PITCHER
Recognizing music pitches exploiting ALSA and neural networks*

student: Ilaria Tono
supervisor: Tommaso Cucinotta

May 2019

Contents

*Developed for Computer Based Software Design

1 Abstract

This project deploys a recognizer of musical pitches, played both by instruments and voice. Commercial and free-ware tuners commonly compare the Fourier Transform (frequency plot) of signal coming into the input device with the one of other known signals. The quality of these applications usually depend on how they manage filters, how they reject noise and on the quality of the recorder and its drivers.

In the field of audio processing and computing, the use of neural networks has become very popular. Both in academic and commercial context, recurrent networks are nowadays the most popular for recognizing speech and music in real-time. Simpler categorization problems can be addressed by other neural architectures too.

We address the realization of a tuner that avoids FFT comparison but exploits the simplest neural architecture, the perceptron, for solving the problem of pitch recognition, which is a categorization task.

2 Outline

2.1 ALSA

The Advanced Linux Sound Architecture (ALSA) is an open-source framework embedded in Linux Kernel. It provides a kernel API and a library API; its functionalities are exploited by other frameworks on a higher level, such as PulseAudio and the JACK audio connection kit.

ALSA arranges hardware audio devices and their components into a hierarchy of *cards*, *devices* and *subdevices*. This offers a map of the hardware through drivers that manage them.

ALSA cards correspond one-to-one to hardware sound cards. In this way, we can list the devices on each card with a string ID or a numerical index. Most of the hardware access done by ALSA is applied on devices, that can be opened and used independently.

Subdevices are the most fine-grained objects ALSA can distinguish and are enumerated as devices. An example is about channels of a device: in this case, the use of more subdevices (say multi-channel signals) implies that all of them need to block for playing.

Digitized sound has a number of parameters such as the sampling rate, the number of channels and the format in which sample values are stored. These parameters have to be set when recording or playing an audio file: the process is called *configuration*. ALSA distinguish between *hardware* and *software* configuration: parameters are not independent and cards can have limited possibilities (not all sampling rates can be applied, for example). ALSA accounts for this fact by arranging sets of parameters in an n-dimensional space, the *configuration space*. One of these dimensions corresponds to the sampling rate, one to the sample format, and so on. If the parameters of one specific sound card are

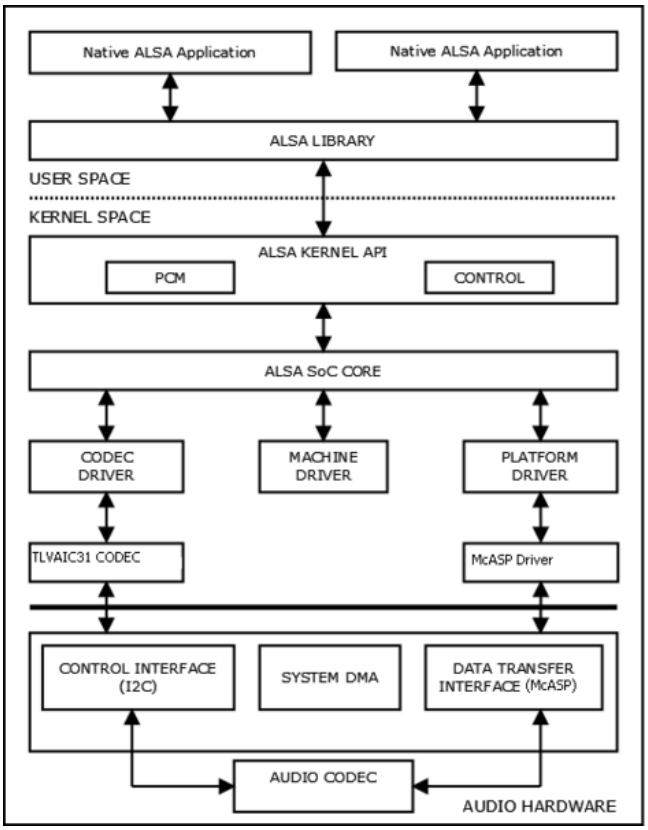


Figure 1: Generic map of ALSA interface between user, kernel and hardware cards

all independent, all legal configurations lie in one big n-dimensional box and we could describe them by giving separate ranges for all parameters. On the contrary, if the parameters are not independent, the architecture varies acquiring complexity.

When a hardware device is accessed with ALSA, parameters are not fixed independently of each other, but the legal configuration space for a device is narrowed down successively by restricting specific parameters. This makes it possible to set some minimal values rather than exact ones. In order to avoid problems and to manage devices more efficiently, ALSA comes with a large set of *plugins*, enlisted in the official documentation. Each plugin is a minimal component that is used to work with ALSA's basic or complex services. One of the most important is `hw`, used for merely access an hardware driver.

2.2 Neural networks: perceptrons

The evolution of biology research in last century has led to the creation of systems that can emulate the behavior of human brain in very different ways. The brain has many special abilities that make it interesting to be applied also to artificial machines:

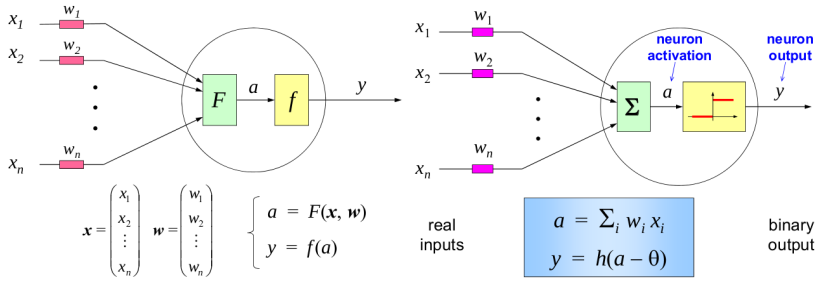
- efficiency
- fault tolerance
- graceful degradation

In last three decades neural networks have become more and more popular, mainly thanks to the fact that, after a long and hard time of research on how to overcome mathematical difficulties, the computational abilities and the huge amount of accessible data has literally exploded.

Artificial neural networks are not all the same, but they are divided in many categories depending on the problem they solve and the requirements they fulfill. We can use supervised training (learning by examples) or unsupervised one (reinforcement learning), and we can apply a variety of designs, for instance perceptrons, self-organizing maps, Boltzman machines, convolutional nets and so on.

Since in this work we focus on categorization, that is the problem to assign a seen example to a defined category, we decided to use the very simple approach of *perceptrons*. This model has been proposed by Rosemblatt in 1957 and its hardware implementation was one of the first and famous achievements in technology before strong limitations were demonstrated in 1969, causing the so called "AI winter". Thanks to a great algorithm improvement called Backpropagation (Rumelhart, Hinton and Williams, 1986) the research over artificial neural networks spread again until the progress seen nowadays.

The *perceptron* in a specialization of the binary-threshold neuron presented some years before, as visible in **Figure 2**.



(a) *Generic neuron* (b) *Perceptron*

Figure 2: Basic schema of artificial neurons under consideration.

N weights are associated to inputs and are processed with $F(w, x) = a$. A final activation function $f(a) = y$ determines the actual output of the neuron. Perceptron applies the sum as F and multiple choices of activation function can be made: sigmoid, hyperbolic tangent, signum and Rectified Linear functions (ReLU) are of common use.

3 Developement

We design a modular component based software that relies on:

- ALSA management of sound cards for reading files and input device
- a library for building, training, saving and loading a networks that can make predictions on properly adapted audio samples
- thread based programs for the realization of a soft real-time tuner

Our tuner is capable of recognizing both instrumental and voice, though the training of the network demonstrated to be affected by difficulties that we are going to explain in detail.

The project is maintained on GitHub [1].

3.1 ALSA utilities

As mentioned in **Chapter 2.1**, while using drivers with ALSA we must take care of parameters. After that, we can use devices in a way that remembers the one followed we files: we open a so called *handler* on a specified device (for our purposes, `default` is sufficient); we allocate, set and apply hardware parameters; we use the device for reading or writing until we close the handler. See **Figure 3** for a summary.

The parameters of our interest are:

- **access**: the way frames are managed into the buffer, dealing with channels. If we have a single channel, frames are simply subsequent, whereas with more channels we can *interleave* them, namely put samples of more channels one after another, or *non-interleave* them, namely put samples of the same channel in series until the end of frames, then begin again with other channels. The type we chose is INTERLEAVED
- **sampling format**: the kind of datum we are going to extract or put into the buffer submitted to the sound card. It is not only related to the size (`short int`, `int`, `float` etc.) but also to the *endianess*. The format we chose is 32-bit `float` Little Endian
- **sample rate**: this is a parameter that is usually rounded down by ALSA, depending on the capability of the soundcard. If the rounded value is not acceptable for the user, he can decide to treat is as an error. We always work with the basic 44100 Hz frequency

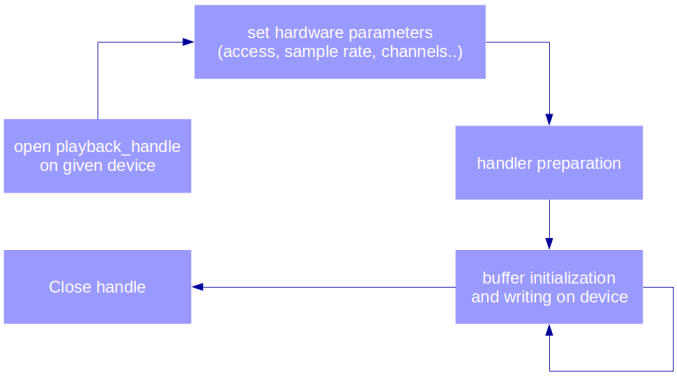


Figure 3: Brief summary of passages to follow while preparing an example app for playback.

- **channel count**: number of channel requested to read or write. We have to remember to prepare the read/write buffer in the proper way, according to the number of channels and the access type just set

The `autil` library provides similar functions to read and write buffer using the ALSA API. They follow the same paradigm followed by the ALSA project and other libraries available on the web: for each format, a proper function can be called.

For further details, refer to `autil.h/autil.c`

3.2 Reading `.wav` files

We trained our networks using two kinds of elements: artificially generated sinusoidals of known frequency, from which we take the FFT, and wav files. Some of these files have been recorded from an electric piano, while other have been downloaded and properly formatted.

The files `wav.h/wav.c` exploit services delivered by the `libsndfile` [2] library, which is designed to deal with a variety of audio files and formats. The process to follow for extract samples is pretty similar to what ALSA requires, except of course doing specific parameter setting before the start. After a wav file is open, characteristics can be stored and easily used for playback or recording, if needed. Our `wav` focuses of opening and reading chunks of files of a certain number of frames. In particular, each chunks is usually not greater of 2205 frames, which is the amount of frames read in 50 milliseconds while capturing audio at 44.100 KHz. Also in this case, functions are organized following the type of datum we need to read, be it `short`, `int` or `float`.

There is also an additional "complete" method to read a wav entirely and playback it out. This can be useful to roughly check the quality and the correctness of the files we upload.

3.3 Library for perceptron-based networks

This is the backbone of the application. There are two distinguished modules about networks: the `pnet` and the `pnetlib` library. We discuss each of them separately.

pnet It contains the definition of perceptron, layer of perceptrons and network of perceptrons, alongside with methods for dynamically instantiate, initialize and destroy every part of it. Moreover, we implement *backpropagation*, to be applied both for weights and for bias.

An instance of *perceptron* contains:

- `int nweights`: number of weights (bias excluded)
- `float* weights`: dinamically allocated array of weights
- `float bias`: trained as an additional constant, whose input is set to -1. It's influence varies with the objective function chosen
- `float a`: sum of all `weights*inputs - bias`
- `float y`: actual output given from objective function evaluated on the previous sum, $f(a)$
- `float d`:δ stored during a backpropagation step
- `float (*fx) (float x)`: pointer to the objective function
- `float (*dfx)(float x)`: derivative of $f(a)$, needed in backpropagation

A `p_layer` and a `p_net` are simply dynamic arrays to build up the entire structure. The *backpropagation algorithm* is implemented here and, given a network and an examples, returns an object that keeps the amount of updates needed for weights and bias: this is the so called *delta rule*. Specifically, in a network of L layers, where w_{ji} is the i-th weight of neuron j in layer l, we get Δw_{ji} as the delta rule applied for w_{ji} (**Figure 4**).

The delta rule is computed as follows. The generalized formula is:

$$\Delta w_{ji} = \eta \delta_j x_i$$

where η is the *learning rate*, x_i is the i-th input (associated to w_{ji}) and δ_j is defined as:

$$\delta_j := -\frac{\partial E_k}{\partial a_j}$$

being E_k the error en example k.

Given L layers (enumerated from 1 to L), and assuming to use as error $E_k = (target_j - y_j)^2$ we start from the output layer computing for each example k:

$$\delta_j^L = (target_j - y_j) f'(a_j)$$

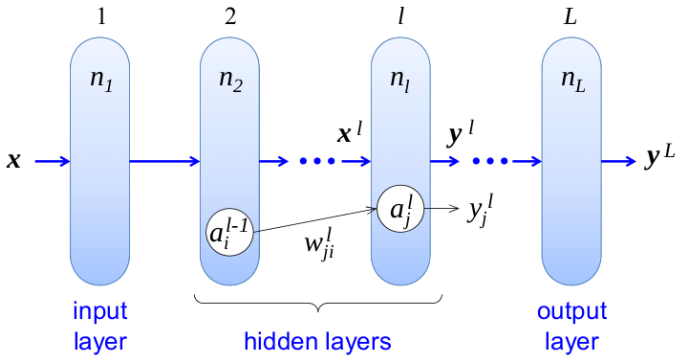


Figure 4: Definitions needed by the algorithm

For any other other layers $l < L$, going from the top to the bottom, being n_l the number of weights of weights incoming on any neuron of layer l, we evaluate:

$$\delta_i^l = f'(a_j^{l-1}) \sum_{j=1}^{n_l} w_{ji}^l \delta_j^l$$

Normally, the delta rule is not applied immediately, but an average is taken. This helps minimizing the error fluctuations and also to limit the *gradient vanishing* phenomenon. We followed this technique too in the actual training algorithm implemented in `pnetlib.h/pnetlib.c`

pnelib This library defines a generalization of examples that can be provided to a perceptron network freely created. A `struct example` is formed by:

- `float* samples, *label`: dynamic arrays for samples of the example, and relative label. The label can vary depending on how we address categorization: for instance, label could be binary or a list of n categories. An object belongs to category i if the i-th element is 1
- `int ns, int nl`: size of the `samples` and `label` arrays

Some methods help to build up a list of this examples in order to create the *training set* required for the network.

Examples can be *normalized* and/or *standardized*, which are common practices used to get a better training set to issue. In fact, with normalized data, the error function tends to be symmetric and the minimum can be reached more quickly. Otherwise, the error function squeezes and a small Δw can cause the error to oscillate. The formula we used for *normalization* (taking all values in the range $[0, 1]$) is:

$$x_i^{new} = \frac{x^{old} - x_{min}}{x_{max} - x_{min}}$$

while, for *standardization*, given M examples of n_s samples, and being μ_i and σ_i^2 the *mean* and *std variation respectively*:

$$\mu_i = \frac{1}{M}\sum_{k=1}^{M} x_i(k)$$

$$\sigma_i^2 = \frac{1}{M}\sum_{k=1}^{M}(x_i(k) - \mu_i)^2$$

$$x_i^{new} = \frac{x^{old} - \mu_i}{\sigma_i}$$

This stated, `pnetlib` is used to perform the training itself. We apply the *mini-batch* approach: we split the training set into batches and we accumulate the delta rule over each example of the batch. At the end of each batch, we average the delta rule on the batch and then apply it. Hence, we can monitor a "local" error averaged over the batch and a "global" error average over the entire training set.

The batch dimension can be set at run time, and if this dimension does not divide the set in equal parts, the surplus is managed in the same way as well, considering the actual dimension of the residual.

This approach is called *mini-batch Stochastic Gradient Descent* algorithm, whose pseudo-code is described below

Initialize weights, η, ε, m, max_epochs; iter = 0;
do {
  Initialize the global error: $E = 0$;
  Shuffle the TS; epoch++;
  for each mini-batch { *epoch*
    iter++;
    for each $(x_k, t_k) \in$ mini-batch { *iteration*
      compute y_{jk} $\forall$ layer (from 2 to L);
      compute δ_{jk} and $\Delta w_{ji}(k)$ $\forall$ layer (from L to 2);
      compute E_k and update global error: $E = E + E_k$;
    }
  }
  update all the weights averaging on mini-batch;
} **while** (($E > \varepsilon$) and (epoch < max_epochs));

In order to contrast error's fluctuation, two methods are commonly used:

1. dynamic variation of the learning rate η, as a function of th error

2. use of *momentum* m

We implemented the second approach with a *momentum* m that can be set at run-time. At the end of the batch, we apply:

$$\Delta w_{ji}(t) = \eta \delta_j x_i + m \Delta w_{ji}(t-1)$$

3.4 A closer look to `ftrain` and `fpitcher`

Since we are using a neural network, the use of Fourier Transforms, filters and other signal processing techniques should be negligible. In fact, by definition, a neural network build itself according to shown examples and to the way it is programmed for learning. This means, for example, that a network could learn to recognize a pattern even in presence of errors, if it categorizes them through examples. For this reason, we designed the training and the "pitcher" to work directly with raw audio samples. Nevertheless, to make the debugging much easier, we shifted to the use of Fast Fourier Transforms: libraries as FFTW3 are widely available and stable [3]. Using FFT, it possible to clearly see the spectrum of a signal together with the frequencies on which a potential noise accumulates (**Figure 5**).

The training set is built up using frequencies of the 4^{th} octave (C4-B4) at different volumes. Each has been normalized before the FFT application. Frequencies are listed in the following table.

PITCH	FREQUENCY [Hz]
C	261.63
C#	277.18
D	293.66
D#	311.13
E	329.63
F	349.23
F#	369.99
G	392.00
G#	415.30
A	440.00
A#	466.17
B	493.86

Then, the neural network is created with 2 layers. The first is the hidden layer (input layer is implicit) with `nhd` neurons, `NUM_INPUTS` weights and `sigmoid` as objective functions. Since we work with real numbers, the FFT of N values will have N complex numbers, whose the first $\frac{N}{2}+1$ are relevant, and the remaining are the conjugate of the first ones and then can be neglected [4].

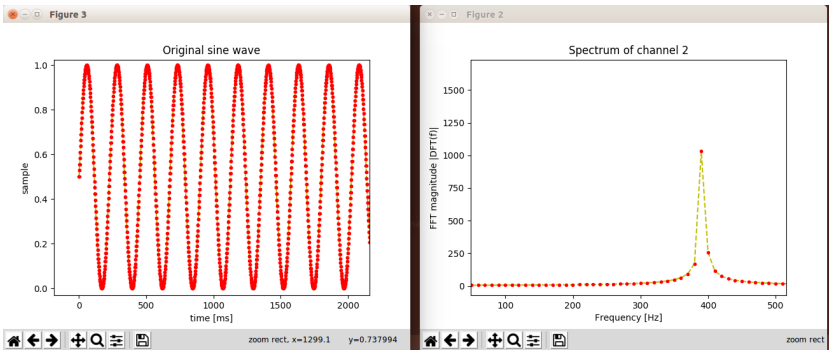


Figure 5: Example of auto-generated sine wave and FFT, using normalization on the initial samples: here, pitch is G at 392.00 Hz.

The second layer is the top layer, hence it will have as many neurons as the number of pitches and a number of weights equal to the number of perceptrons of the previous layer. The code is going to be the following:

```
102 network = p_net_create();
103 p_net_init(network, 2);
104 // add hidden in front of input
105 add_layer(network, 0, nhd, NUM_INPUTS, sigmoid, ddx_sigmoid);
106 // add output layer
107 add_layer(network, 1, NPITCHES, nhd, sigmoid, ddx_sigmoid);
```

After normalization and proper setting of training parameters, we will call:

```
213 p_net_train_SGD (network, max_epoch, batches,
      learning_rate, momentum, min_err,
      &training_set, train_size, lerr_file, gerr_file);
```

The scheme `fpitcher` follows is:

- create 2 threads called `capturer`(ALSA recorder) and `capturer`
- `capturer`loads the network prepared by the previous training, then waits for new samples coming from the `capturer`
- the 2 threads act in mutual exclusion according to reader/writer problem

The thread `capturer`*writes* on a shared buffer locking a mutex each time a new chunk is available on the device, after it has been activated (periodically every 50 ms). A variable `max_overwrite` counts how many times this happens without the `recognizer`has extracted the last written chunk. If the counter crosses a threshold evaluated as 5 seconds: this means the latency of the `recognizer`, that is the latency of the neural network, is too long. Both threads stop in this situation.

The variable `max_overwrite` can be used also for delivering ALSA errors (which cause proper program termination too), throughout `THREAD_EXIT_FLAG`.

It's important to notice that, for every captured (or read from file) chunk, the power of the signal is checked before being issued to `recognizer`and then to the network. We stabilize a threshold computing an average on the `.wav` files we recorded (this approach has been also in **Section 3.6**) and we use it for discarding chunks that are not acceptable because too low. We also made use of the plugin `alsamixer` [5] to tune the capabilities of the embedded microphone of our sound card properly.

The thread `recognizer`*reads* the shared buffer and clears the counter as soon it has output the prediction of the network.

3.5 Problems and future work

Even though the development has been linear and many successful tests have been performed on simple perceptron networks (see `hello_test_and.c`, `hello_t estxor.c`, `hello_t estchar.c` and other), training the network for our purpose efficiently demonstrated to cause a real hard time.

The main discrepancy is that, even if the error on the training seems to go quite small (about 7%), no further improvement is visible and a counter check on the training set itself causes a high failure rate (about 88-91%, meaning an accuracy of about only 10%). This seems to happen because the values of predictions are very small (much less than 0.1) but all similar on average. This means that, while we are seeking for a prediction near to "all 0s and a one 1" on the correct category, we are getting a prediction with values very similar one to each other and no strong winner. The training set has been verified accurately, using plots and synthesized samples, and labels too.

In general, increasing the number of neurons causes *overfitting* ("best" results have been obtained with very few neurons, at most `NPITCHES=12` as the output layer), while is good to maintain high momentum m (about 0.9). We would expect a small *learning rate* to be a good choice (for example 10^{-3}): instead, results tend to degrade fast even under $\eta = 0.1$. Changing objective function also does not give appreciable improvements.

The topic is intriguing, but for the scope of this exam the resolution is left to future work.

3.6 Cross comparison between FFT

As a completion for the work done, we can exploit `autil` and `wav` to get a valuable training set that now will not be used for a network but for providing a base of comparison for raw signals coming from an ALSA device.

We developed `fcross` and `fcross_wav` programs to do apply the schema shown in **Figure 6**. They follow these steps:

1. prepare the reference set
    - prepare N samples of sinusoidal waves or chunks read from `.wav` files

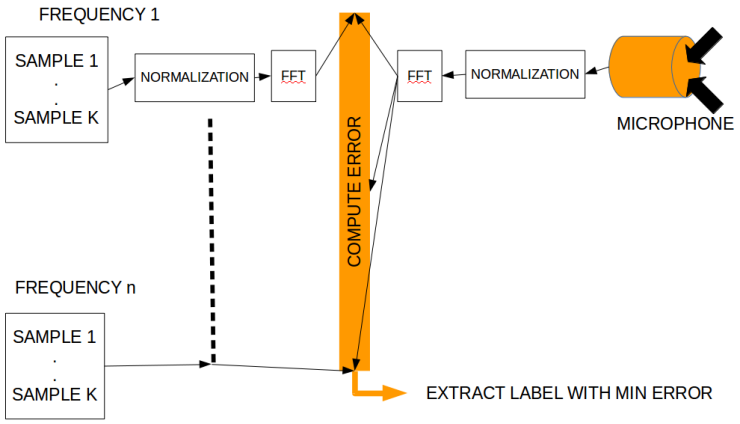


Figure 6: Schema of what's coded in `fcross.c`

- normalize the samples
- store their FFT in a list

2. periodically (we exploit a single `capturer`thread again)
   - capture a 50 ms chunk of audio samples (namely, 2205 frames per channel)
   - compute the minimum average squared error with all the examples of the reference set
   - elect as winner the sample of the reference set corresponding to the minimum error, and print the result

These programs demonstrate to work well (with just some error when changing pitches or while sliding) both with instruments and voice. The first, `fcross.c`, is limited to the 4^{th} octave as seen for `ftrain.c`, while the second, `fcross_wav.c` can distinguish pitches from 2^{nd} to 5^{th} octave, because we scanned wav files for all of them.

References

[1] https://github.com/Ilancia/pitcher

[2] http://www.mega-nerd.com/libsndfile/

[3] http://www.fftw.org/fftw3_doc/

[4] http://www.fftw.org/fftw3_doc/One_002dDimensional-DFTs-of-Real-Data.html#One_002dDimensional-DFTs-of-Real-Data

[5] https://it.wikipedia.org/wiki/Alsamixer