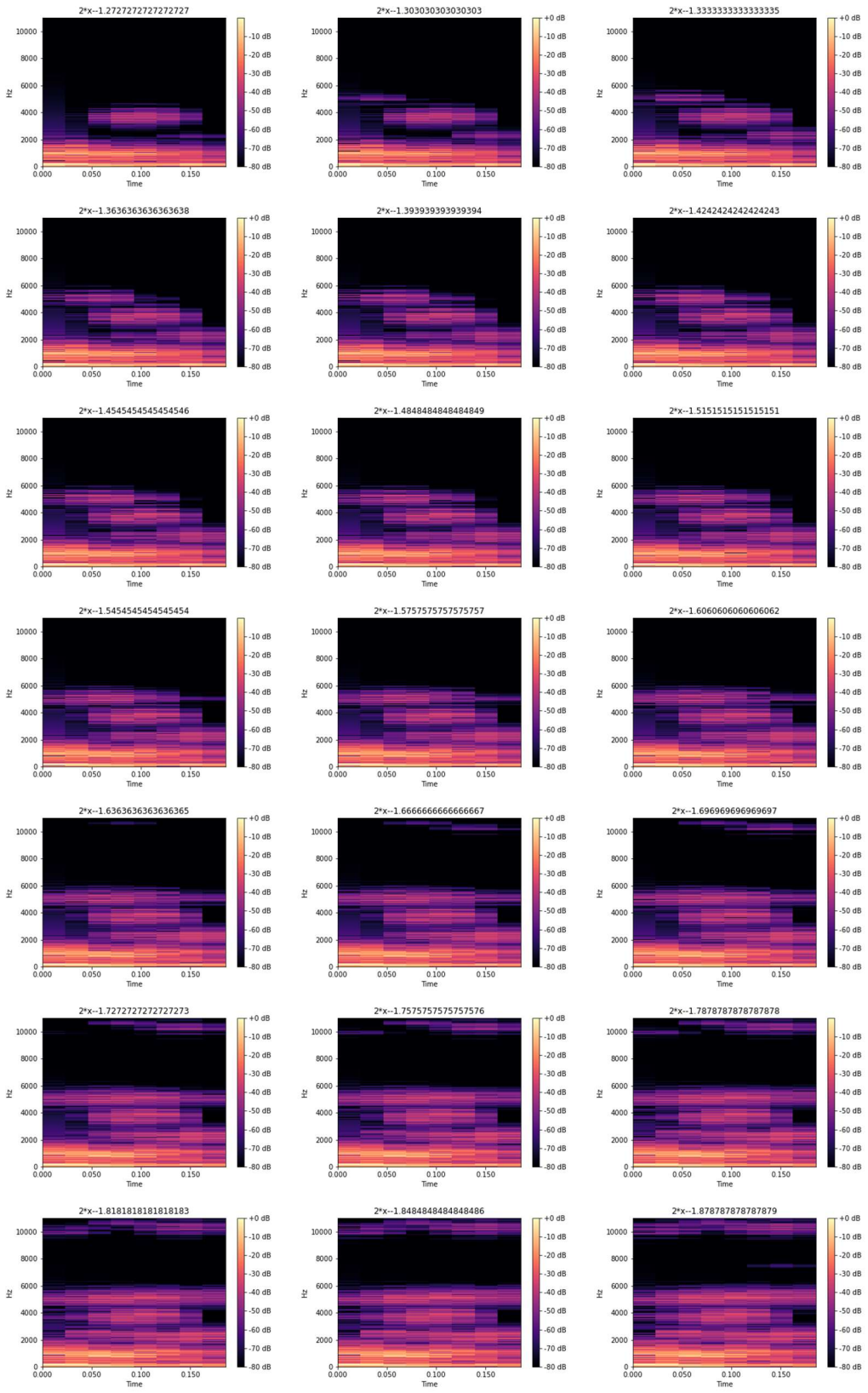

-2*x+-1.9090909090909092
-2*x+-1.9393939393939394
-2*x+-1.9696969696969697
-2*x+-2.0
-2*x+-2.0303030303030303
-2*x+-2.0606060606060606
-2*x+-2.090909090909091
-2*x+-2.121212121212121
-2*x+-2.1515151515151514
-2*x+-2.1818181818181817
-2*x+-2.2121212121212124
-2*x+-2.2424242424242427
-2*x+-2.272727272727273
-2*x+-2.303030303030303
-2*x+-2.3333333333333335
-2*x+-2.3636363636363638
-2*x+-2.393939393939394
-2*x+-2.4242424242424243
-2*x+-2.4545454545454546
-2*x+-2.484848484848485
-2*x+-2.515151515151515
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

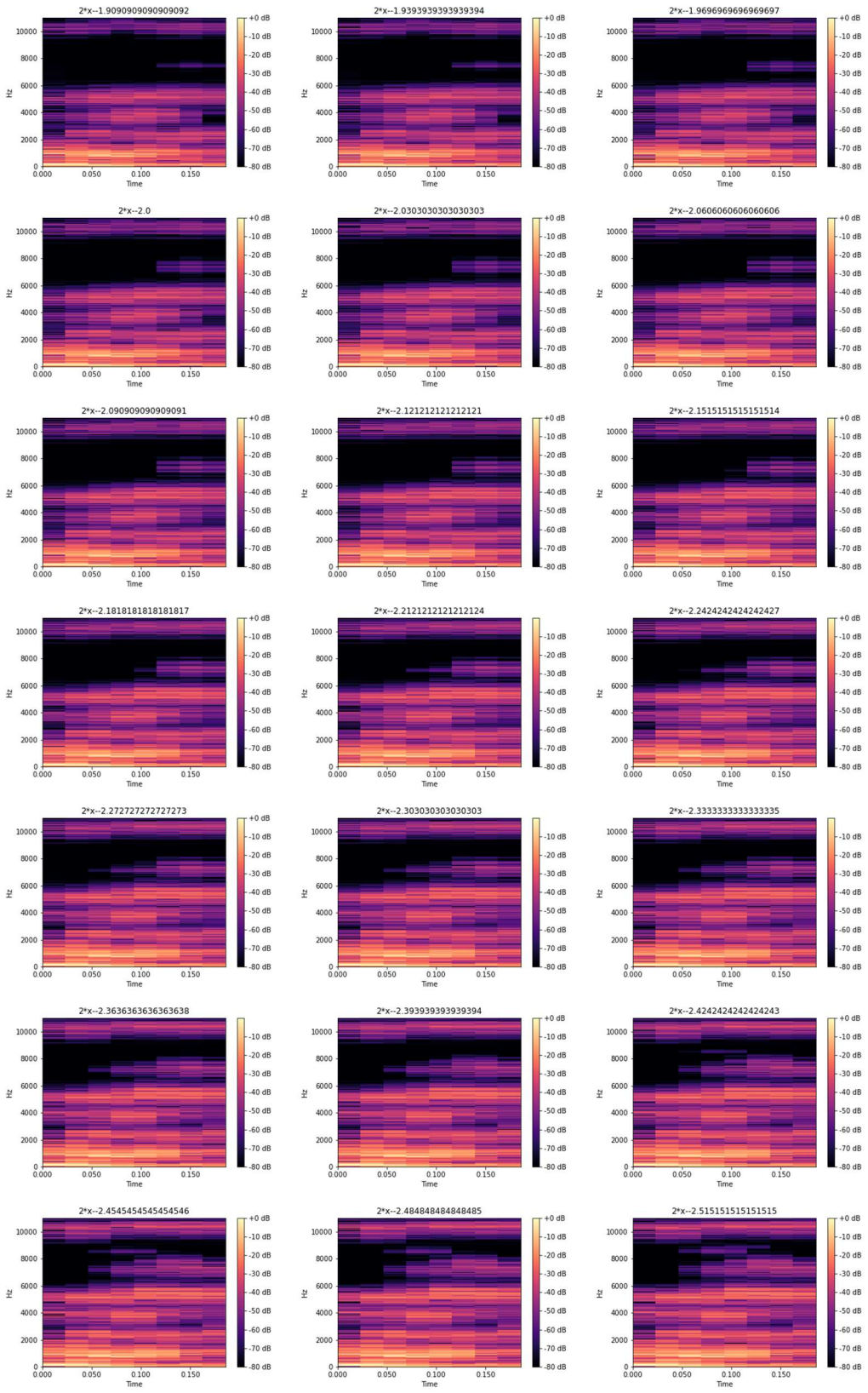


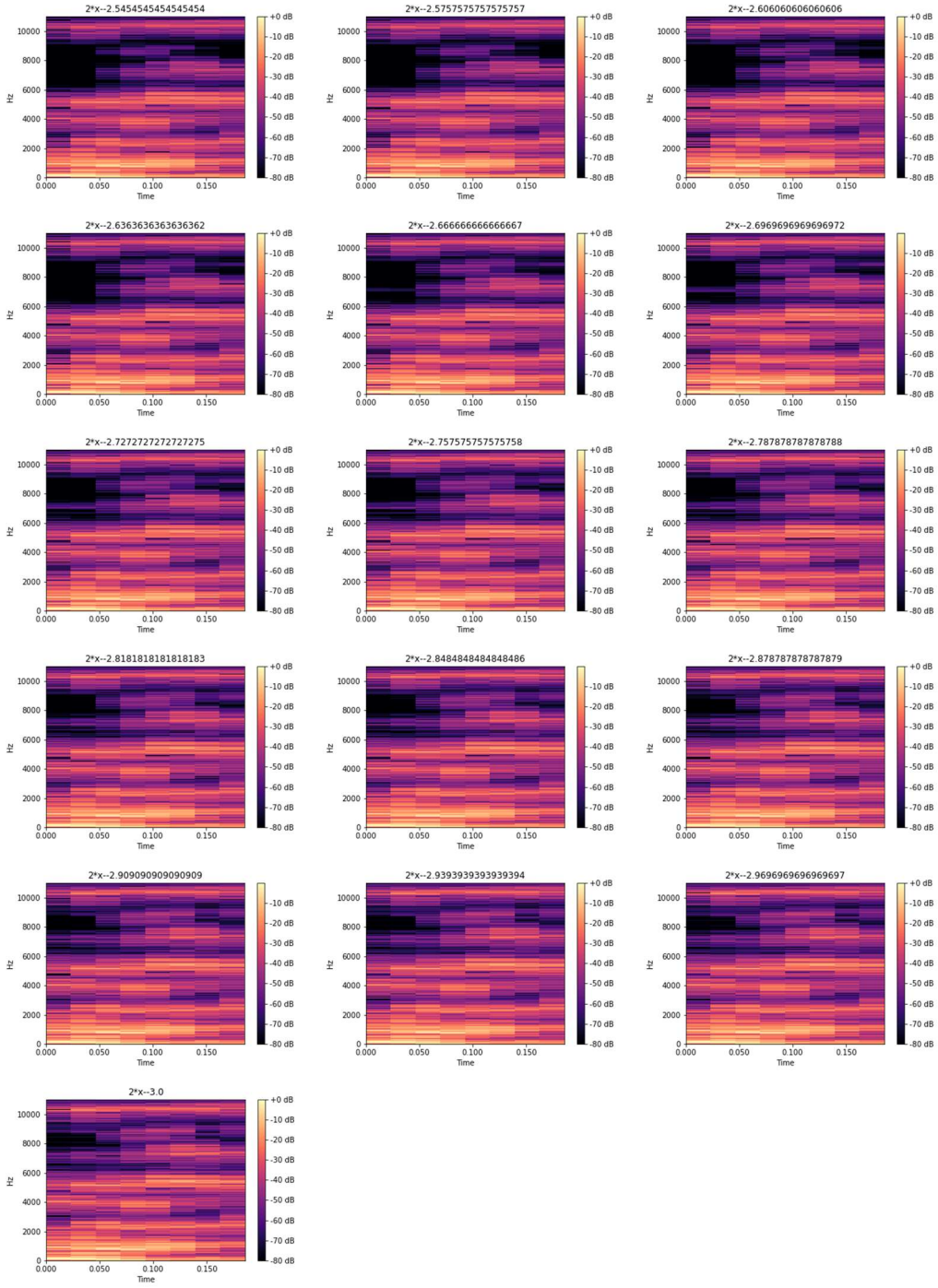

### *7.4.2.2 F(x) = 2x – m for m = [0,3] in steps of 0.03*


2*x--0.0
2*x--0.030303030303030304
2*x--0.06060606060606061
2*x--0.09090909090909091
2*x--0.12121212121212122
2*x--0.15151515151515152
2*x--0.18181818181818182
2*x--0.21212121212121213
2*x--0.24242424242424243
2*x--0.2727272727272727
2*x--0.30303030303030304
2*x--0.33333333333333337
2*x--0.36363636363636365
2*x--0.3939393939393939
2*x--0.42424242424242425
2*x--0.4545454545454546
2*x--0.48484848484848486
2*x--0.5151515151515151
2*x--0.5454545454545454
2*x--0.5757575757575758
2*x--0.6060606060606061
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


2*x--0.6363636363636364
2*x--0.6666666666666667
2*x--0.6969696969696697
2*x--0.7272727272727273
2*x--0.7575757575757576
2*x--0.7878787878787878
2*x--0.8181818181818182
2*x--0.8484848484848485
2*x--0.8787878787878788
2*x--0.9090909090909092
2*x--0.9393939393939394
2*x--0.9696969696969697
2*x--1.0
2*x--1.0303030303030303
2*x--1.0606060606060606
2*x--1.0909090909090908
2*x--1.1212121212121213
2*x--1.1515151515151516
2*x--1.1818181818181819
2*x--1.2121212121212122
2*x--1.2424242424242424
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


2*x--1.2727272727272727
2*x--1.3030303030303303
2*x--1.3333333333333335
2*x--1.3636363636363638
2*x--1.393939393939394
2*x--1.4242424242424243
2*x--1.4545454545454546
2*x--1.4848484848484849
2*x--1.5151515151515151
2*x--1.5454545454545454
2*x--1.5757575757575757
2*x--1.6060606060606062
2*x--1.6363636363636365
2*x--1.6666666666666667
2*x--1.696969696969697
2*x--1.7272727272727273
2*x--1.7575757575757576
2*x--1.7878787878787878
2*x--1.8181818181818183
2*x--1.8484848484848486
2*x--1.878787878787879
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

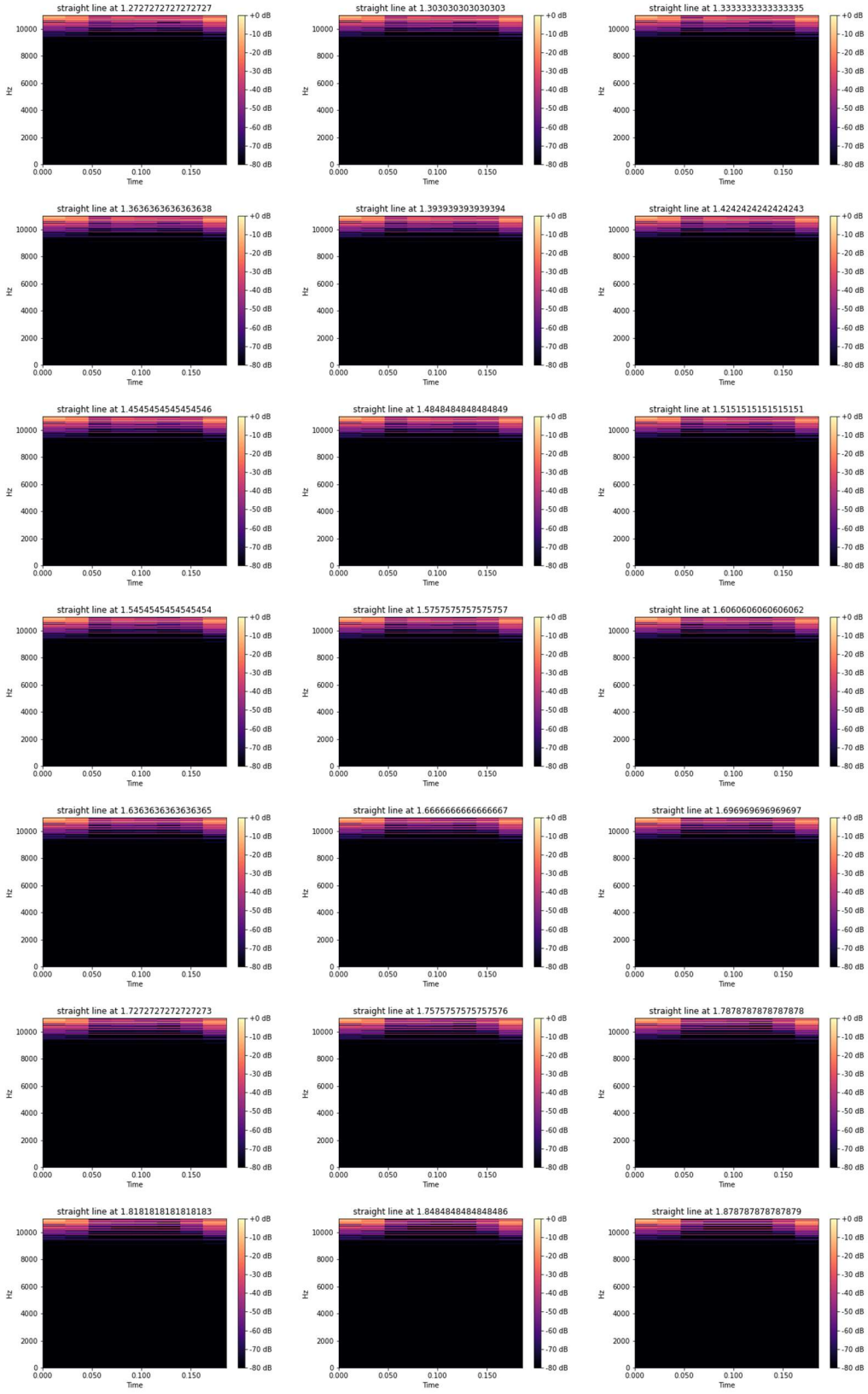

2*x--1.9090909090909092
2*x--1.9393939393939394
2*x--1.9696969696969697
2*x--2.0
2*x--2.0303030303030303
2*x--2.0606060606060606
2*x--2.090909090909091
2*x--2.121212121212121
2*x--2.1515151515151514
2*x--2.1818181818181817
2*x--2.2121212121212124
2*x--2.2424242424242427
2*x--2.272727272727273
2*x--2.303030303030303
2*x--2.3333333333333335
2*x--2.3636363636363638
2*x--2.393939393939394
2*x--2.4242424242424243
2*x--2.4545454545454546
2*x--2.484848484848485
2*x--2.515151515151515
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB



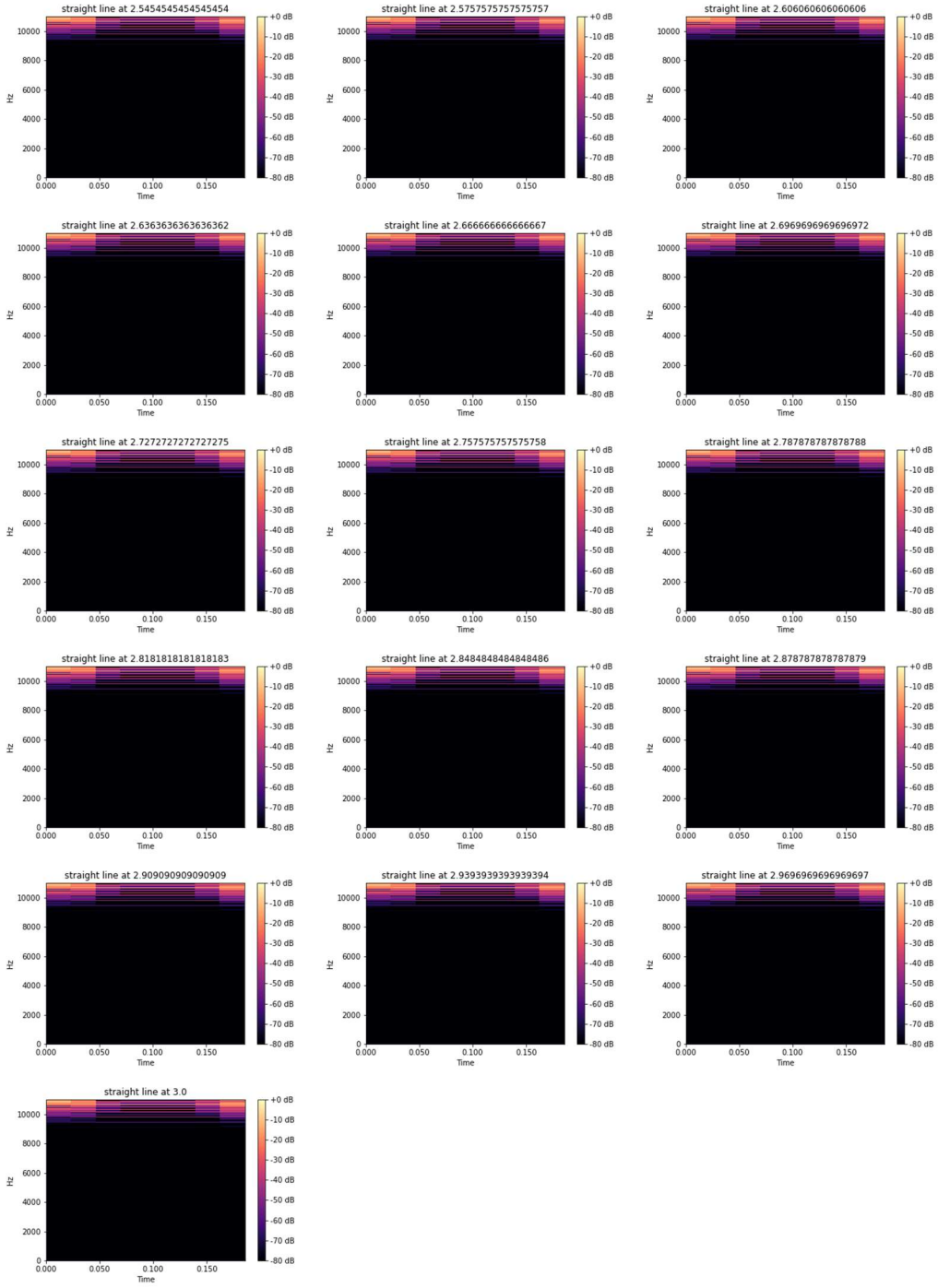

### *7.4.2.3 F(x) = G for G = [0,3] in steps of 0.03*


straight line at 0.0
straight line at 0.030303030303030304
straight line at 0.06060606060606061
straight line at 0.09090909090909091
straight line at 0.12121212121212122
straight line at 0.15151515151515152
straight line at 0.18181818181818182
straight line at 0.21212121212121213
straight line at 0.24242424242424243
straight line at 0.2727272727272727
straight line at 0.30303030303030304
straight line at 0.33333333333333337
straight line at 0.36363636363636365
straight line at 0.3939393939393939
straight line at 0.42424242424242425
straight line at 0.4545454545454546
straight line at 0.48484848484848486
straight line at 0.5151515151515151
straight line at 0.5454545454545454
straight line at 0.5757575757575758
straight line at 0.6060606060606061
Hz
Time

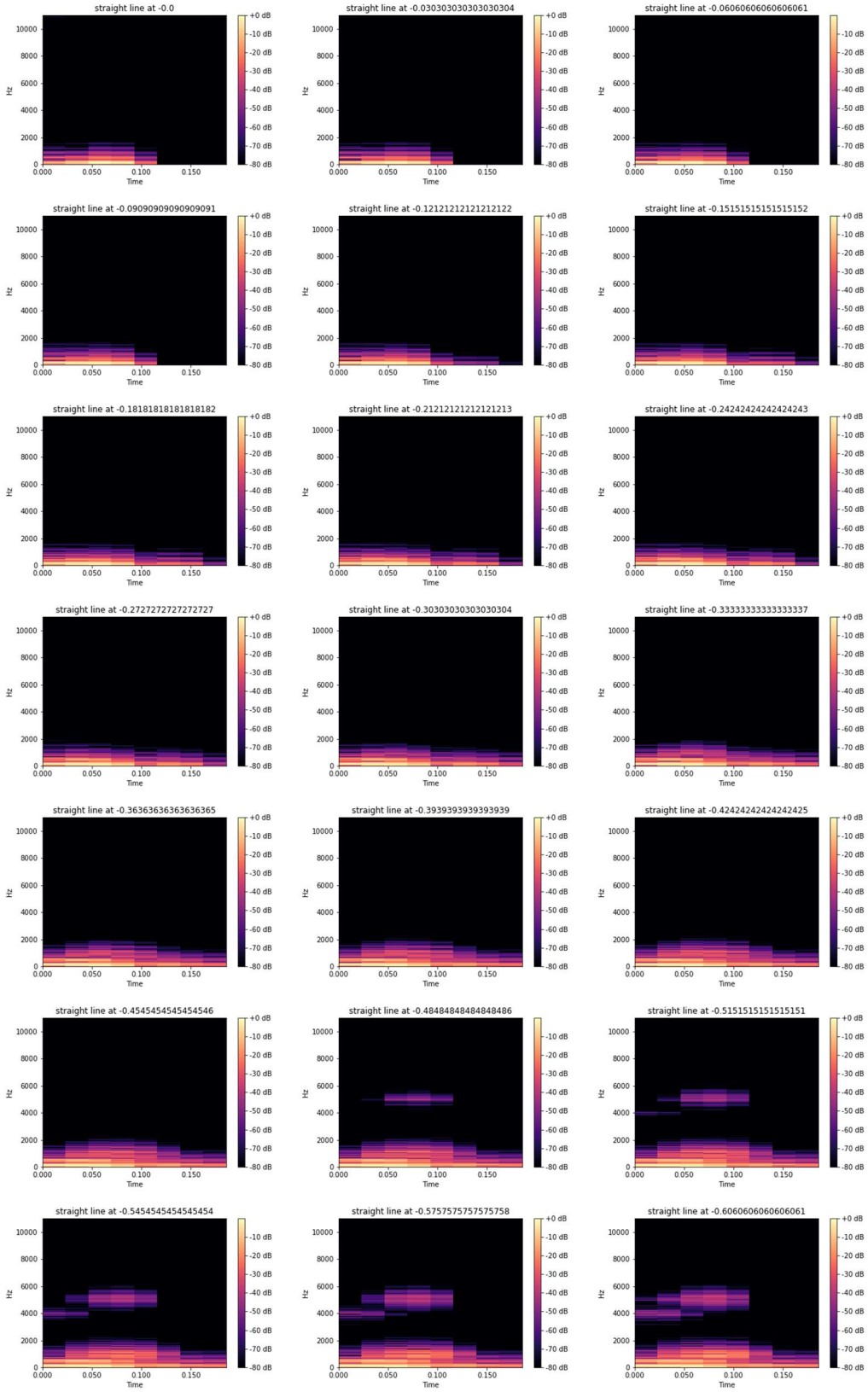

straight line at 0.6363636363636364
straight line at 0.6666666666666667
straight line at 0.696969696969697
straight line at 0.7272727272727273
straight line at 0.7575757575757576
straight line at 0.7878787878787878
straight line at 0.8181818181818182
straight line at 0.8484848484848485
straight line at 0.8787878787878788
straight line at 0.9090909090909092
straight line at 0.9393939393939394
straight line at 0.9696969696969697
straight line at 1.0
straight line at 1.0303030303030303
straight line at 1.0606060606060606
straight line at 1.0909090909090908
straight line at 1.1212121212121213
straight line at 1.1515151515151516
straight line at 1.1818181818181819
straight line at 1.2121212121212122
straight line at 1.2424242424242424
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

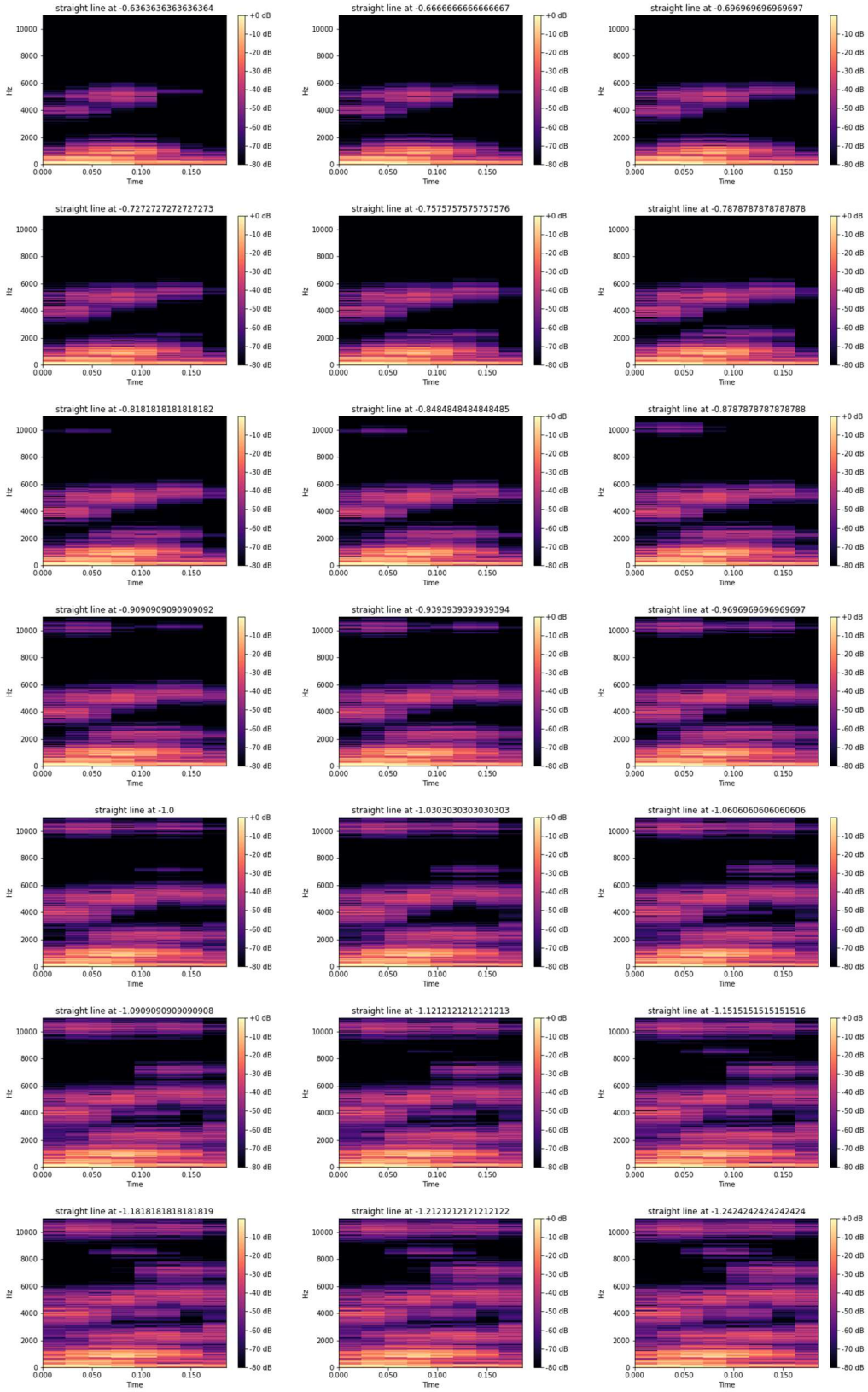

straight line at 1.27272727272727
straight line at 1.3030303030303
straight line at 1.3333333333335
straight line at 1.3636363636363638
straight line at 1.393939393939394
straight line at 1.4242424242424243
straight line at 1.4545454545454546
straight line at 1.4848484848484849
straight line at 1.5151515151515151
straight line at 1.5454545454545454
straight line at 1.5757575757575757
straight line at 1.6060606060606062
straight line at 1.6363636363636365
straight line at 1.6666666666666667
straight line at 1.696969696969697
straight line at 1.7272727272727273
straight line at 1.7575757575757576
straight line at 1.7878787878787878
straight line at 1.8181818181818183
straight line at 1.8484848484848486
straight line at 1.878787878787879
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

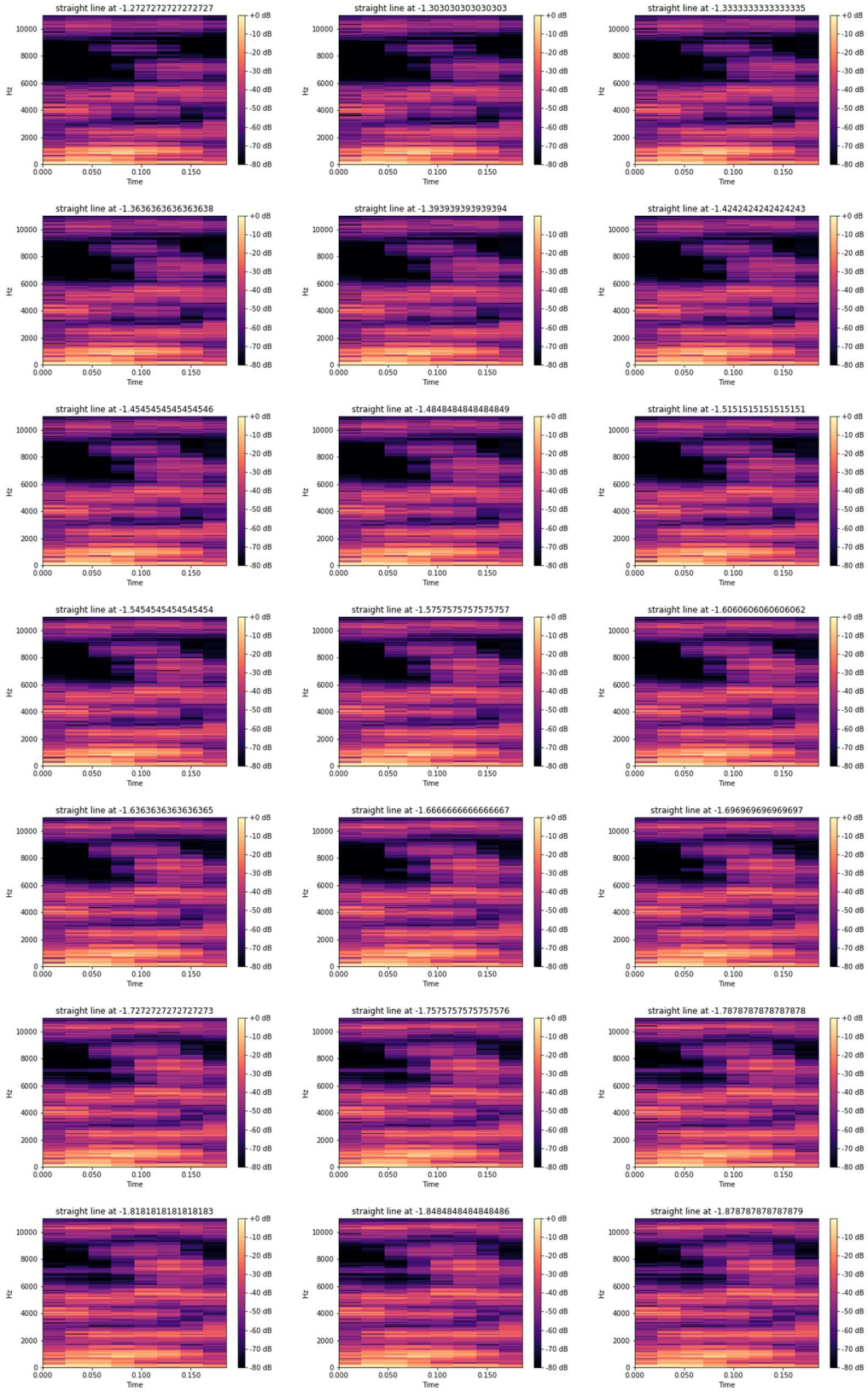

straight line at 1.9090909090909092
straight line at 1.9393939393939394
straight line at 1.9696969696969697
straight line at 2.0
straight line at 2.0303030303030303
straight line at 2.0606060606060606
straight line at 2.090909090909091
straight line at 2.121212121212121
straight line at 2.1515151515151514
straight line at 2.1818181818181817
straight line at 2.2121212121212124
straight line at 2.2424242424242427
straight line at 2.272727272727273
straight line at 2.303030303030303
straight line at 2.3333333333333335
straight line at 2.3636363636363638
straight line at 2.393939393939394
straight line at 2.4242424242424243
straight line at 2.4545454545454546
straight line at 2.484848484848485
straight line at 2.515151515151515
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

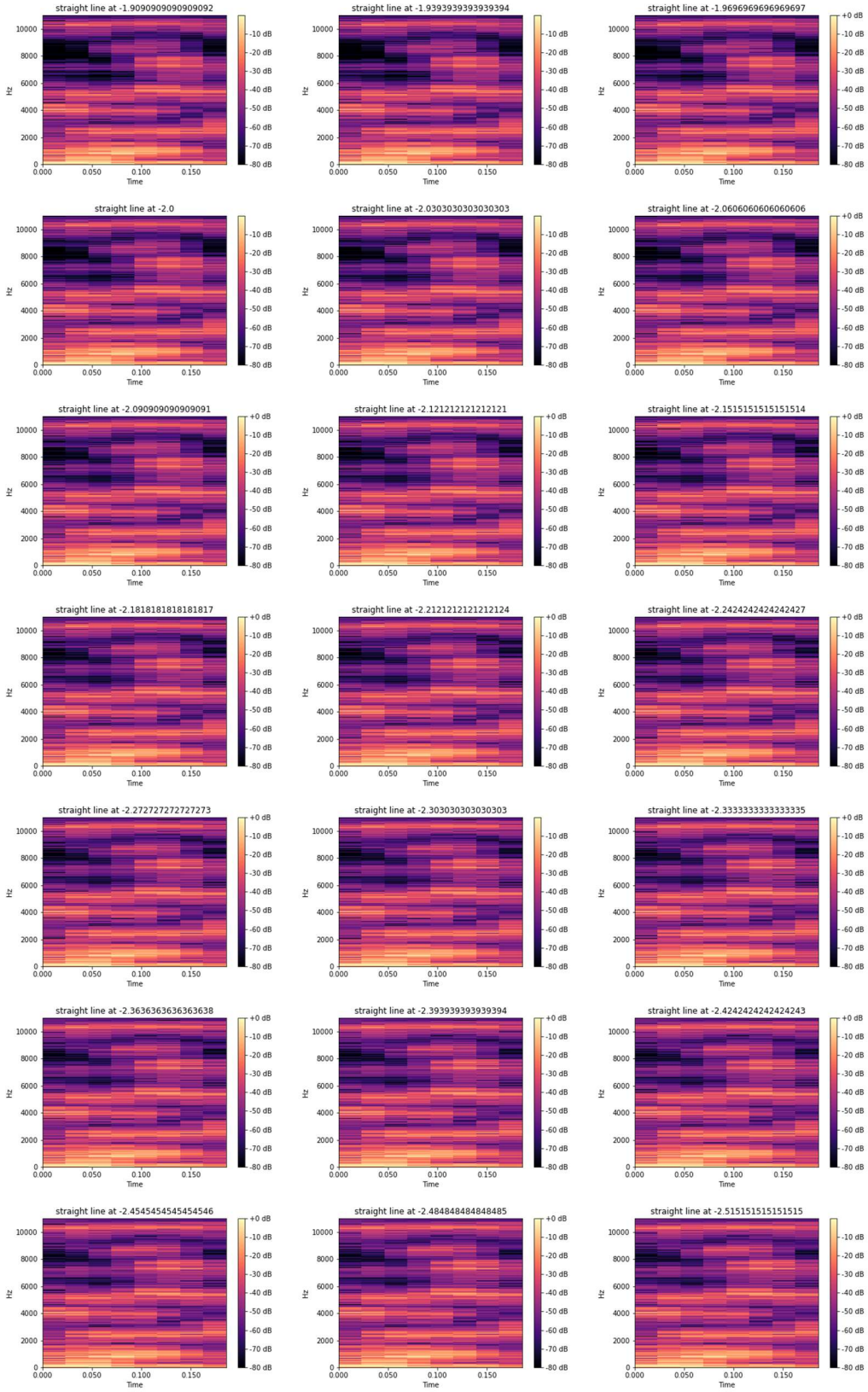


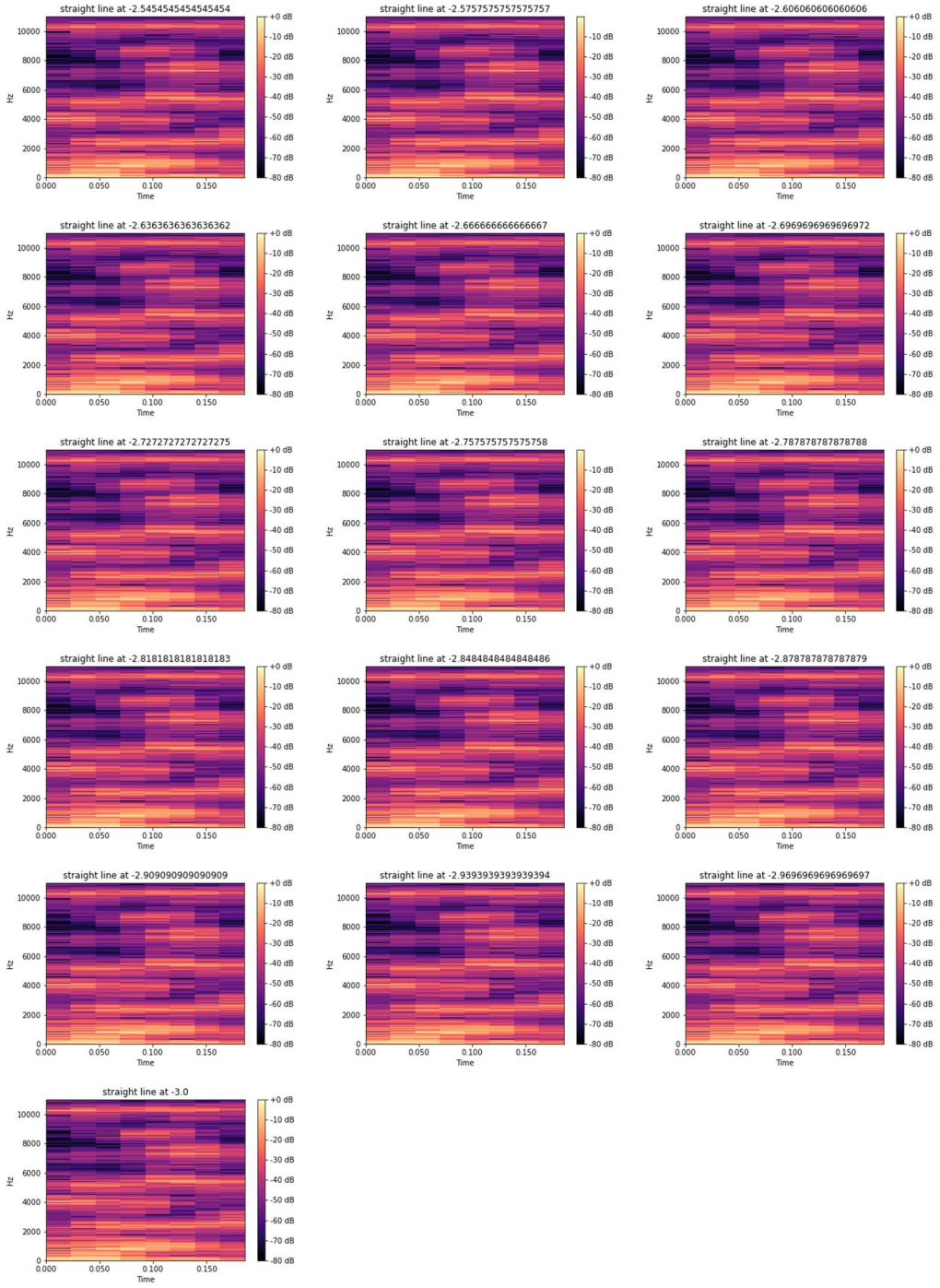

### *7.4.2.4 F(x) = -G for G = [0,3] in steps of 0.03*


straight line at -0.0
straight line at -0.030303030303030304
straight line at -0.06060606060606061
straight line at -0.09090909090909091
straight line at -0.12121212121212122
straight line at -0.15151515151515152
straight line at -0.18181818181818182
straight line at -0.21212121212121213
straight line at -0.24242424242424243
straight line at -0.2727272727272727
straight line at -0.30303030303030304
straight line at -0.33333333333333337
straight line at -0.36363636363636365
straight line at -0.3939393939393939
straight line at -0.42424242424242425
straight line at -0.4545454545454546
straight line at -0.48484848484848486
straight line at -0.5151515151515151
straight line at -0.5454545454545454
straight line at -0.5757575757575758
straight line at -0.6060606060606061
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


straight line at -0.6363636363636364
straight line at -0.6666666666666667
straight line at -0.6969696969696697
straight line at -0.7272727272727273
straight line at -0.7575757575757576
straight line at -0.7878787878787878
straight line at -0.8181818181818182
straight line at -0.8484848484848485
straight line at -0.8787878787878788
straight line at -0.9090909090909092
straight line at -0.9393939393939394
straight line at -0.9696969696969697
straight line at -1.0
straight line at -1.0303030303030303
straight line at -1.0606060606060606
straight line at -1.0909090909090908
straight line at -1.1212121212121213
straight line at -1.1515151515151516
straight line at -1.1818181818181819
straight line at -1.2121212121212122
straight line at -1.2424242424242424
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB


straight line at -1.2727272727272727
straight line at -1.3030303030303303
straight line at -1.3333333333333335
straight line at -1.3636363636363638
straight line at -1.393939393939394
straight line at -1.4242424242424243
straight line at -1.4545454545454546
straight line at -1.4848484848484849
straight line at -1.5151515151515151
straight line at -1.5454545454545454
straight line at -1.5757575757575757
straight line at -1.6060606060606062
straight line at -1.6363636363636365
straight line at -1.6666666666666667
straight line at -1.696969696969697
straight line at -1.7272727272727273
straight line at -1.7575757575757576
straight line at -1.7878787878787878
straight line at -1.8181818181818183
straight line at -1.8484848484848486
straight line at -1.878787878787879
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

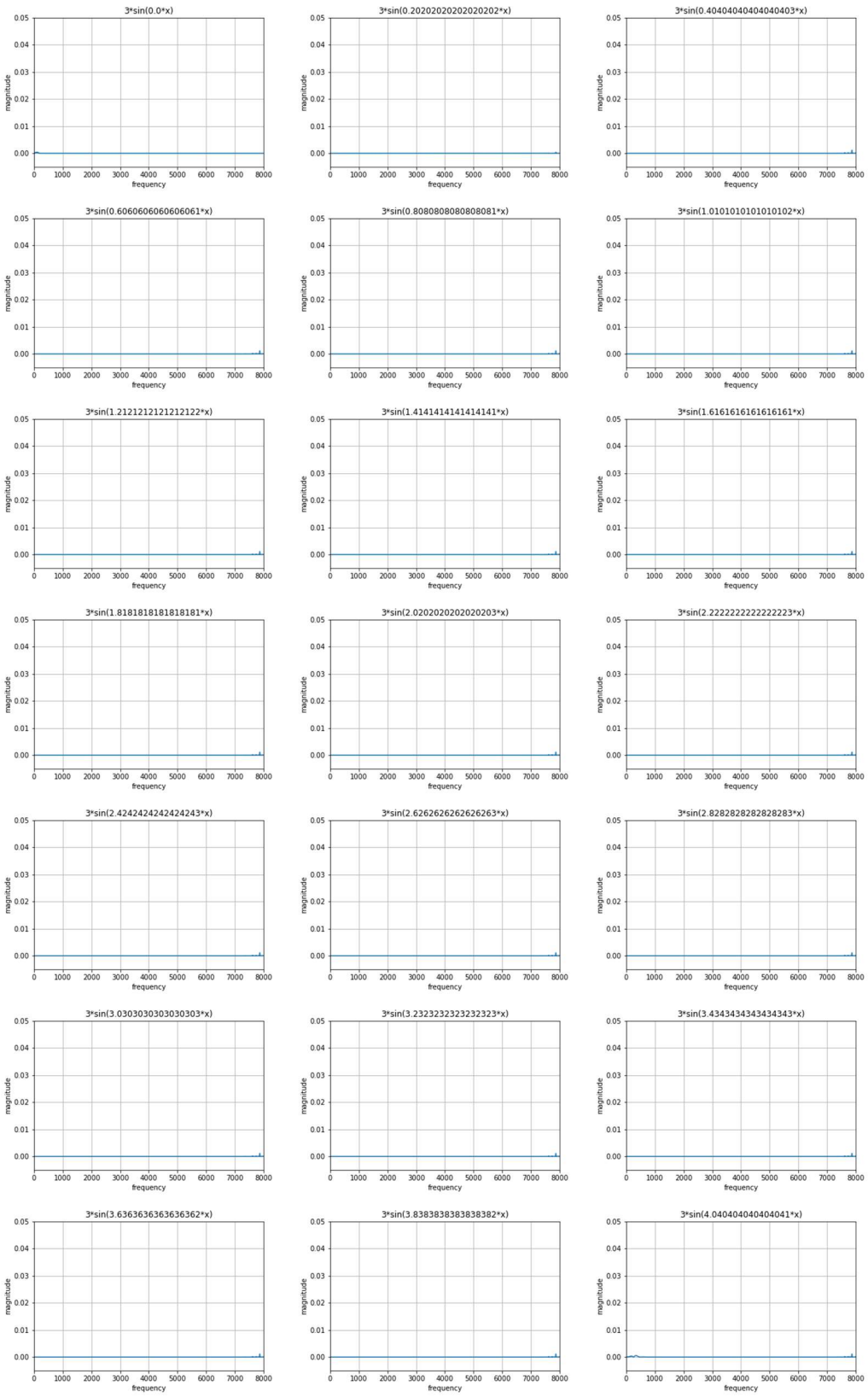

straight line at -1.9090909090909092
straight line at -1.9393939393939394
straight line at -1.9696969696969697
straight line at -2.0
straight line at -2.030303030303303
straight line at -2.0606060606060606
straight line at -2.090909090909091
straight line at -2.121212121212121
straight line at -2.1515151515151514
straight line at -2.1818181818181817
straight line at -2.2121212121212124
straight line at -2.2424242424242427
straight line at -2.272727272727273
straight line at -2.303030303030303
straight line at -2.3333333333333335
straight line at -2.3636363636363638
straight line at -2.393939393939394
straight line at -2.4242424242424243
straight line at -2.4545454545454546
straight line at -2.484848484848485
straight line at -2.515151515151515
Hz
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

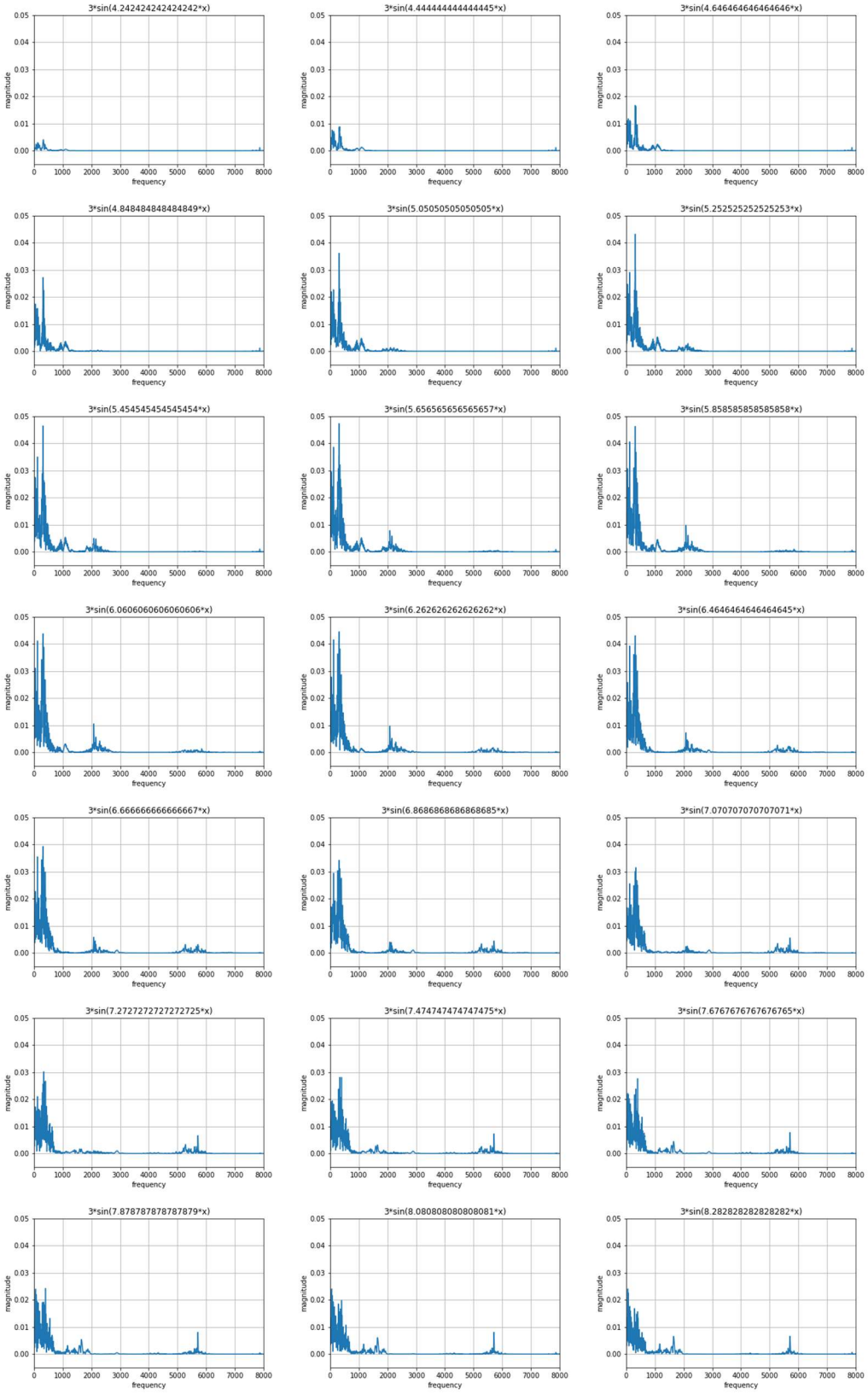


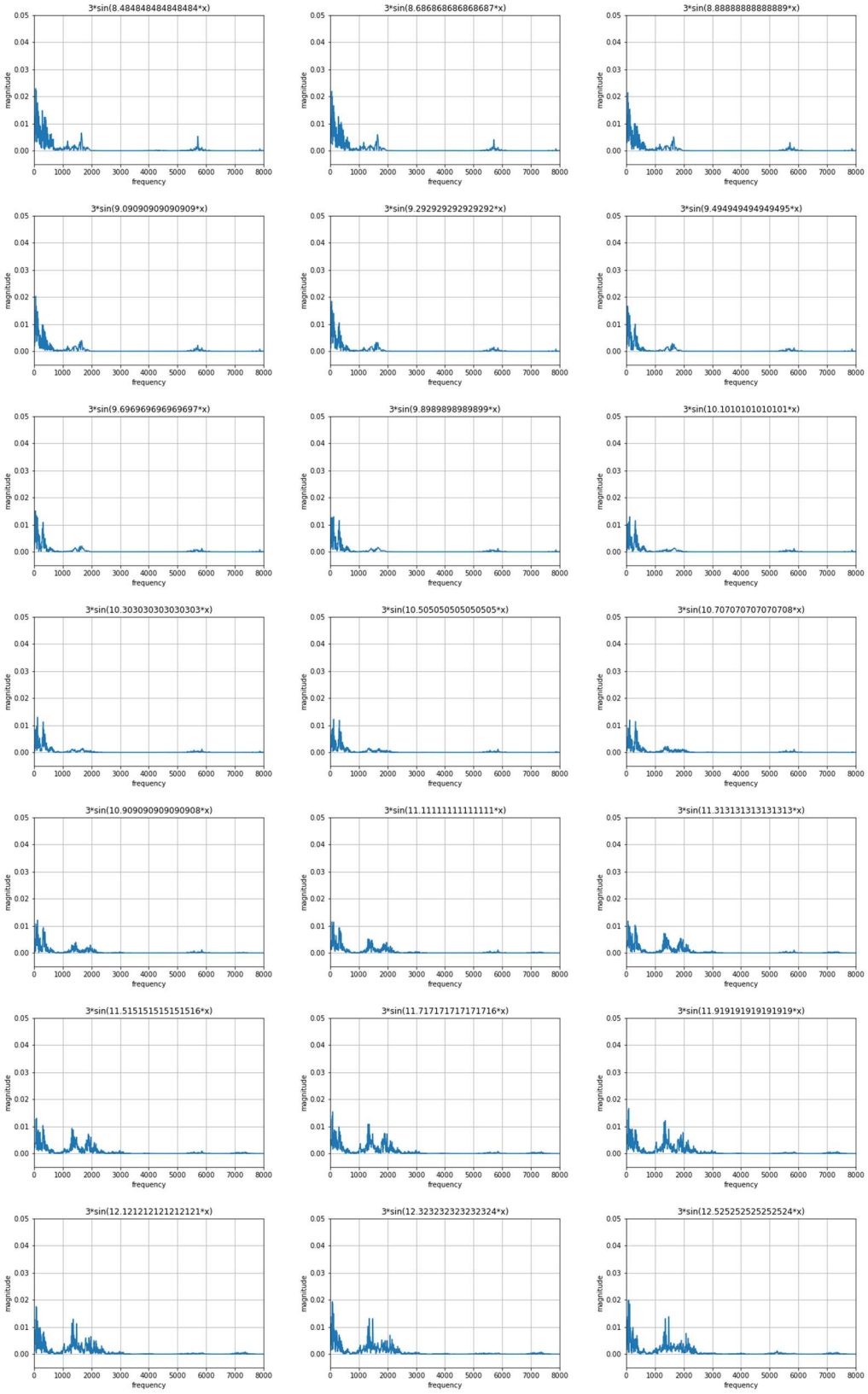

## 7.5 APPENDIX E. FFT SPECTROGRAMS

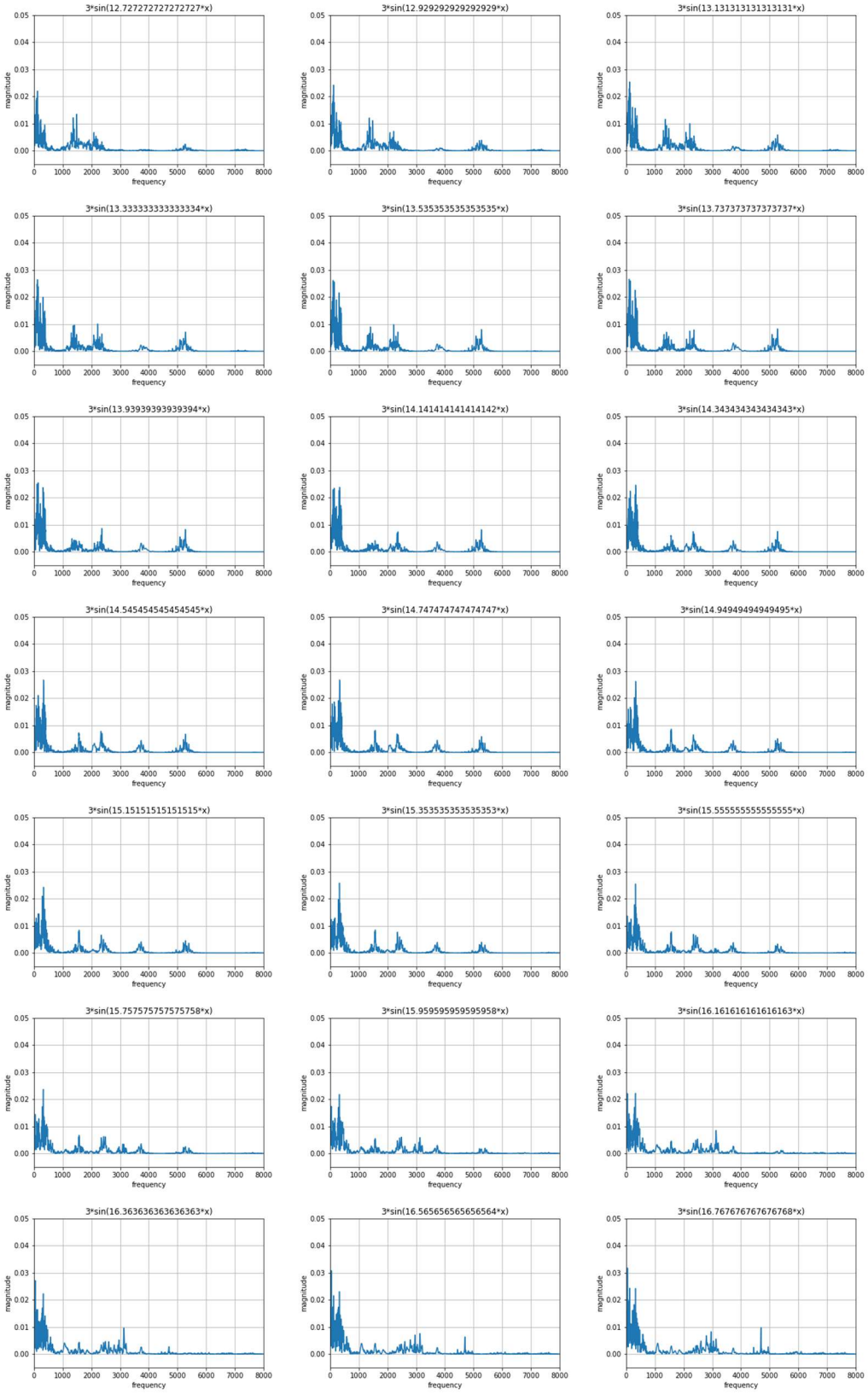
### 7.5.1 Sine/cosine functions

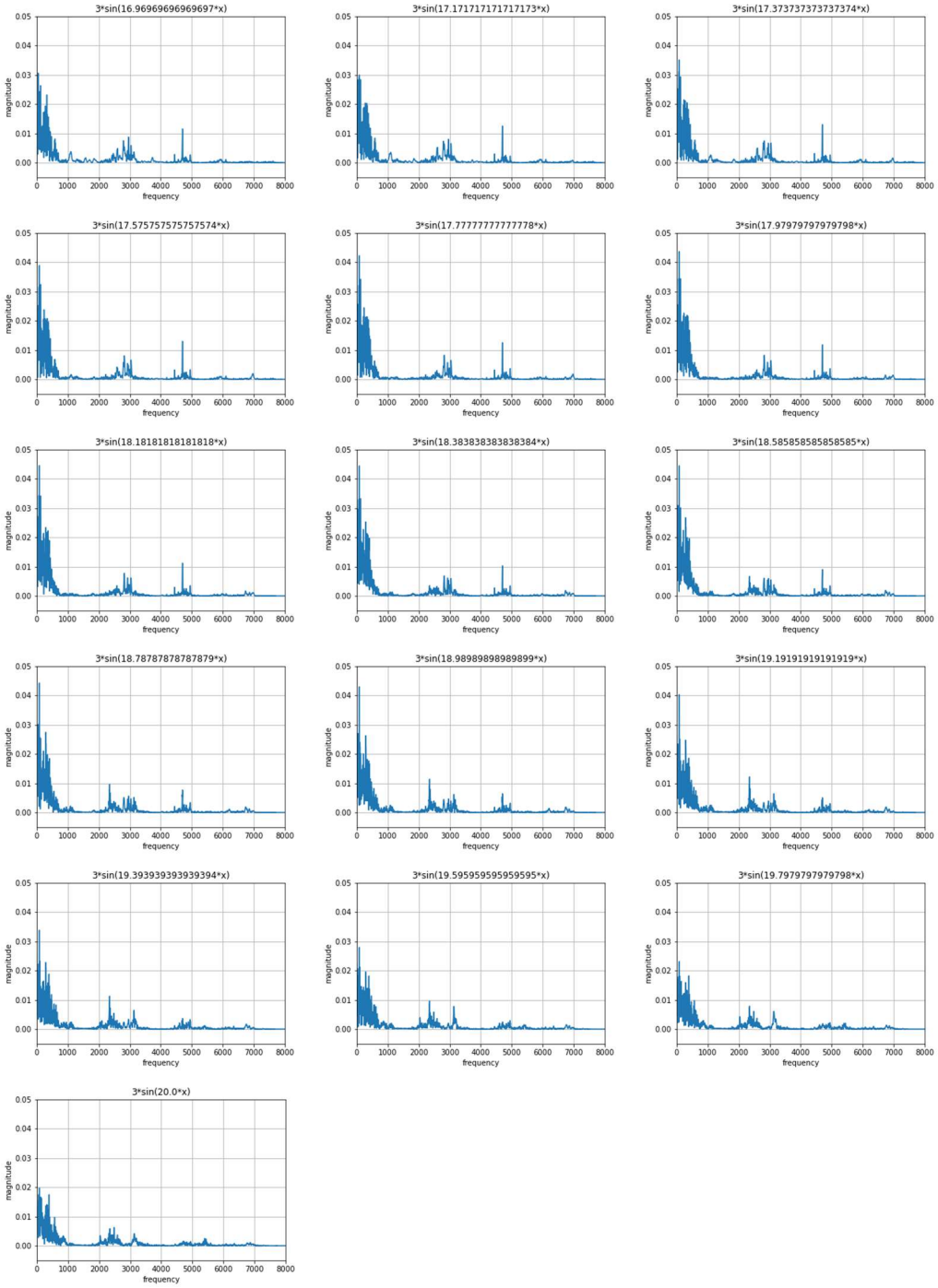
#### *7.5.1.1 F(x) = 3*sin(w*x) for w= [0,20] in steps of 0.3*


3*sin(0.0*x)
3*sin(0.20202020202020202*x)
3*sin(0.40404040404040403*x)
3*sin(0.6060606060606061*x)
3*sin(0.8080808080808081*x)
3*sin(1.0101010101010102*x)
3*sin(1.2121212121212122*x)
3*sin(1.4141414141414141*x)
3*sin(1.6161616161616161*x)
3*sin(1.8181818181818181*x)
3*sin(2.0202020202020203*x)
3*sin(2.2222222222222223*x)
3*sin(2.4242424242424243*x)
3*sin(2.6262626262626263*x)
3*sin(2.8282828282828283*x)
3*sin(3.0303030303030303*x)
3*sin(3.2323232323232323*x)
3*sin(3.4343434343434343*x)
3*sin(3.6363636363636362*x)
3*sin(3.8383838383838382*x)
3*sin(4.040404040404041*x)
magnitude
frequency

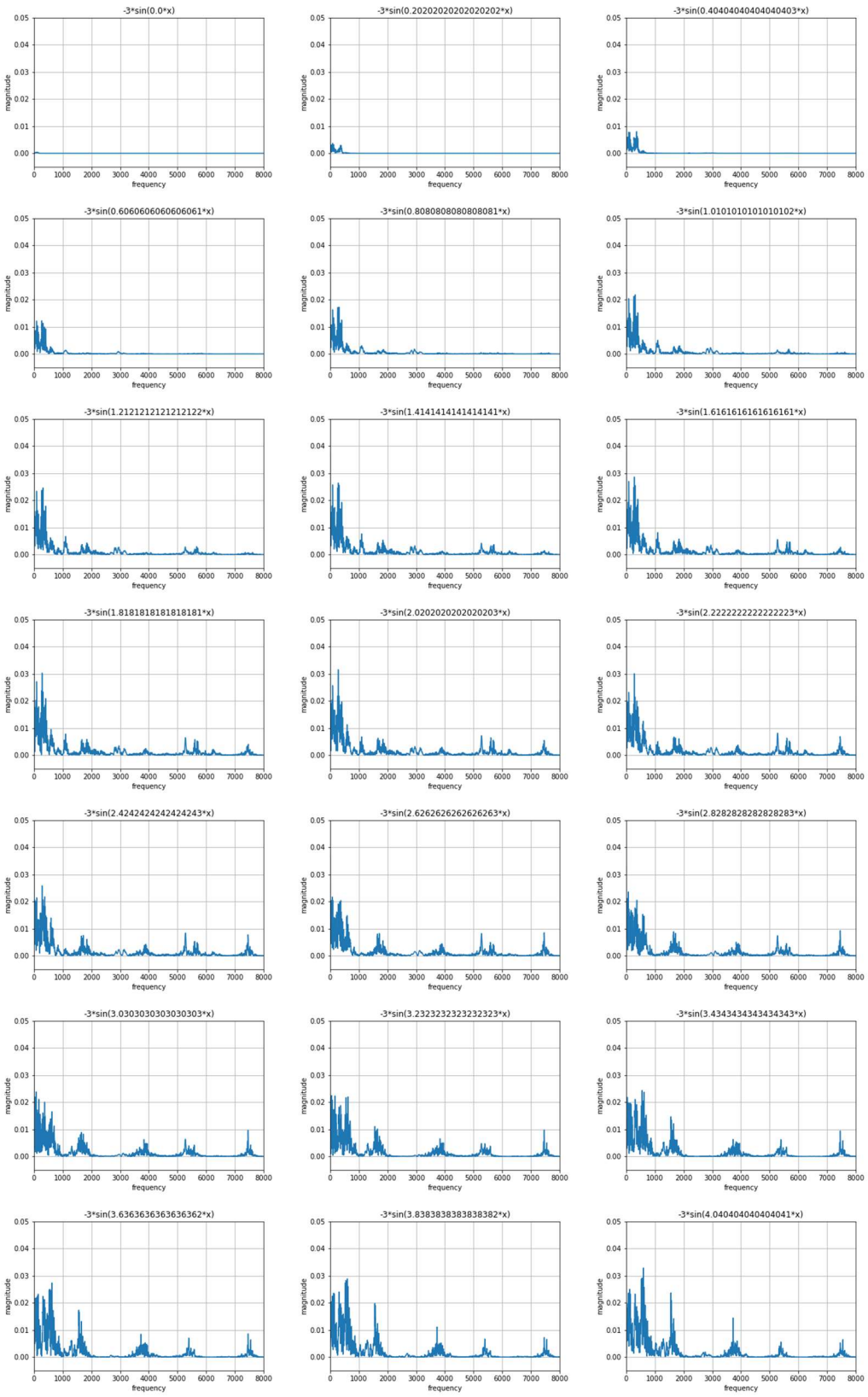

3*sin(4.242424242424242*x)
3*sin(4.444444444444445*x)
3*sin(4.646464646464646*x)
3*sin(4.848484848484849*x)
3*sin(5.050505050505505*x)
3*sin(5.252525252525253*x)
3*sin(5.454545454545454*x)
3*sin(5.656565656565657*x)
3*sin(5.858585858585858*x)
3*sin(6.0606060606060606*x)
3*sin(6.262626262626262*x)
3*sin(6.4646464646464645*x)
3*sin(6.666666666666667*x)
3*sin(6.8686868686868685*x)
3*sin(7.070707070707071*x)
3*sin(7.2727272727272725*x)
3*sin(7.4747474747474745*x)
3*sin(7.6767676767676765*x)
3*sin(7.878787878787879*x)
3*sin(8.080808080808081*x)
3*sin(8.282828282828282*x)
magnitude
frequency

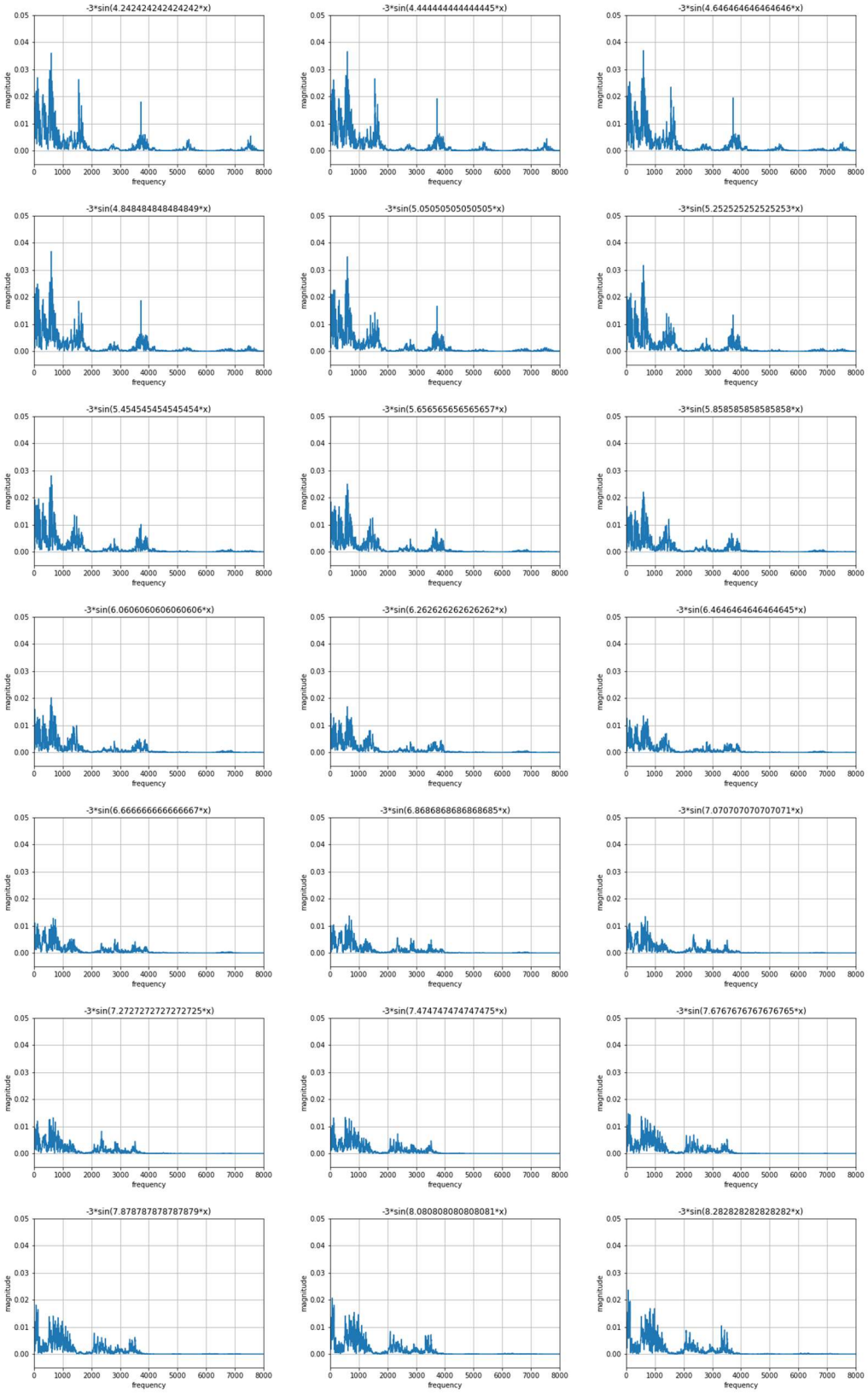

3*sin(8.4848484848484*x)
3*sin(8.686868686868687*x)
3*sin(8.888888888888889*x)
3*sin(9.09090909090909*x)
3*sin(9.292929292929292*x)
3*sin(9.494949494949495*x)
3*sin(9.696969696969697*x)
3*sin(9.8989898989899*x)
3*sin(10.1010101010101*x)
3*sin(10.303030303030303*x)
3*sin(10.505050505050505*x)
3*sin(10.707070707070708*x)
3*sin(10.909090909090908*x)
3*sin(11.111111111111111*x)
3*sin(11.313131313131313*x)
3*sin(11.515151515151516*x)
3*sin(11.717171717171716*x)
3*sin(11.919191919191919*x)
3*sin(12.121212121212121*x)
3*sin(12.323232323232324*x)
3*sin(12.525252525252524*x)
magnitude
frequency

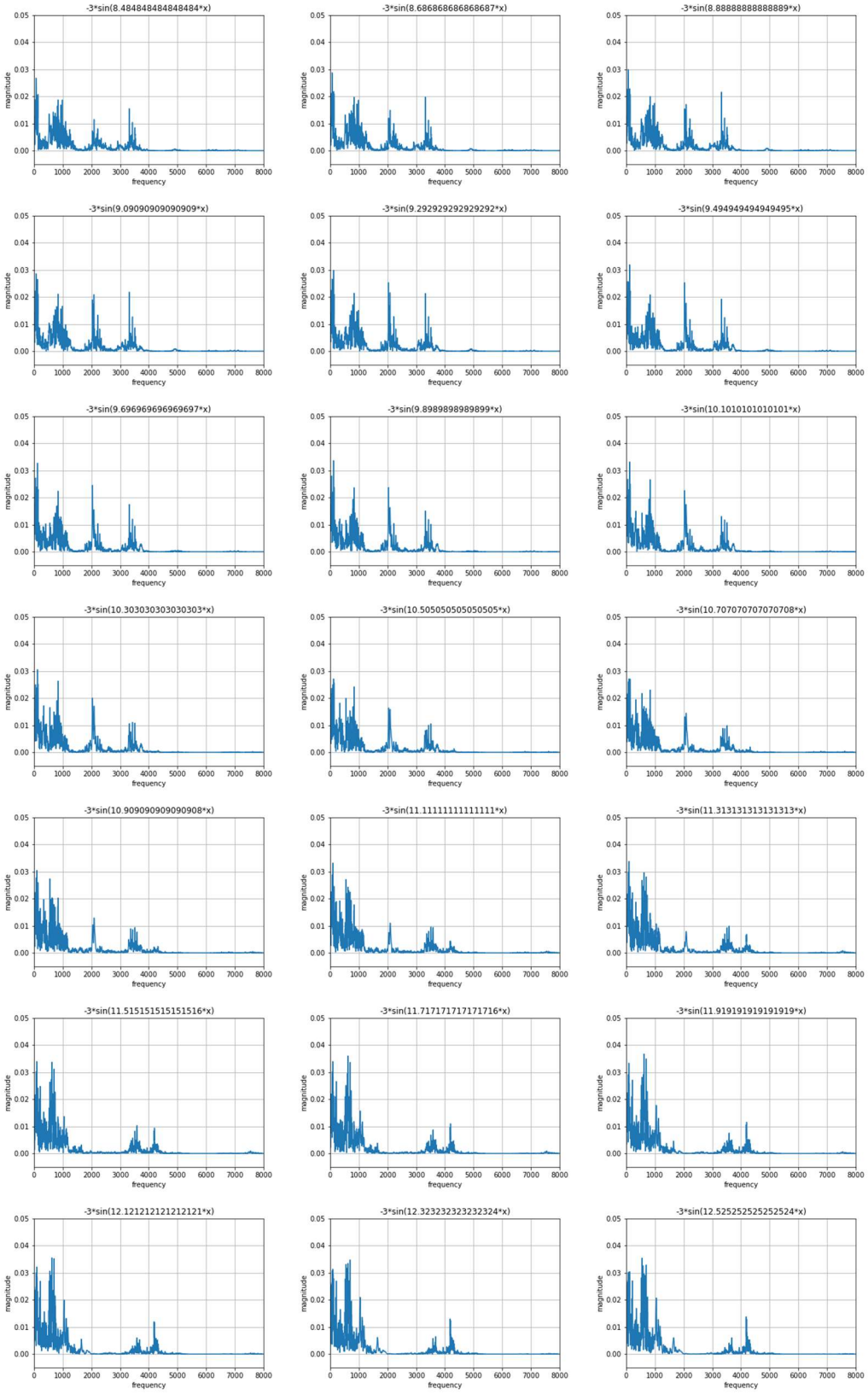

3*sin(12.7272727272727*x)
3*sin(12.929292929292929*x)
3*sin(13.1313131313131*x)
3*sin(13.333333333333334*x)
3*sin(13.535353535353535*x)
3*sin(13.737373737373737*x)
3*sin(13.939393939393394*x)
3*sin(14.141414141414142*x)
3*sin(14.343434343434343*x)
3*sin(14.545454545454545*x)
3*sin(14.747474747474747*x)
3*sin(14.94949494949495*x)
3*sin(15.151515151515515*x)
3*sin(15.353535353535353*x)
3*sin(15.5555555555555*x)
3*sin(15.757575757575758*x)
3*sin(15.959595959595958*x)
3*sin(16.161616161616163*x)
3*sin(16.363636363636363*x)
3*sin(16.565656565656564*x)
3*sin(16.767676767676768*x)
magnitude
frequency

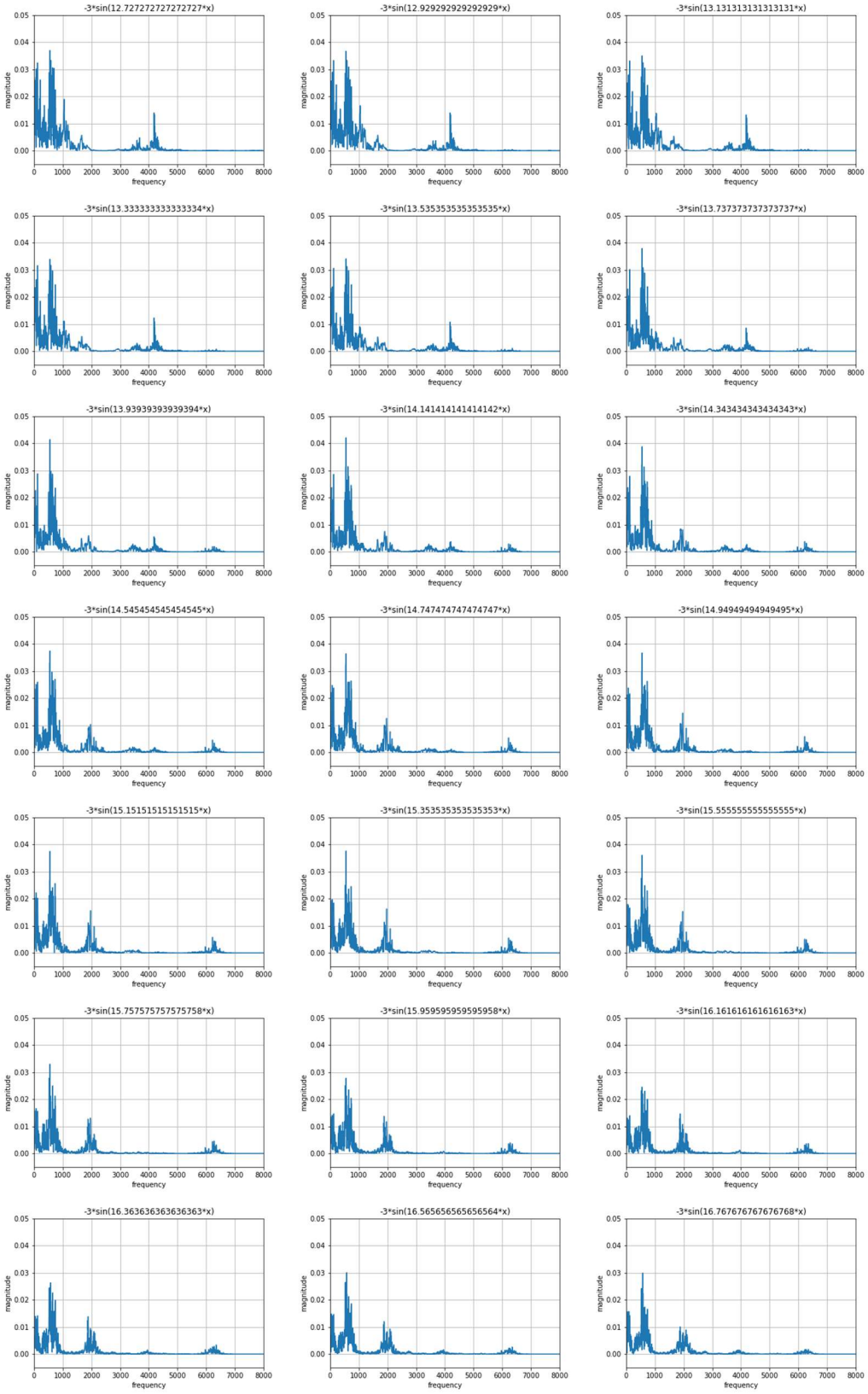


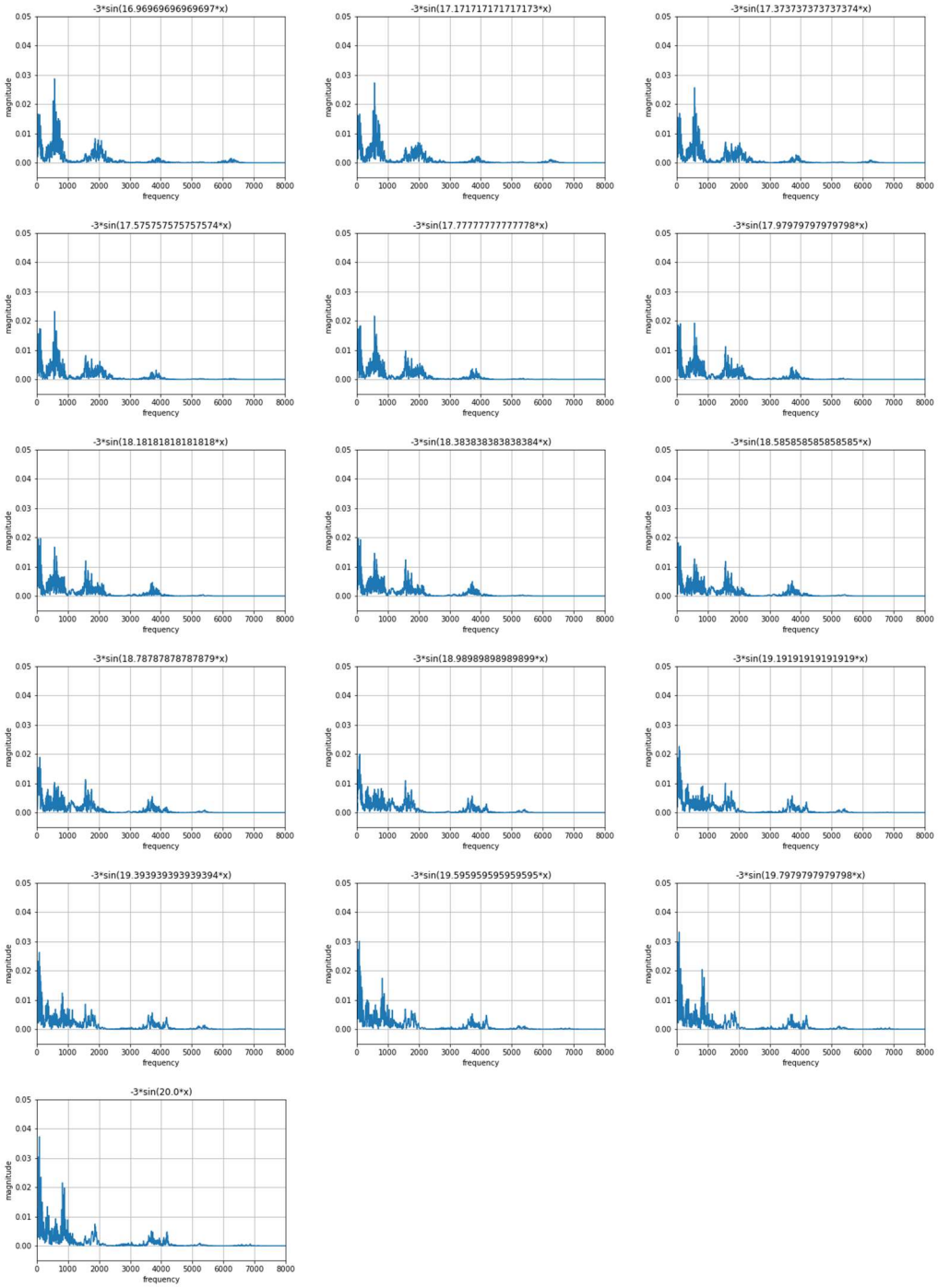

### *7.5.1.2 F(x) = -3*sin(w*x) for w = [0,20] in steps of 0.3*


-3*sin(0.0*x)
-3*sin(0.20202020202020202*x)
-3*sin(0.40404040404040403*x)
-3*sin(0.6060606060606061*x)
-3*sin(0.8080808080808081*x)
-3*sin(1.0101010101010102*x)
-3*sin(1.2121212121212122*x)
-3*sin(1.4141414141414141*x)
-3*sin(1.6161616161616161*x)
-3*sin(1.8181818181818181*x)
-3*sin(2.0202020202020203*x)
-3*sin(2.2222222222222223*x)
-3*sin(2.4242424242424243*x)
-3*sin(2.6262626262626263*x)
-3*sin(2.8282828282828283*x)
-3*sin(3.0303030303030303*x)
-3*sin(3.2323232323232323*x)
-3*sin(3.4343434343434343*x)
-3*sin(3.6363636363636362*x)
-3*sin(3.8383838383838382*x)
-3*sin(4.040404040404041*x)
magnitude
frequency


-3*sin(4.2424242424242*x)
-3*sin(4.444444444444445*x)
-3*sin(4.646464646464646*x)
-3*sin(4.848484848484849*x)
-3*sin(5.050505050505*x)
-3*sin(5.252525252525253*x)
-3*sin(5.454545454545454*x)
-3*sin(5.656565656565657*x)
-3*sin(5.858585858585858*x)
-3*sin(6.0606060606060606*x)
-3*sin(6.262626262626262*x)
-3*sin(6.4646464646464645*x)
-3*sin(6.666666666666667*x)
-3*sin(6.8686868686868685*x)
-3*sin(7.070707070707071*x)
-3*sin(7.2727272727272725*x)
-3*sin(7.474747474747475*x)
-3*sin(7.6767676767676765*x)
-3*sin(7.878787878787879*x)
-3*sin(8.080808080808081*x)
-3*sin(8.282828282828282*x)
magnitude
frequency

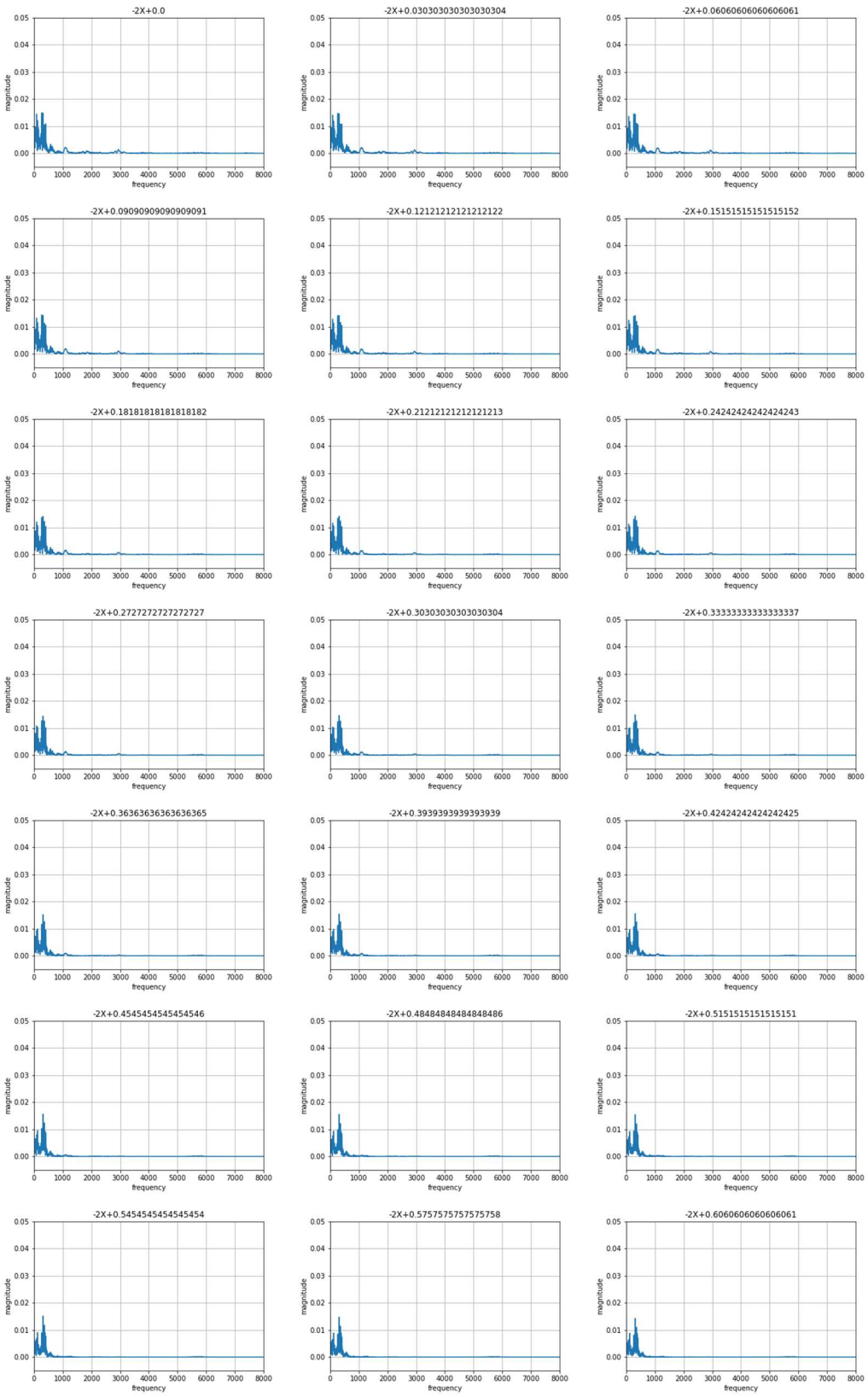

-3*sin(8.4848484848484*x)
-3*sin(8.6868686868687*x)
-3*sin(8.888888888889*x)
-3*sin(9.090909090909*x)
-3*sin(9.292929292929292*x)
-3*sin(9.494949494949495*x)
-3*sin(9.696969696969697*x)
-3*sin(9.8989898989899*x)
-3*sin(10.1010101010101*x)
-3*sin(10.303030303030303*x)
-3*sin(10.505050505050505*x)
-3*sin(10.707070707070708*x)
-3*sin(10.909090909090908*x)
-3*sin(11.111111111111111*x)
-3*sin(11.313131313131313*x)
-3*sin(11.515151515151516*x)
-3*sin(11.717171717171716*x)
-3*sin(11.919191919191919*x)
-3*sin(12.121212121212121*x)
-3*sin(12.323232323232324*x)
-3*sin(12.525252525252524*x)
magnitude
frequency

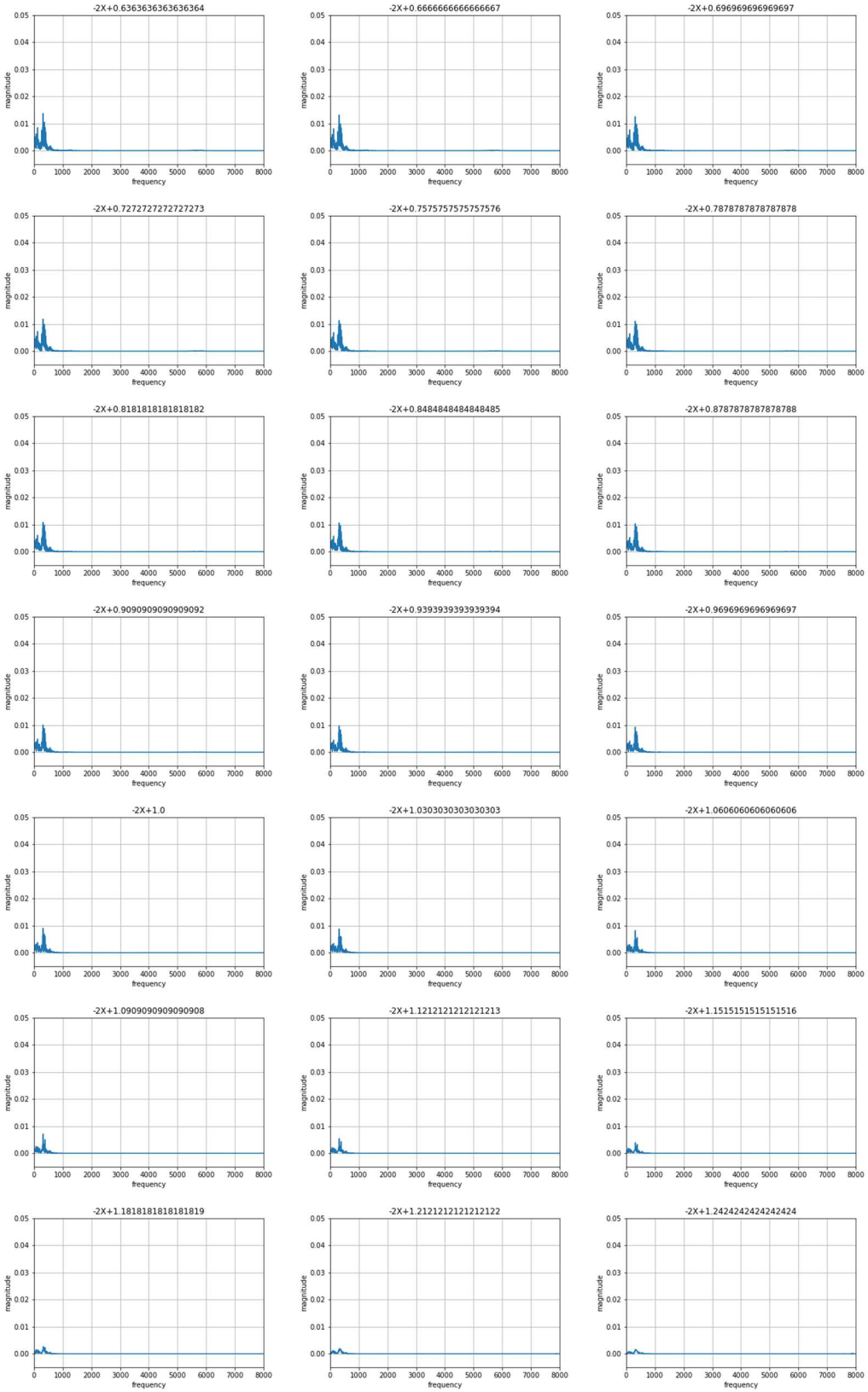

-3*sin(12.7272727272727*x)
-3*sin(12.929292929292929*x)
-3*sin(13.1313131313131*x)
-3*sin(13.333333333333334*x)
-3*sin(13.535353535353535*x)
-3*sin(13.737373737373737*x)
-3*sin(13.93939393939394*x)
-3*sin(14.141414141414142*x)
-3*sin(14.343434343434343*x)
-3*sin(14.545454545454545*x)
-3*sin(14.747474747474747*x)
-3*sin(14.949494949494495*x)
-3*sin(15.151515151515515*x)
-3*sin(15.353535353535353*x)
-3*sin(15.5555555555555*x)
-3*sin(15.757575757575758*x)
-3*sin(15.959595959595958*x)
-3*sin(16.161616161616163*x)
-3*sin(16.363636363636363*x)
-3*sin(16.565656565656564*x)
-3*sin(16.767676767676768*x)
magnitude
frequency

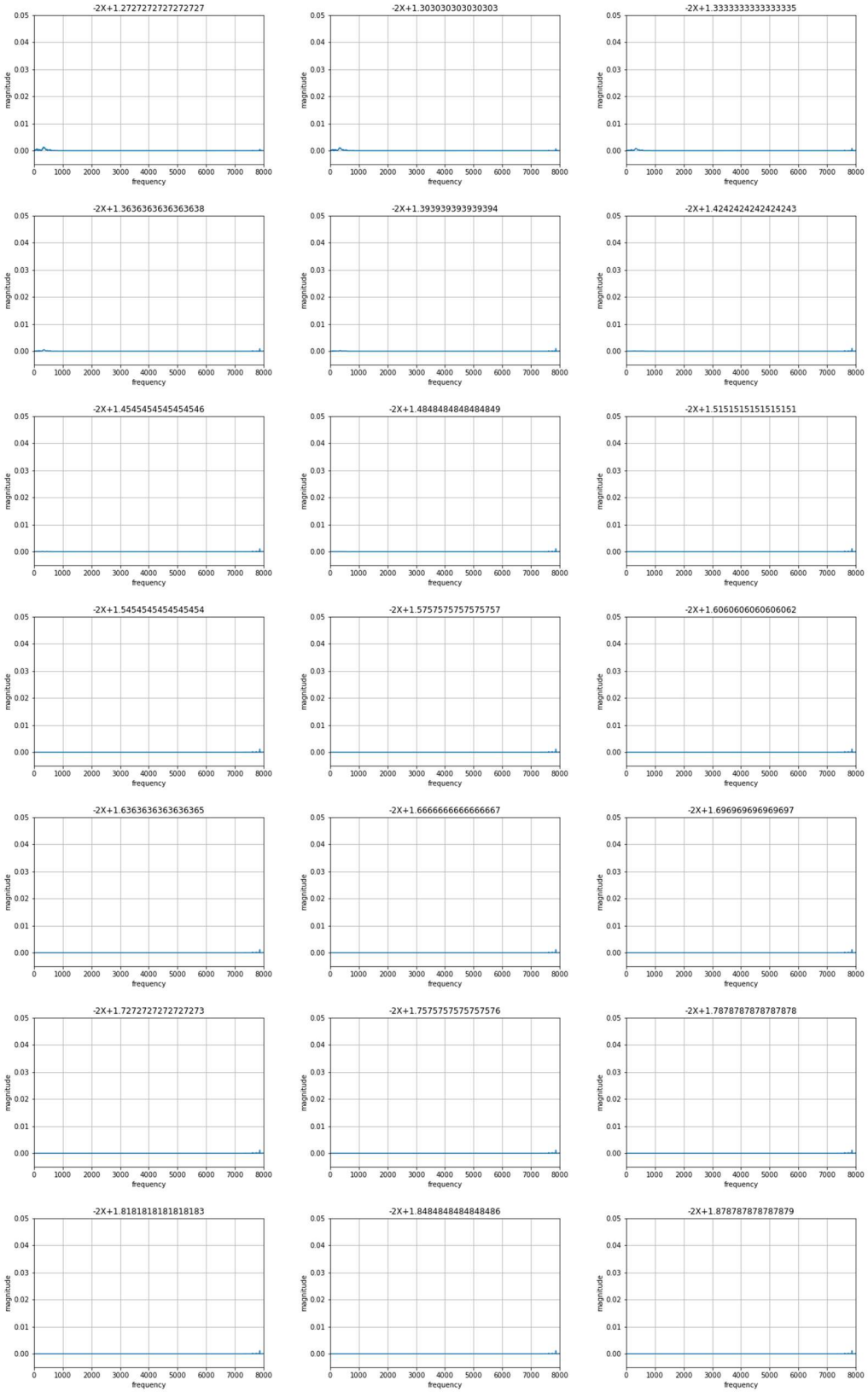


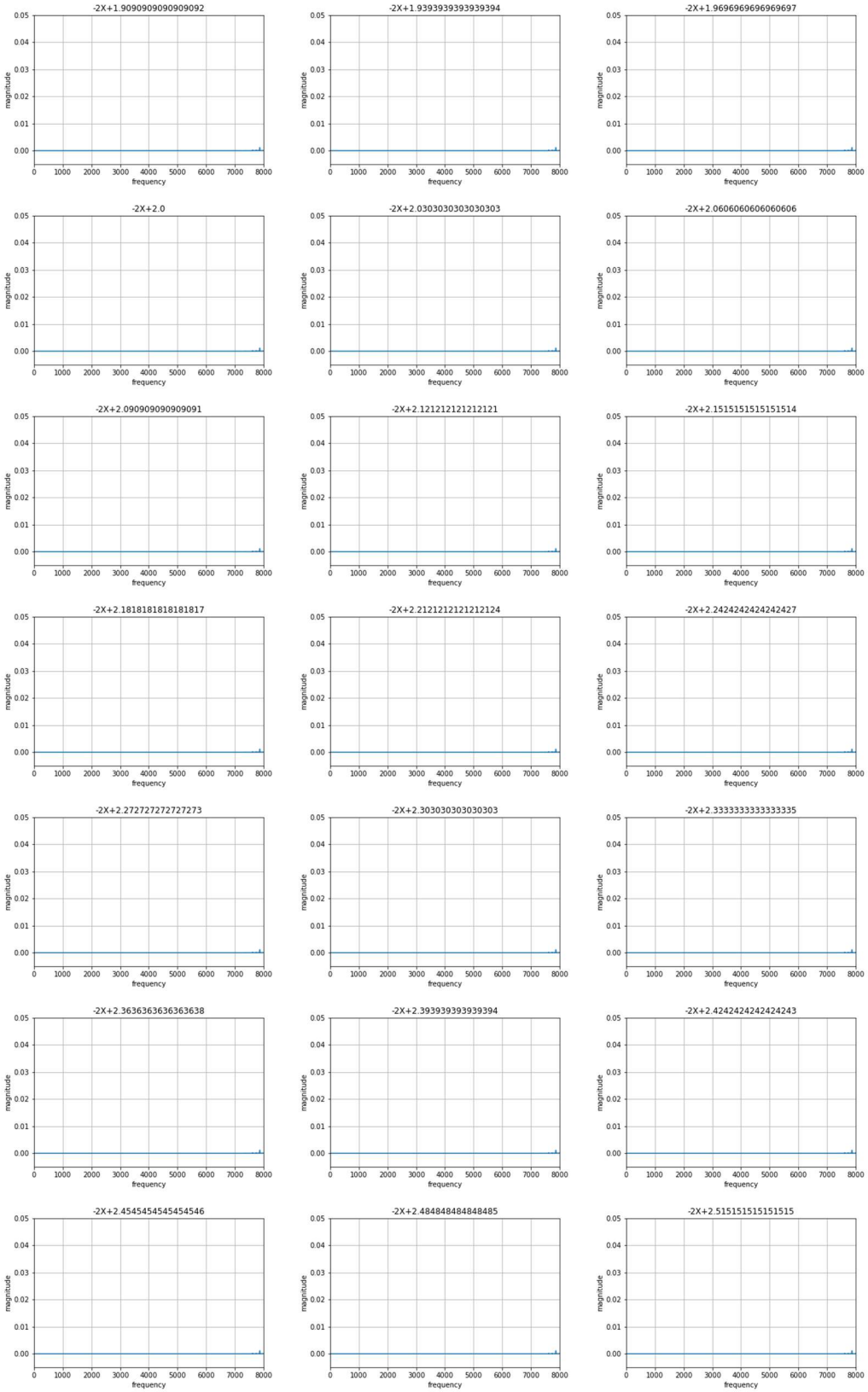

## 7.5.2 Linear functions

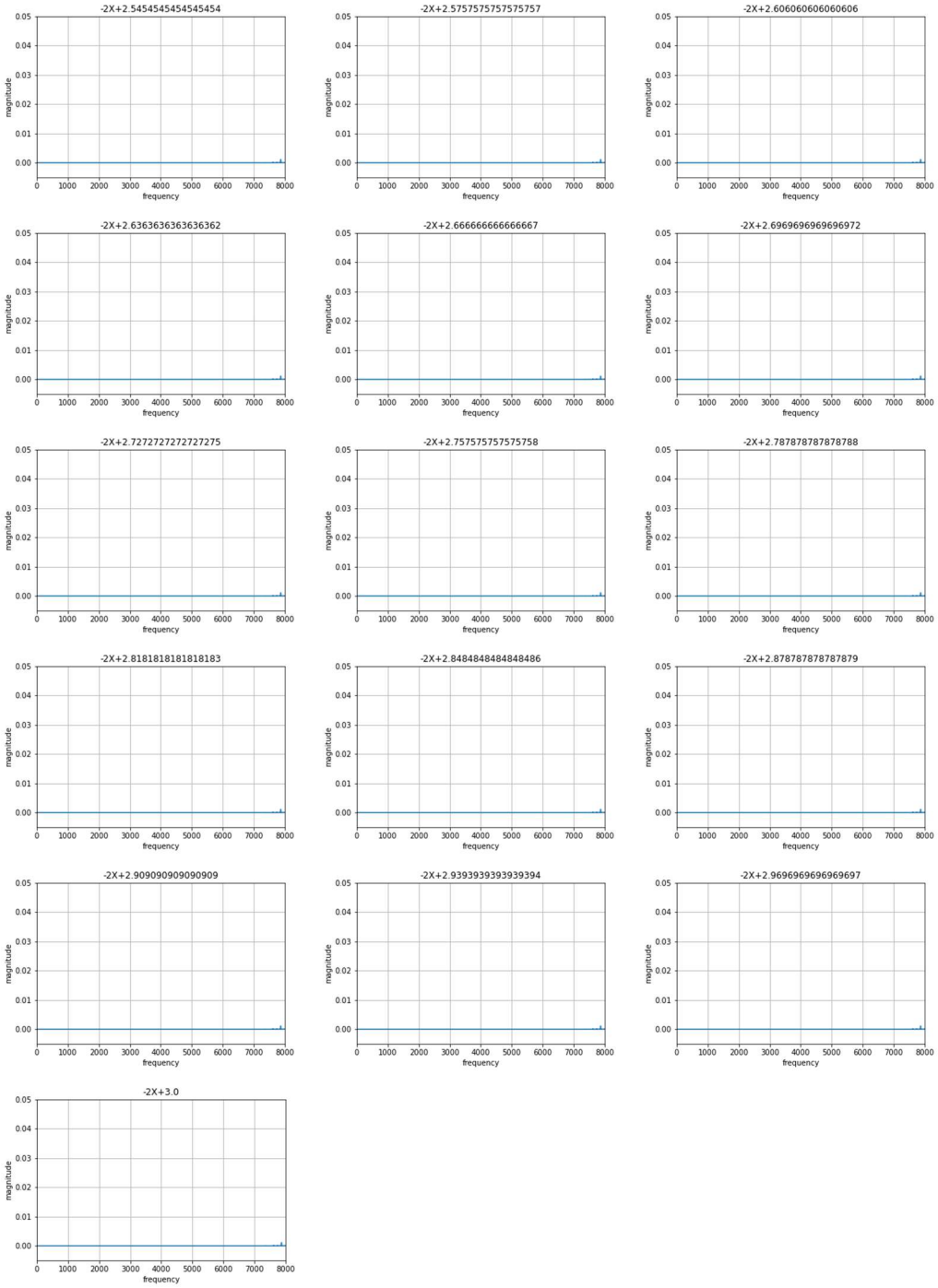
### *7.5.2.1 F (x) = -2x + m for m = [0,3] in steps of 0.03*


-2X+0.0
-2X+0.030303030303030304
-2X+0.06060606060606061
-2X+0.09090909090909091
-2X+0.12121212121212122
-2X+0.15151515151515152
-2X+0.18181818181818182
-2X+0.21212121212121213
-2X+0.24242424242424243
-2X+0.2727272727272727
-2X+0.30303030303030304
-2X+0.33333333333333337
-2X+0.36363636363636365
-2X+0.3939393939393939
-2X+0.42424242424242425
-2X+0.4545454545454546
-2X+0.48484848484848486
-2X+0.5151515151515151
-2X+0.5454545454545454
-2X+0.5757575757575758
-2X+0.6060606060606061
magnitude
frequency

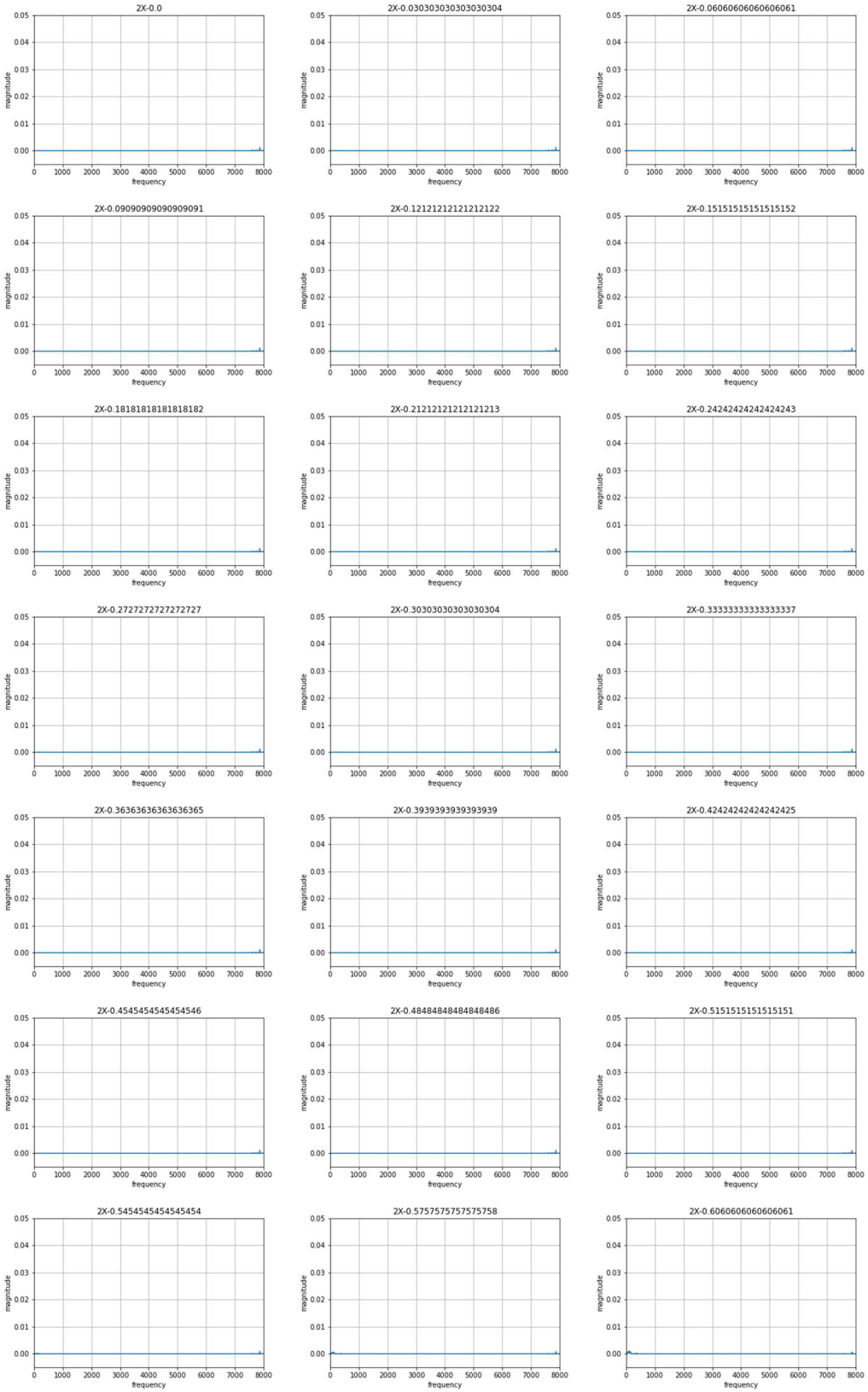

-2X+0.6363636363636364
-2X+0.6666666666666667
-2X+0.6969696969696697
-2X+0.7272727272727273
-2X+0.7575757575757576
-2X+0.7878787878787878
-2X+0.8181818181818182
-2X+0.8484848484848485
-2X+0.8787878787878788
-2X+0.9090909090909092
-2X+0.9393939393939394
-2X+0.9696969696969697
-2X+1.0
-2X+1.0303030303030303
-2X+1.0606060606060606
-2X+1.0909090909090908
-2X+1.1212121212121213
-2X+1.1515151515151516
-2X+1.1818181818181819
-2X+1.2121212121212122
-2X+1.2424242424242424
magnitude
frequency

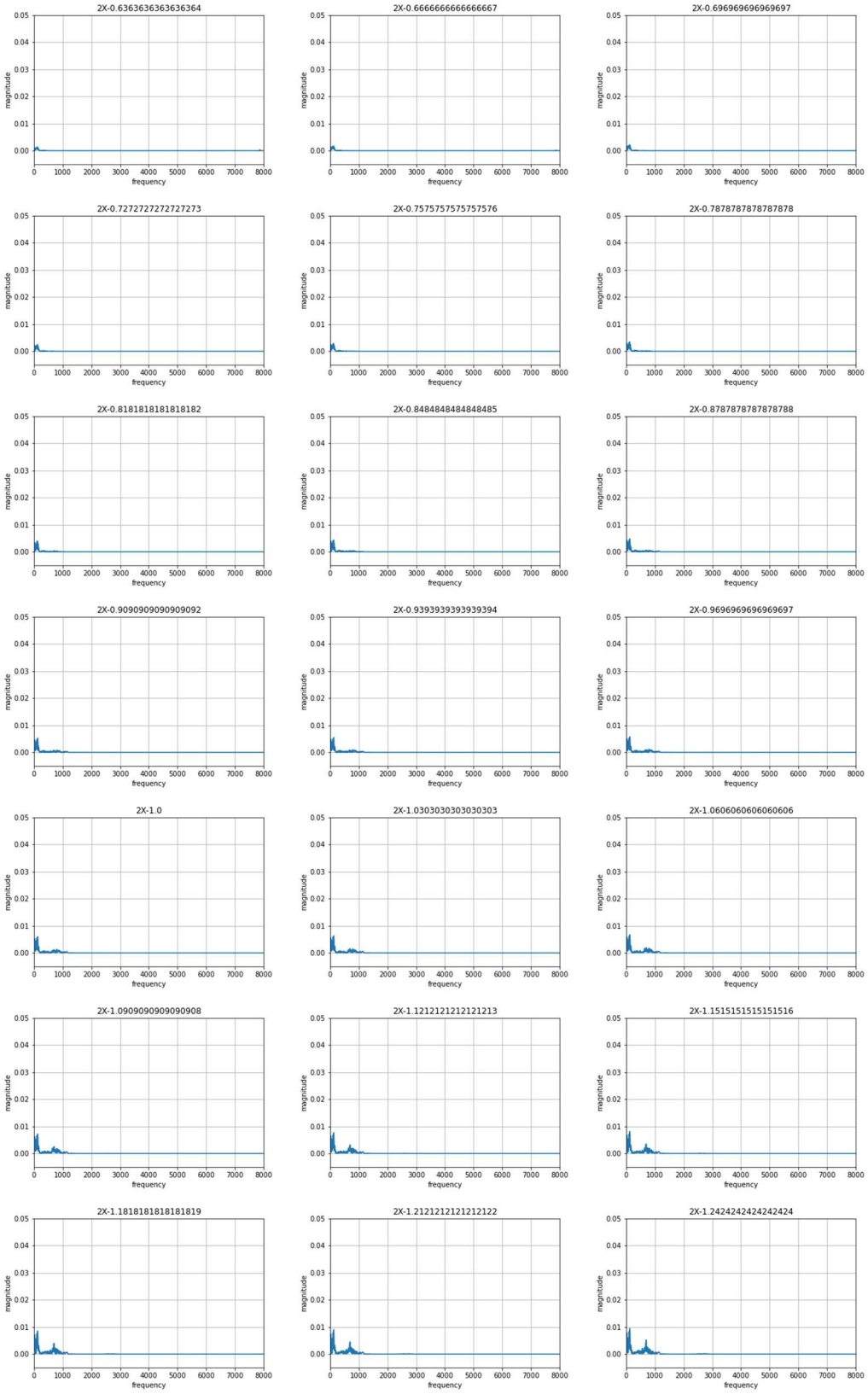

-2X+1.2727272727272727
-2X+1.303030303030303
-2X+1.3333333333333335
-2X+1.3636363636363638
-2X+1.393939393939394
-2X+1.4242424242424243
-2X+1.4545454545454546
-2X+1.4848484848484849
-2X+1.5151515151515151
-2X+1.5454545454545454
-2X+1.5757575757575757
-2X+1.6060606060606062
-2X+1.6363636363636365
-2X+1.6666666666666667
-2X+1.696969696969697
-2X+1.7272727272727273
-2X+1.7575757575757576
-2X+1.7878787878787878
-2X+1.8181818181818183
-2X+1.8484848484848486
-2X+1.878787878787879
magnitude
frequency

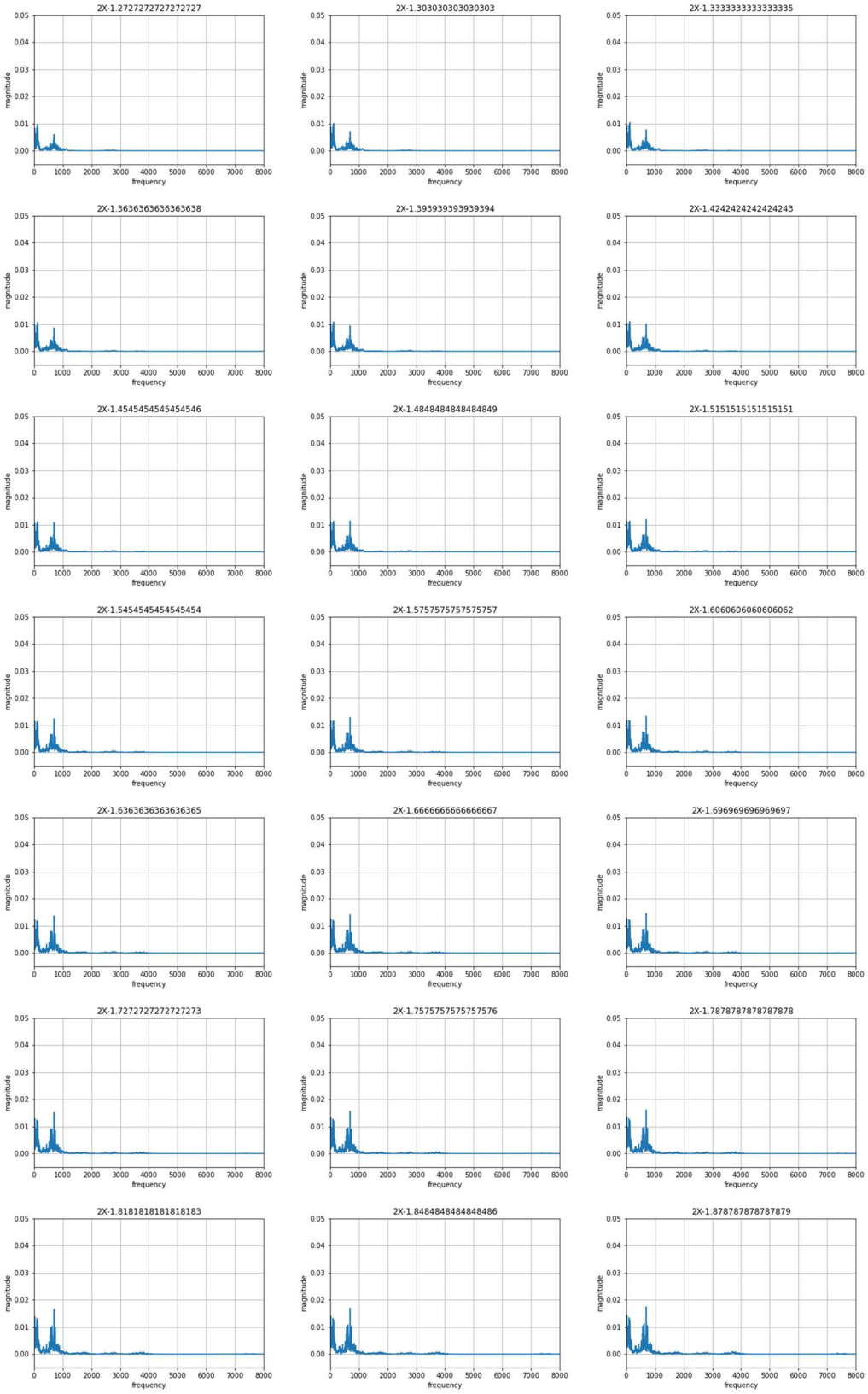

-2X+1.9090909090909092
-2X+1.9393939393939394
-2X+1.9696969696969697
-2X+2.0
-2X+2.0303030303030303
-2X+2.0606060606060606
-2X+2.090909090909091
-2X+2.121212121212121
-2X+2.1515151515151514
-2X+2.1818181818181817
-2X+2.2121212121212124
-2X+2.2424242424242427
-2X+2.272727272727273
-2X+2.303030303030303
-2X+2.3333333333333335
-2X+2.3636363636363638
-2X+2.393939393939394
-2X+2.4242424242424243
-2X+2.4545454545454546
-2X+2.484848484848485
-2X+2.515151515151515
magnitude
frequency

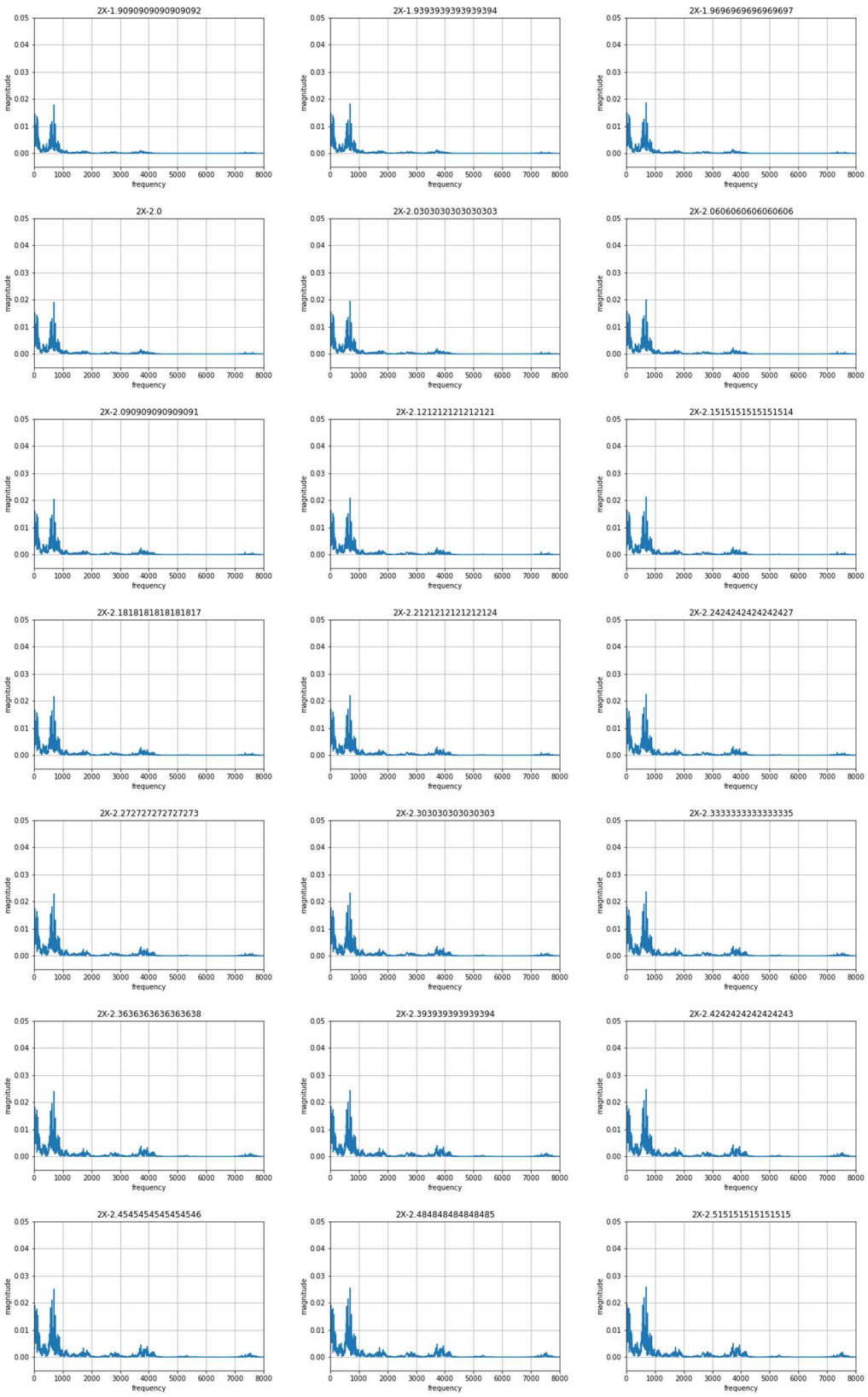


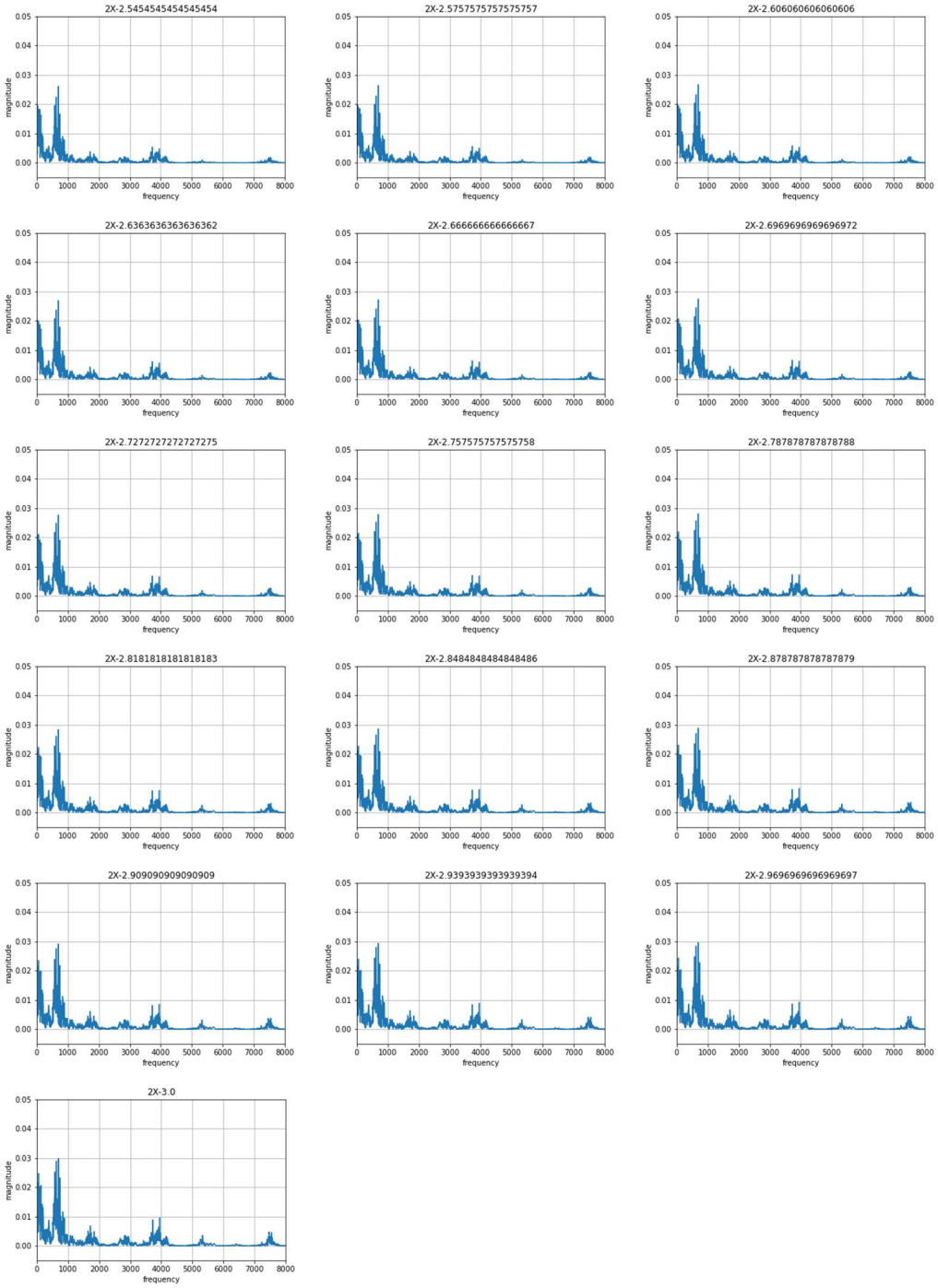

### *7.5.2.2 F(x) = 2x – m for m = [0,3] in steps of 0.03*


2X-0.0
2X-0.030303030303030304
2X-0.06060606060606061
2X-0.09090909090909091
2X-0.121212121212122
2X-0.15151515151515152
2X-0.18181818181818182
2X-0.21212121212121213
2X-0.24242424242424243
2X-0.2727272727272727
2X-0.30303030303030304
2X-0.33333333333333337
2X-0.36363636363636365
2X-0.3939393939393939
2X-0.42424242424242425
2X-0.4545454545454546
2X-0.48484848484848486
2X-0.5151515151515151
2X-0.5454545454545454
2X-0.5757575757575758
2X-0.6060606060606061
magnitude
frequency

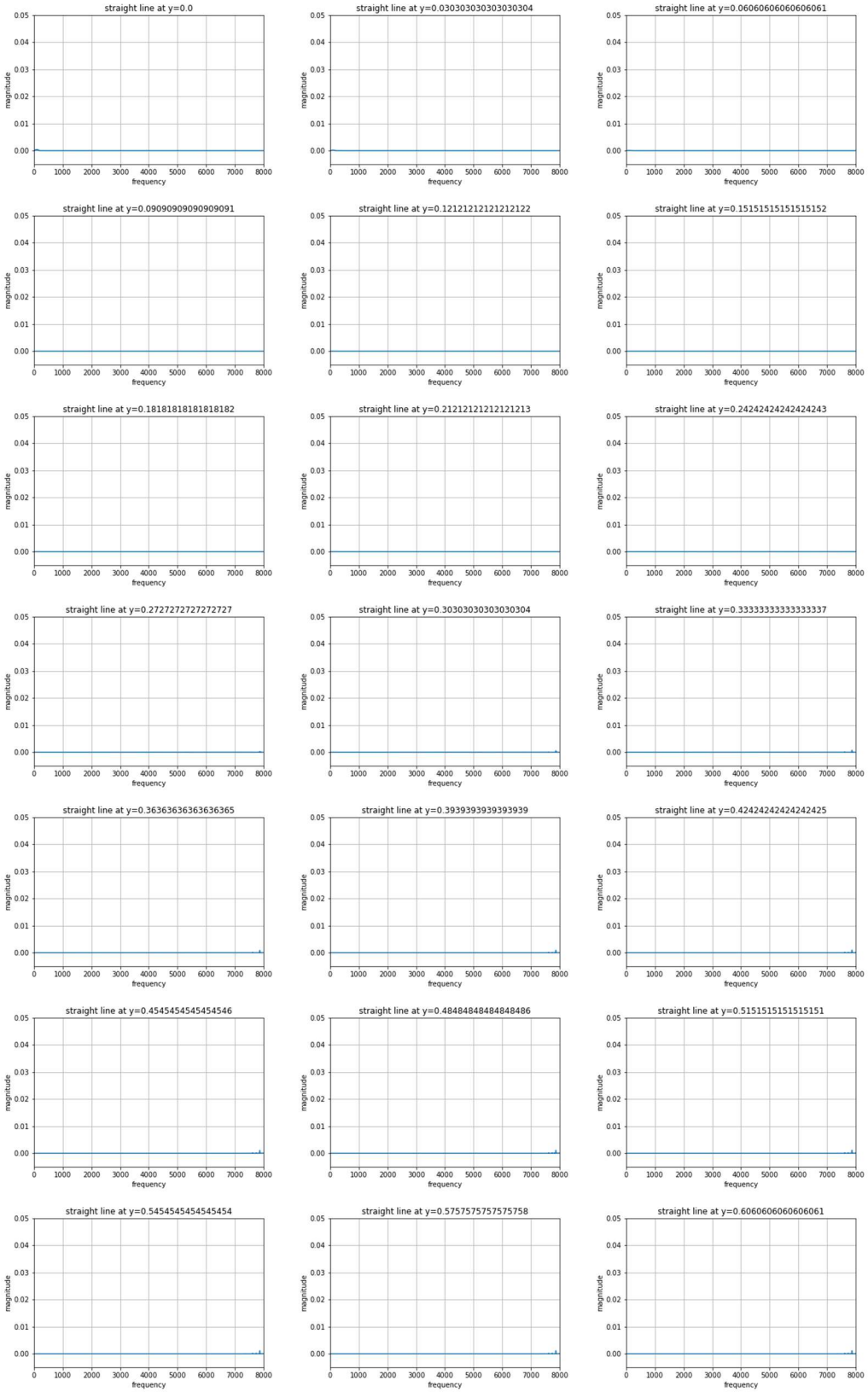

2X-0.6363636363636364
2X-0.6666666666666667
2X-0.6969696969696697
2X-0.7272727272727273
2X-0.7575757575757576
2X-0.7878787878787878
2X-0.8181818181818182
2X-0.8484848484848485
2X-0.8787878787878788
2X-0.9090909090909092
2X-0.9393939393939394
2X-0.9696969696969697
2X-1.0
2X-1.0303030303030303
2X-1.0606060606060606
2X-1.0909090909090908
2X-1.1212121212121213
2X-1.1515151515151516
2X-1.1818181818181819
2X-1.2121212121212122
2X-1.2424242424242424
magnitude
frequency

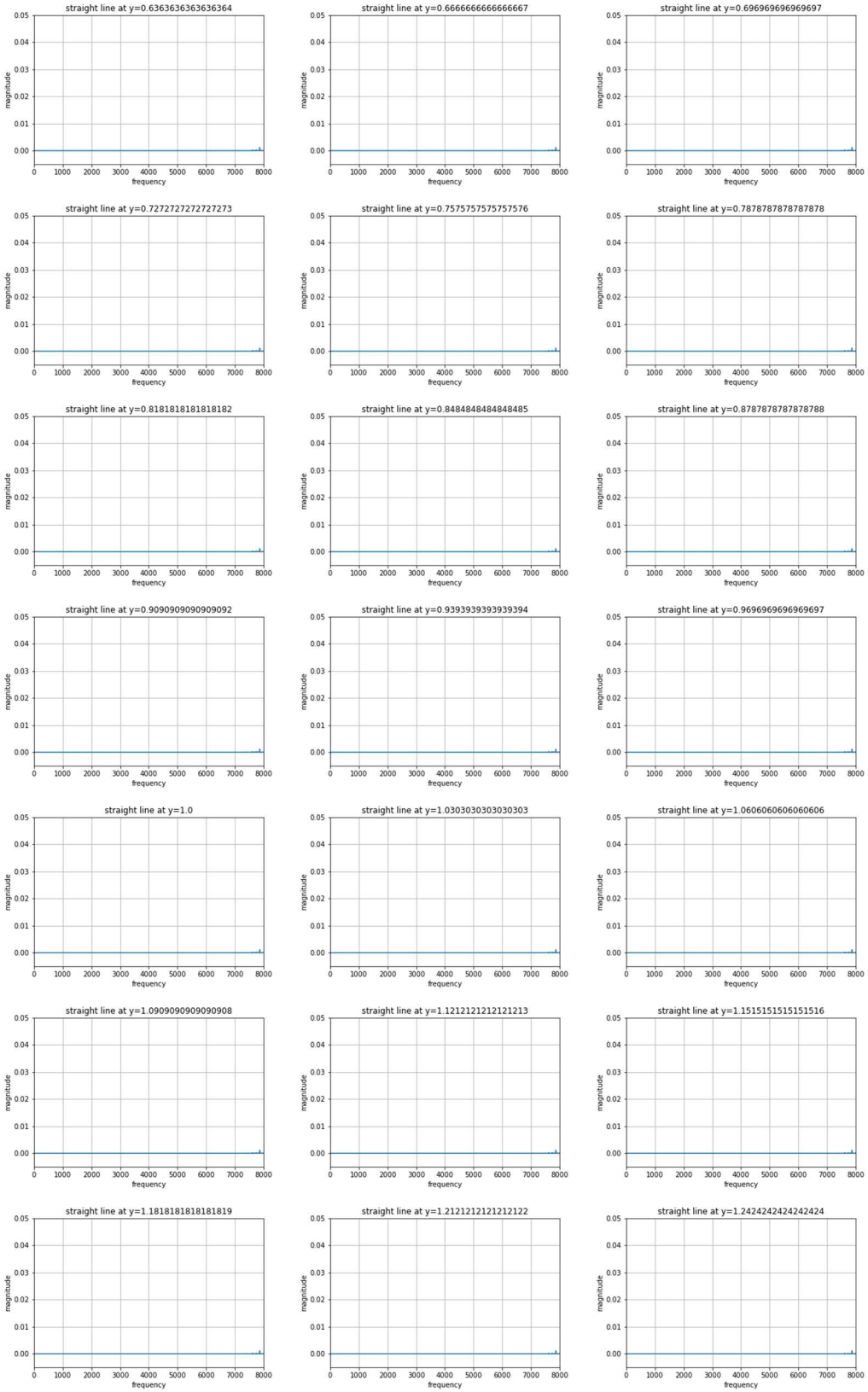

2X-1.27272727272727
2X-1.303030303030303
2X-1.3333333333333335
2X-1.3636363636363638
2X-1.393939393939394
2X-1.4242424242424243
2X-1.4545454545454546
2X-1.4848484848484849
2X-1.5151515151515151
2X-1.5454545454545454
2X-1.575757575757577
2X-1.6060606060606062
2X-1.6363636363636365
2X-1.6666666666666667
2X-1.696969696969697
2X-1.7272727272727273
2X-1.7575757575757576
2X-1.7878787878787878
2X-1.8181818181818183
2X-1.8484848484848486
2X-1.878787878787879
magnitude
frequency

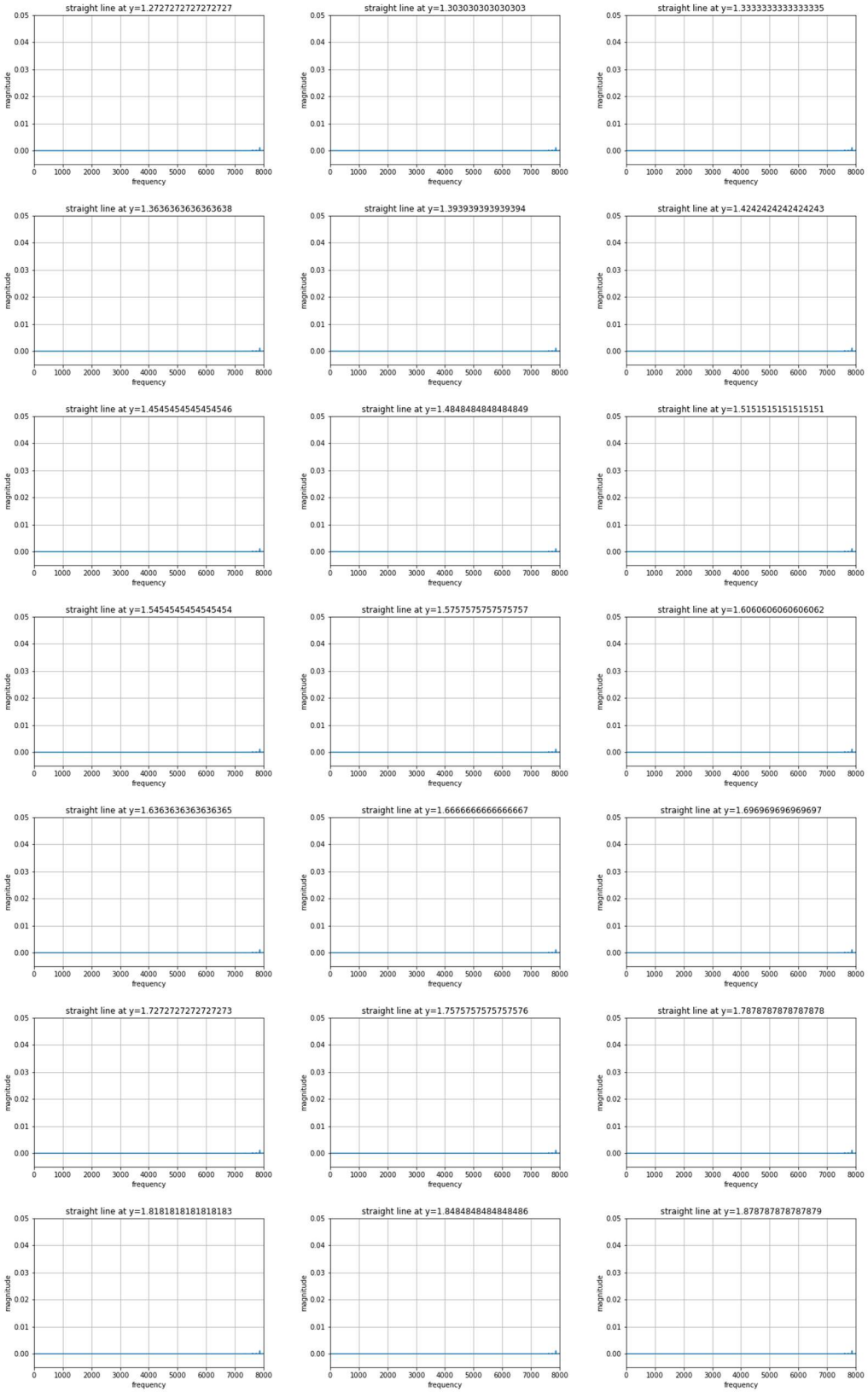

2X-1.9090909090909092
2X-1.9393939393939394
2X-1.9696969696969697
2X-2.0
2X-2.0303030303030303
2X-2.0606060606060606
2X-2.090909090909091
2X-2.121212121212121
2X-2.1515151515151514
2X-2.1818181818181817
2X-2.2121212121212124
2X-2.2424242424242427
2X-2.272727272727273
2X-2.303030303030303
2X-2.3333333333333335
2X-2.3636363636363638
2X-2.393939393939394
2X-2.4242424242424243
2X-2.4545454545454546
2X-2.484848484848485
2X-2.515151515151515
magnitude
frequency

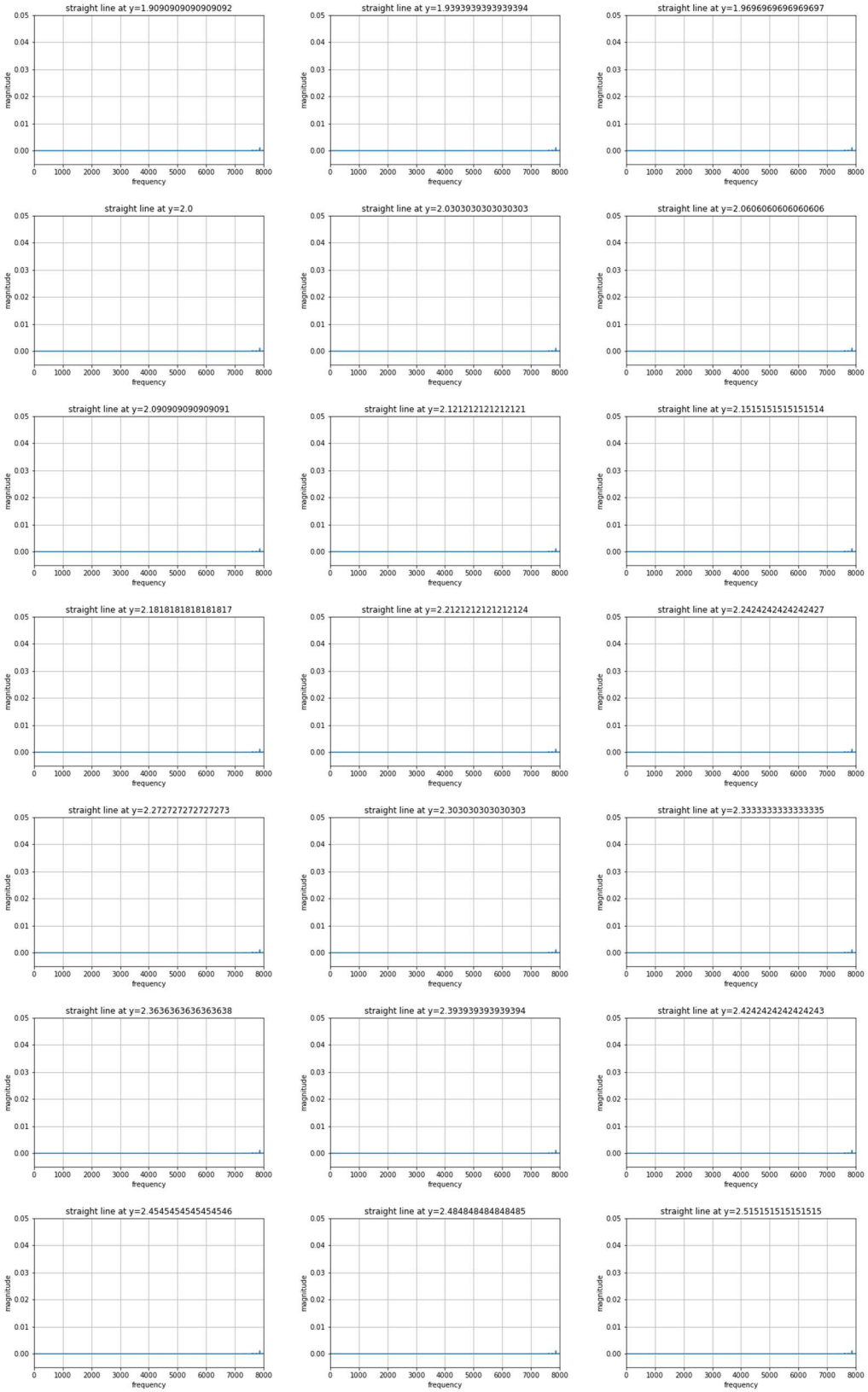


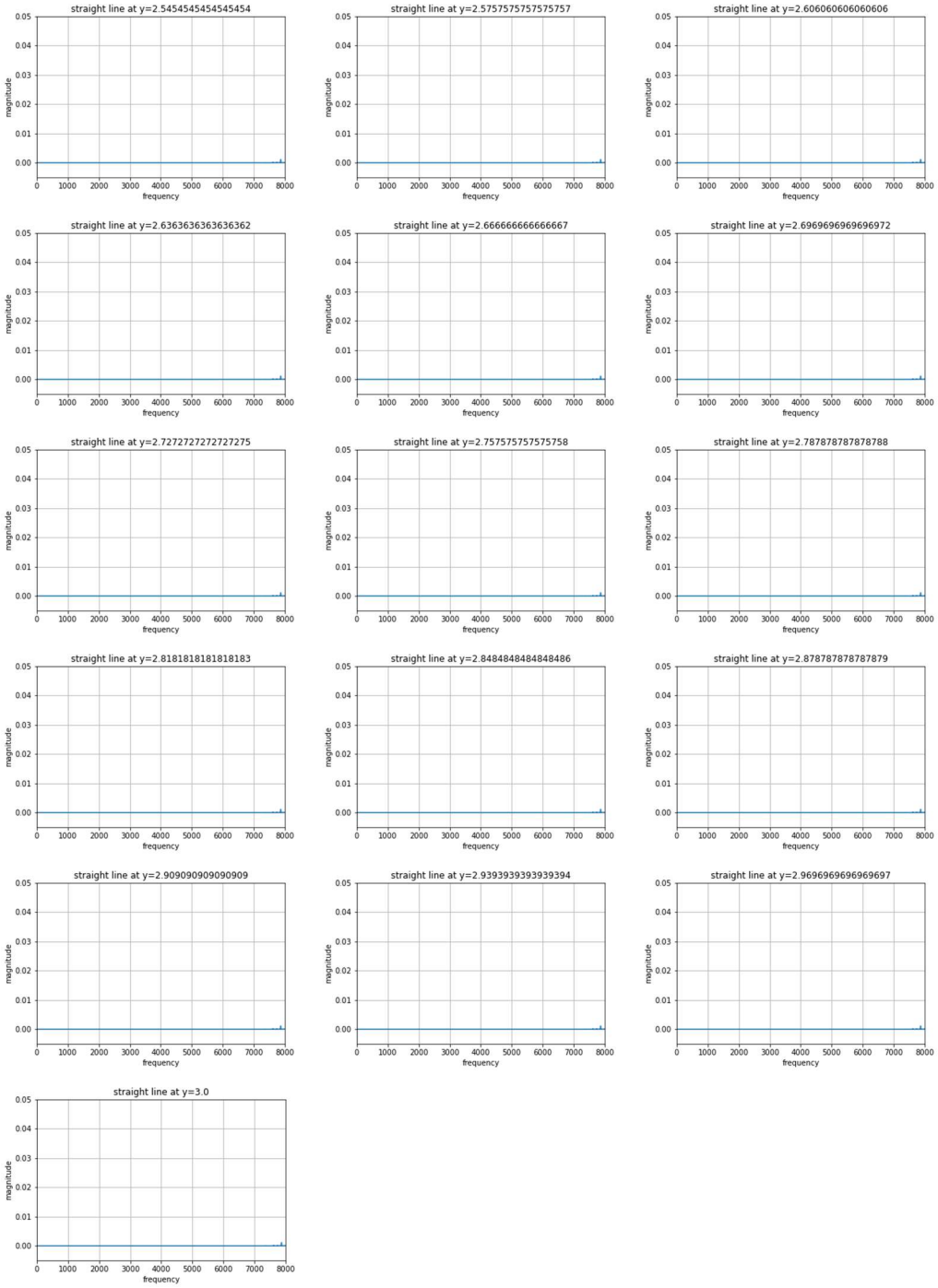

### *7.5.2.3 F(x) = G for G = [0,3] in steps of 0.03*


straight line at y=0.0
straight line at y=0.030303030303030304
straight line at y=0.06060606060606061
straight line at y=0.09090909090909091
straight line at y=0.12121212121212122
straight line at y=0.15151515151515152
straight line at y=0.18181818181818182
straight line at y=0.21212121212121213
straight line at y=0.24242424242424243
straight line at y=0.2727272727272727
straight line at y=0.30303030303030304
straight line at y=0.33333333333333337
straight line at y=0.36363636363636365
straight line at y=0.3939393939393939
straight line at y=0.42424242424242425
straight line at y=0.4545454545454546
straight line at y=0.48484848484848486
straight line at y=0.5151515151515151
straight line at y=0.5454545454545454
straight line at y=0.5757575757575758
straight line at y=0.6060606060606061
magnitude
frequency

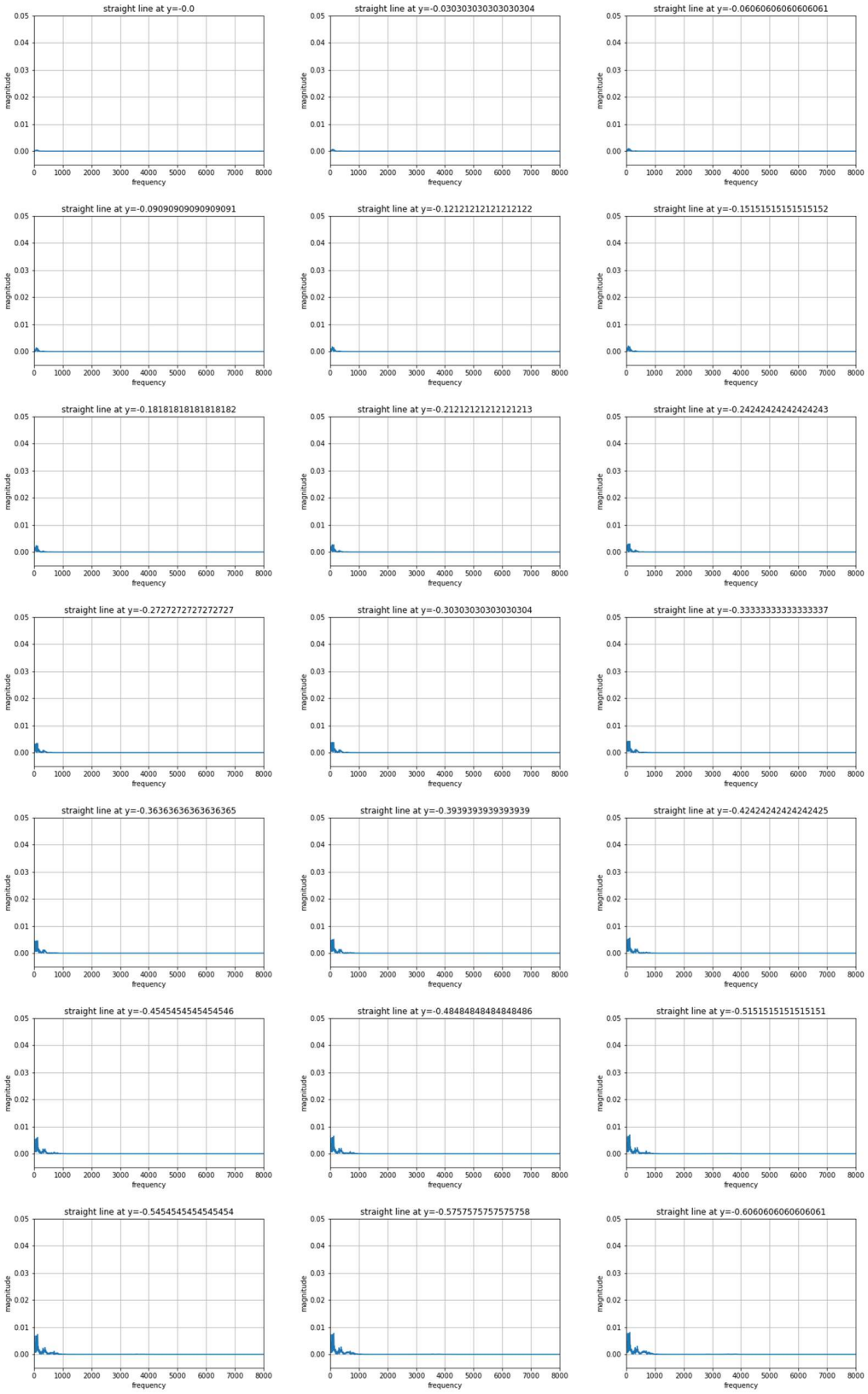

straight line at y=0.6363636363636364
straight line at y=0.6666666666666667
straight line at y=0.6969696969696697
straight line at y=0.7272727272727273
straight line at y=0.7575757575757576
straight line at y=0.7878787878787878
straight line at y=0.8181818181818182
straight line at y=0.8484848484848485
straight line at y=0.8787878787878788
straight line at y=0.9090909090909092
straight line at y=0.9393939393939394
straight line at y=0.9696969696969697
straight line at y=1.0
straight line at y=1.0303030303030303
straight line at y=1.0606060606060606
straight line at y=1.0909090909090908
straight line at y=1.1212121212121213
straight line at y=1.1515151515151516
straight line at y=1.1818181818181819
straight line at y=1.2121212121212122
straight line at y=1.2424242424242424
magnitude
frequency

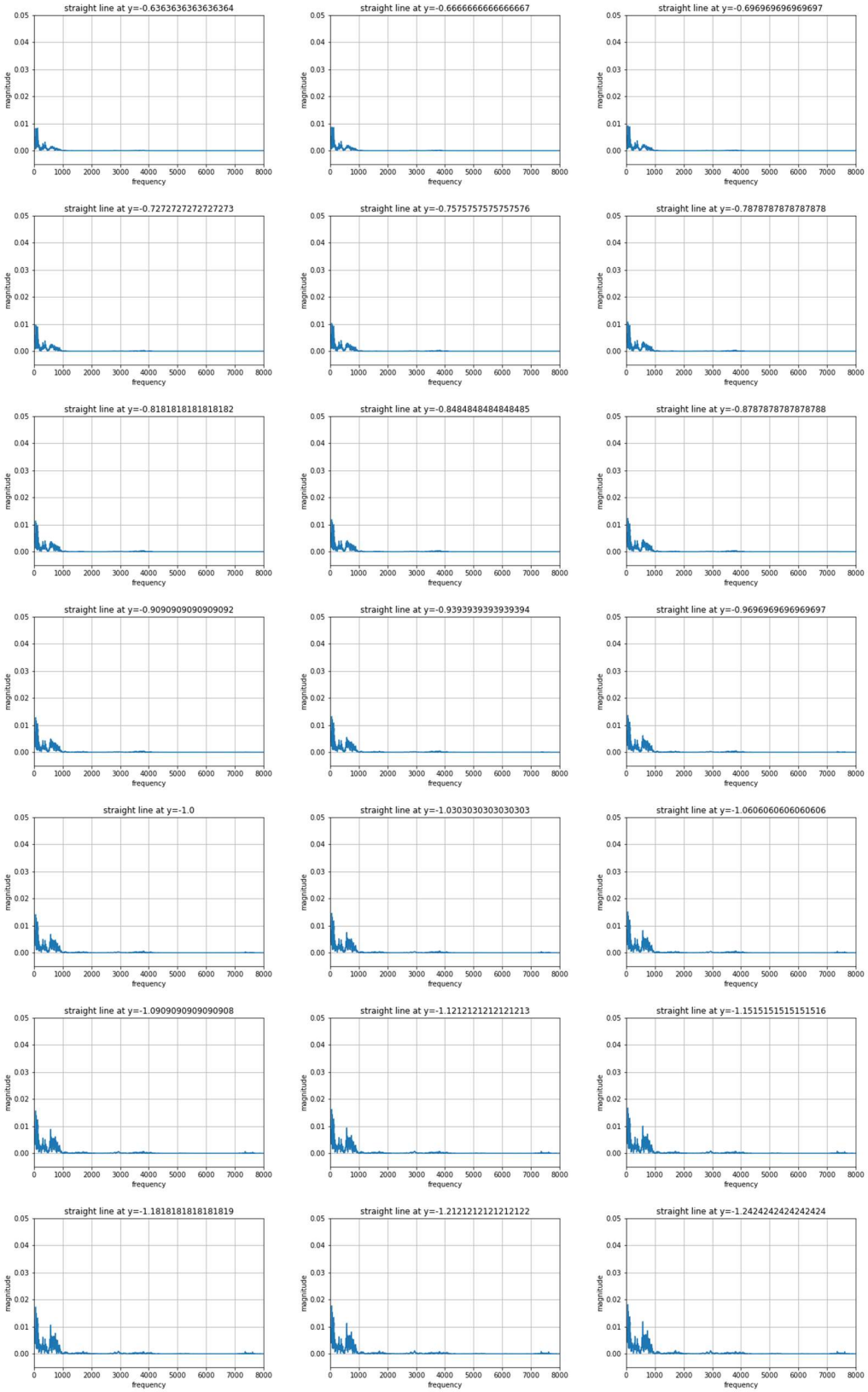

straight line at y=1.2727272727272727
straight line at y=1.303030303030303
straight line at y=1.3333333333333335
straight line at y=1.3636363636363638
straight line at y=1.393939393939394
straight line at y=1.4242424242424243
straight line at y=1.4545454545454546
straight line at y=1.4848484848484849
straight line at y=1.5151515151515151
straight line at y=1.5454545454545454
straight line at y=1.5757575757575757
straight line at y=1.6060606060606062
straight line at y=1.6363636363636365
straight line at y=1.6666666666666667
straight line at y=1.696969696969697
straight line at y=1.7272727272727273
straight line at y=1.7575757575757576
straight line at y=1.7878787878787878
straight line at y=1.8181818181818183
straight line at y=1.8484848484848486
straight line at y=1.878787878787879
magnitude
frequency

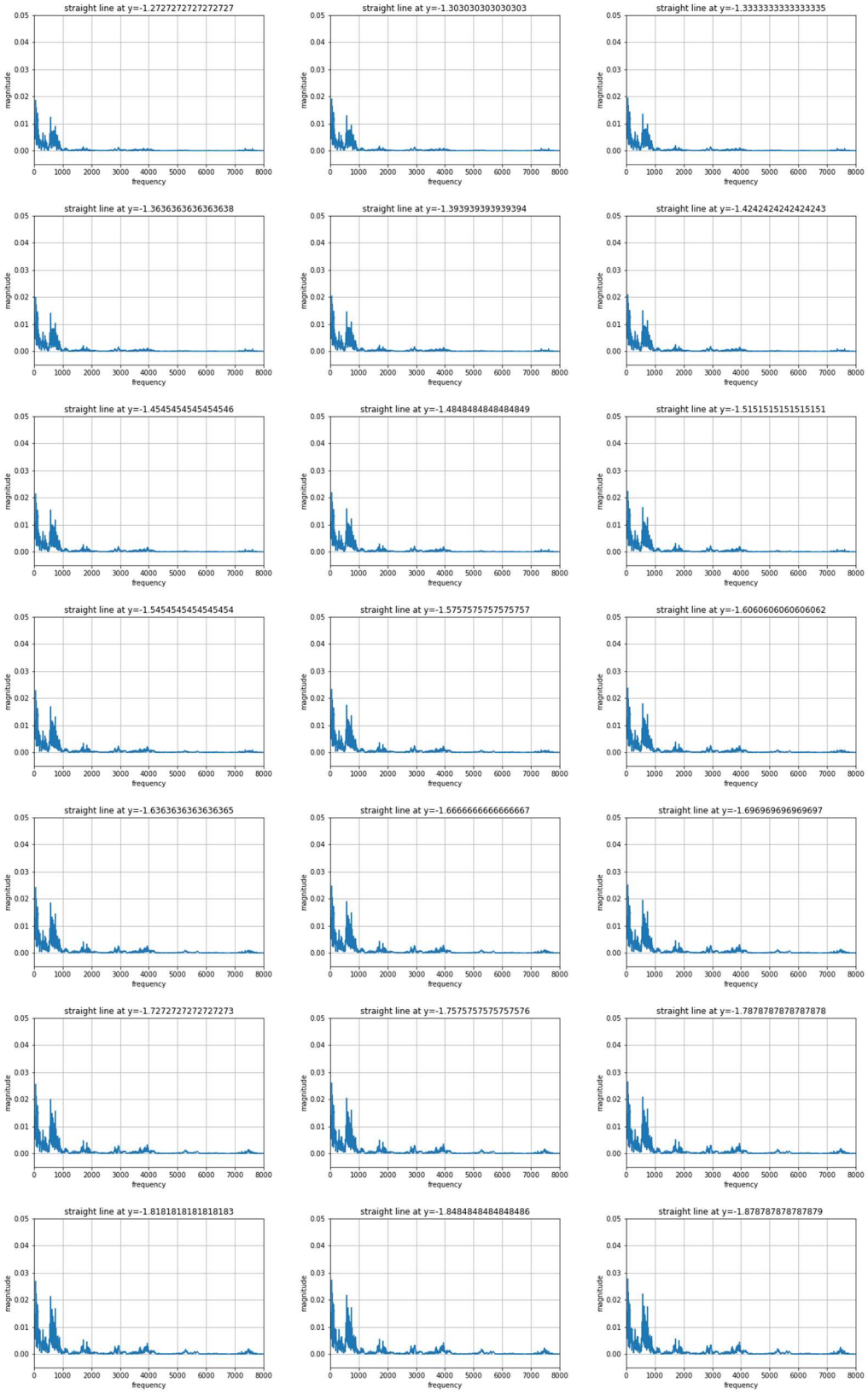

straight line at y=1.9090909090909092
straight line at y=1.9393939393939394
straight line at y=1.9696969696969697
straight line at y=2.0
straight line at y=2.0303030303030303
straight line at y=2.0606060606060606
straight line at y=2.090909090909091
straight line at y=2.121212121212121
straight line at y=2.1515151515151514
straight line at y=2.1818181818181817
straight line at y=2.2121212121212124
straight line at y=2.2424242424242427
straight line at y=2.272727272727273
straight line at y=2.303030303030303
straight line at y=2.3333333333333335
straight line at y=2.3636363636363638
straight line at y=2.393939393939394
straight line at y=2.4242424242424243
straight line at y=2.4545454545454546
straight line at y=2.484848484848485
straight line at y=2.515151515151515
magnitude
frequency

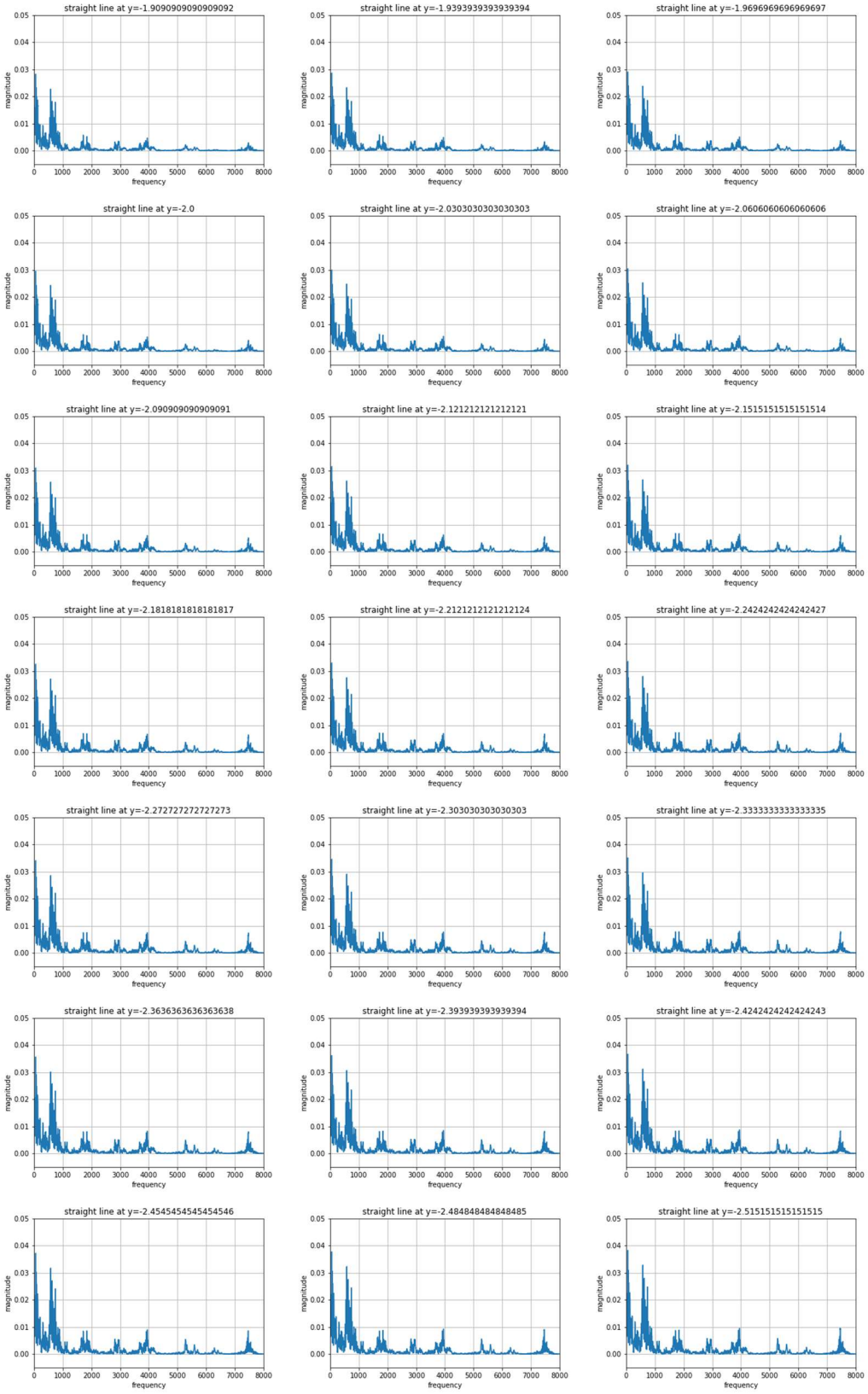

straight line at y=2.5454545454545454
straight line at y=2.5757575757575757
straight line at y=2.606060606060606
straight line at y=2.6363636363636362
straight line at y=2.666666666666667
straight line at y=2.6969696969696972
straight line at y=2.7272727272727275
straight line at y=2.757575757575758
straight line at y=2.787878787878788
straight line at y=2.8181818181818183
straight line at y=2.8484848484848486
straight line at y=2.878787878787879
straight line at y=2.909090909090909
straight line at y=2.9393939393939394
straight line at y=2.9696969696969697
straight line at y=3.0
magnitude
frequency

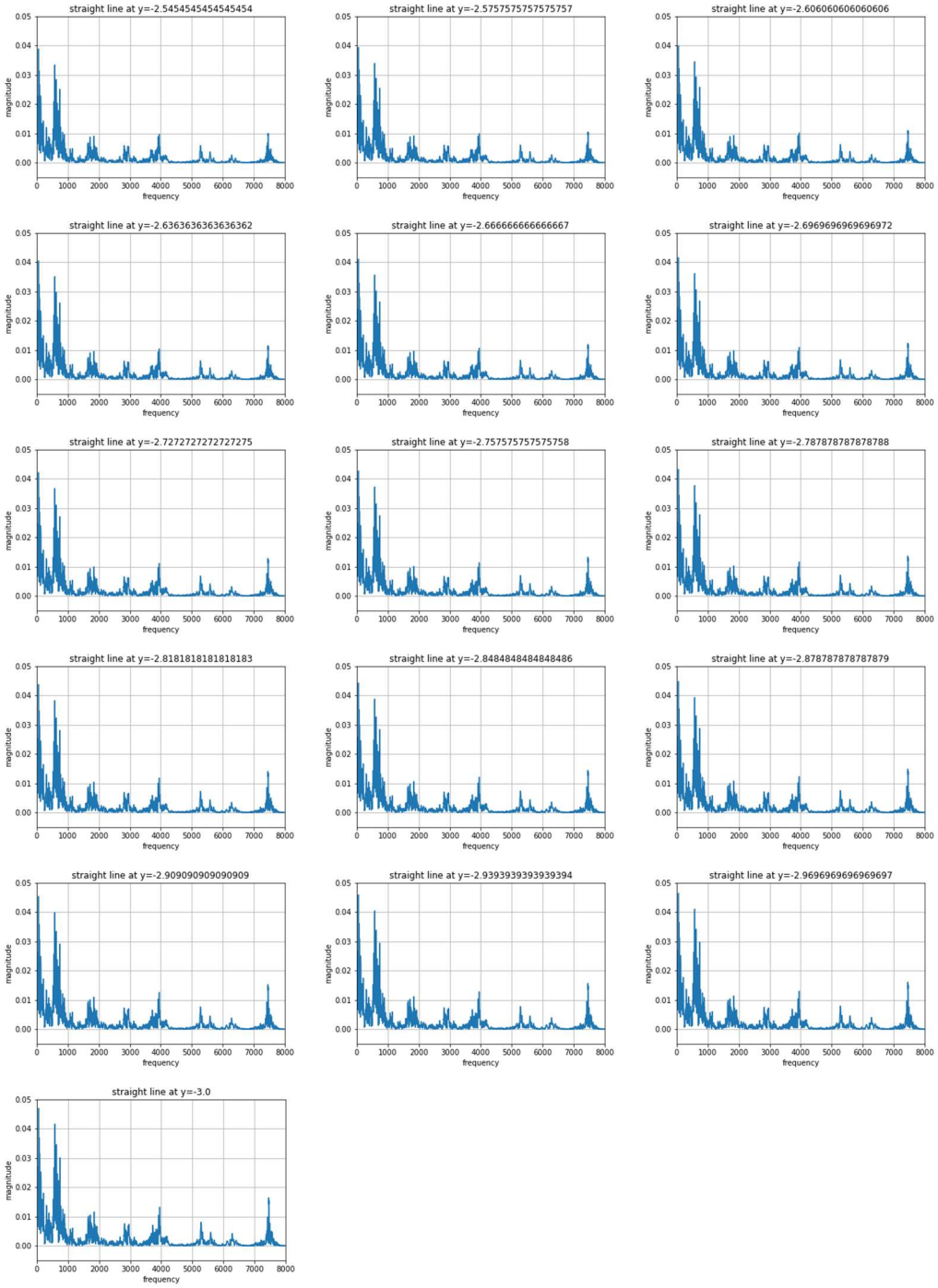
### 7.5.2.4 F(x) = -G for G = [0,3] in steps of 0.03




straight line at y=-0.6363636363636364
straight line at y=-0.6666666666666667
straight line at y=-0.696969696969697
straight line at y=-0.7272727272727273
straight line at y=-0.7575757575757576
straight line at y=-0.7878787878787878
straight line at y=-0.8181818181818182
straight line at y=-0.8484848484848485
straight line at y=-0.8787878787878788
straight line at y=-0.9090909090909092
straight line at y=-0.9393939393939394
straight line at y=-0.9696969696969697
straight line at y=-1.0
straight line at y=-1.0303030303030303
straight line at y=-1.0606060606060606
straight line at y=-1.0909090909090908
straight line at y=-1.1212121212121213
straight line at y=-1.1515151515151516
straight line at y=-1.1818181818181819
straight line at y=-1.2121212121212122
straight line at y=-1.2424242424242424
magnitude
frequency


straight line at y=-1.2727272727272727
straight line at y=-1.303030303030303
straight line at y=-1.3333333333333335
straight line at y=-1.3636363636363638
straight line at y=-1.393939393939394
straight line at y=-1.4242424242424243
straight line at y=-1.4545454545454546
straight line at y=-1.4848484848484849
straight line at y=-1.5151515151515151
straight line at y=-1.5454545454545454
straight line at y=-1.5757575757575757
straight line at y=-1.6060606060606062
straight line at y=-1.6363636363636365
straight line at y=-1.6666666666666667
straight line at y=-1.696969696969697
straight line at y=-1.7272727272727273
straight line at y=-1.7575757575757576
straight line at y=-1.7878787878787878
straight line at y=-1.8181818181818183
straight line at y=-1.8484848484848486
straight line at y=-1.878787878787879
magnitude
frequency


straight line at y=-1.9090909090909092
straight line at y=-1.9393939393939394
straight line at y=-1.9696969696969697
straight line at y=-2.0
straight line at y=-2.0303030303030303
straight line at y=-2.0606060606060606
straight line at y=-2.090909090909091
straight line at y=-2.121212121212121
straight line at y=-2.1515151515151514
straight line at y=-2.1818181818181817
straight line at y=-2.2121212121212124
straight line at y=-2.2424242424242427
straight line at y=-2.272727272727273
straight line at y=-2.303030303030303
straight line at y=-2.3333333333333335
straight line at y=-2.3636363636363638
straight line at y=-2.393939393939394
straight line at y=-2.4242424242424243
straight line at y=-2.4545454545454546
straight line at y=-2.484848484848485
straight line at y=-2.515151515151515
magnitude
frequency


straight line at y=-2.5454545454545454
straight line at y=-2.5757575757575757
straight line at y=-2.606060606060606
straight line at y=-2.6363636363636362
straight line at y=-2.666666666666667
straight line at y=-2.6969696969696972
straight line at y=-2.7272727272727275
straight line at y=-2.757575757575758
straight line at y=-2.787878787878788
straight line at y=-2.8181818181818183
straight line at y=-2.8484848484848486
straight line at y=-2.878787878787879
straight line at y=-2.909090909090909
straight line at y=-2.9393939393939394
straight line at y=-2.9696969696969697
straight line at y=-3.0
magnitude
frequency