Digital Signal Processing 2/ Advanced Digital Signal Processing
Lecture 5,
Vector Quantizer, LBG

Gerald Schuller, TU Ilmenau

Remember, The Lloyd-Max iteration could look like the following:

1) Start (initialize the iteration) with a random assignment of M reconstruction values (codewords) y_k
2) Using the reconstruction values, compute the boundary values b_k as mid-points between 2 reconstruction values / codewords (**nearest neighbour rule**)
3) Using the pdf of our signal and the boundary values, compute new reconstruction values (codewords) y_k as centroids over the quantisation areas (**conditional expectation/centroid**)
4) Go to 2) until update is sufficiently small (< epsilon)

This algorithm usually converges (it finds an equilibrium and doesn't change anymore), and it results in the minimum distortion D.
It is interesting that this can be generalized to the multidimensional case, to the so-called Vector Quantisation.

Vector Quantisation

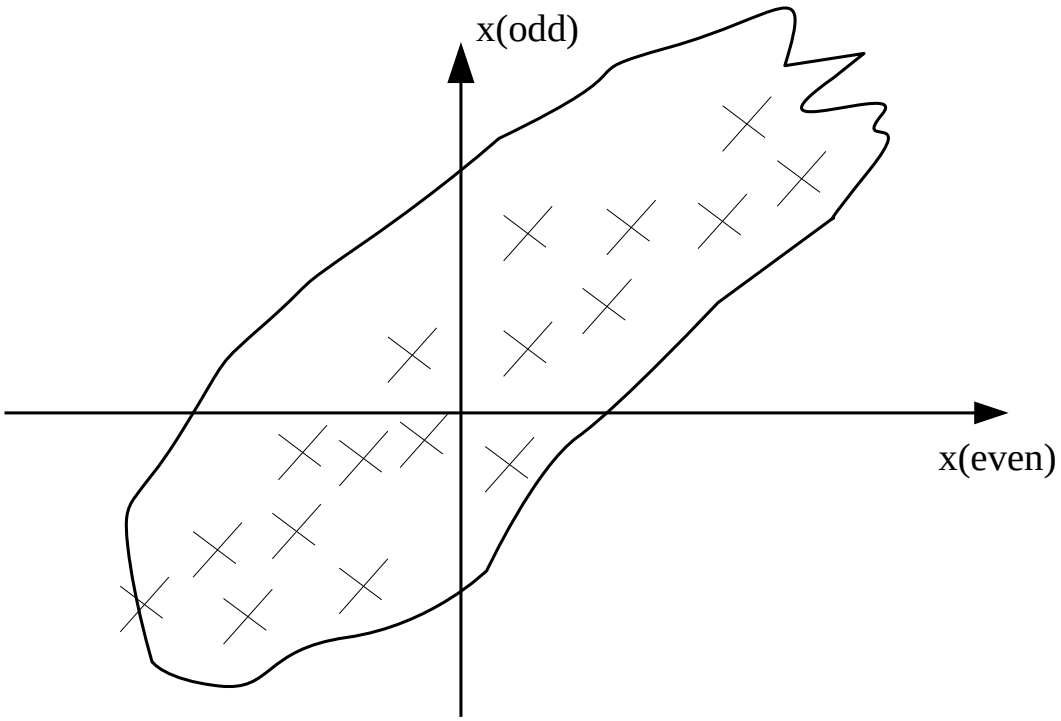
Scalar quantisation usually makes the assumption that the signal to quantize, the source, is **memoryless**, which means each sample is statistically independent of any other sample in the sequence. This can be seen as having no "memory" between the samples. Examples of these sources might be: Thermal noise, white noise, or a sequence of dice tosses, lottery numbers.
But many signals do **have memory**, they have samples which are statistically dependent on other samples in the sequence. Example are: Speech signals, pink noise (noise which has a non-flat spectrum), temperature values over the year, image signals, audio signals.
Since many signals of interest indeed have memory, this suggests that we can do a better job. One possible approach to deal with memory (statistical dependencies) in our signal, is to use the so-called **Vector Quantisation (VQ)**.
A possible **reference** is: "**Introduction to Data Compression**", Section about Vector Quantisation, by Khalid Sayood, Morgan Kaufmann publishers.
How does VQ work? Instead of quantizing each scalar value (each sample) individually, we first group the sequence of samples x(n) into groups of N samples:

[x(0),…,x(N-1)],[x(N),…,X(2N-1)],[x(2N),..,x(3N-1)],…
In this way we obtain a sequence of blocks (also called **vectors**) of size **N samples** each. In this way we obtain a sequence of samples in an **N-dimensional** space. In such a way we can capture or use the memory between samples within each block or vector. The resulting samples with

memory in the N-dimensional space will then lie on or near a **hyperplane** or subspace within this N-dimensional space. Hence we don't need to sample the entire space, but we only need to sample the part of our space where our samples are actually located.
Example: Take correlated samples, such that one sample is always similar to the previous sample, just like in a sequence of speech or audio samples (usually we don't have very high frequencies there, and that means the curve through the samples is more or less smooth).
Now take the dimension N=2. Then we obtain a 2-dimensional vector space of samples, which could look like in the following diagram. The resulting sequence of vectors is: [x(0),x(1)],[x(2),x(3)],[x(4),x(5)],...

For instance, our signal could be:
x=[23,45,21,4,-23,-4], then we get the following **sequence of vectors:**
[23,45]; [21,4]; [-23,-4]

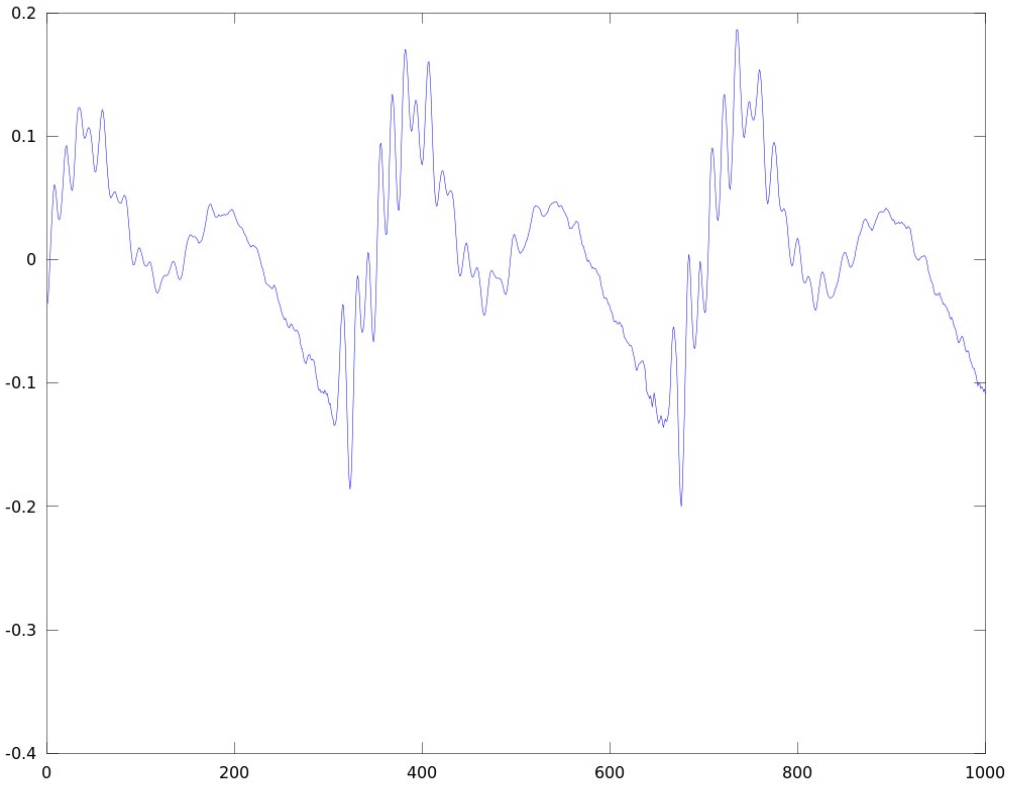
Each of those vectors can now be represented as a point in this 2-dimensional space:



Python Example:

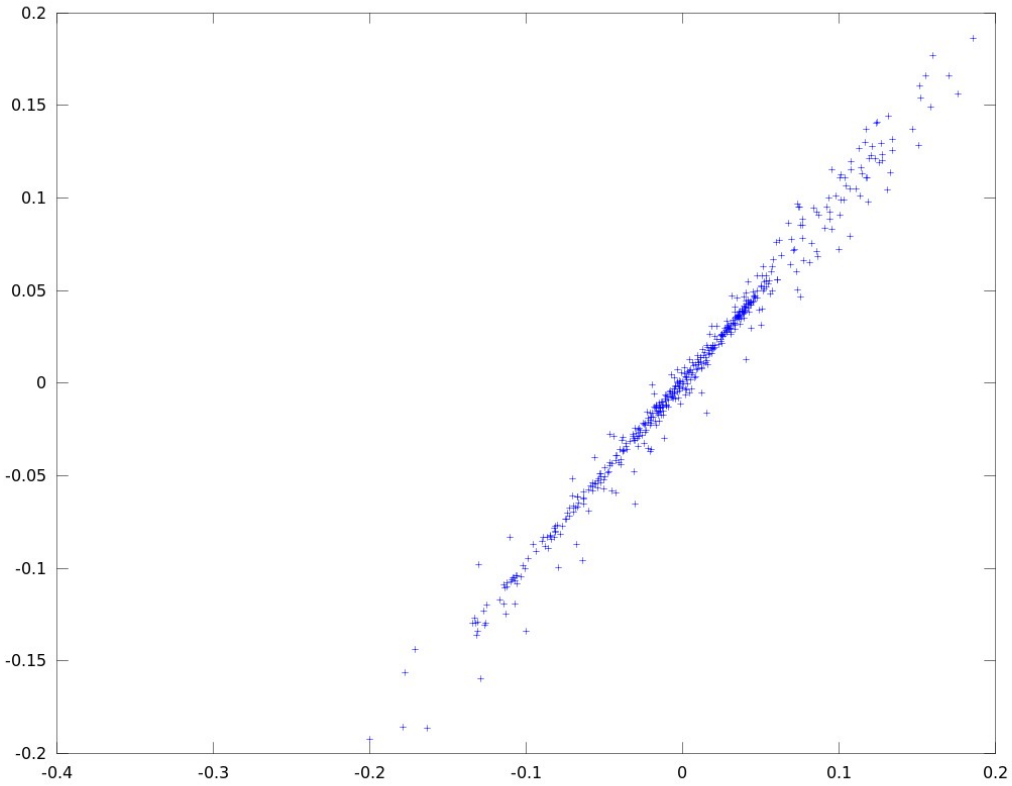
Take a speech signal and read it into Python:

```
ipython --pylab
import scipy.io.wavfile as wav
rate, snd = wav.read('mspeech.wav')
#Take an excerpt of 1000 samples,
#starting at sample 2001 and plot it:
spex=snd[2000+np.arange(1,1000)]
plot(spex)
```



Now plot the 2 dimensional vectors, with their sample values of even indices on the x axes, and their sample values of odd indices on the y axis. Each such pair is plotted as a ‘+’:

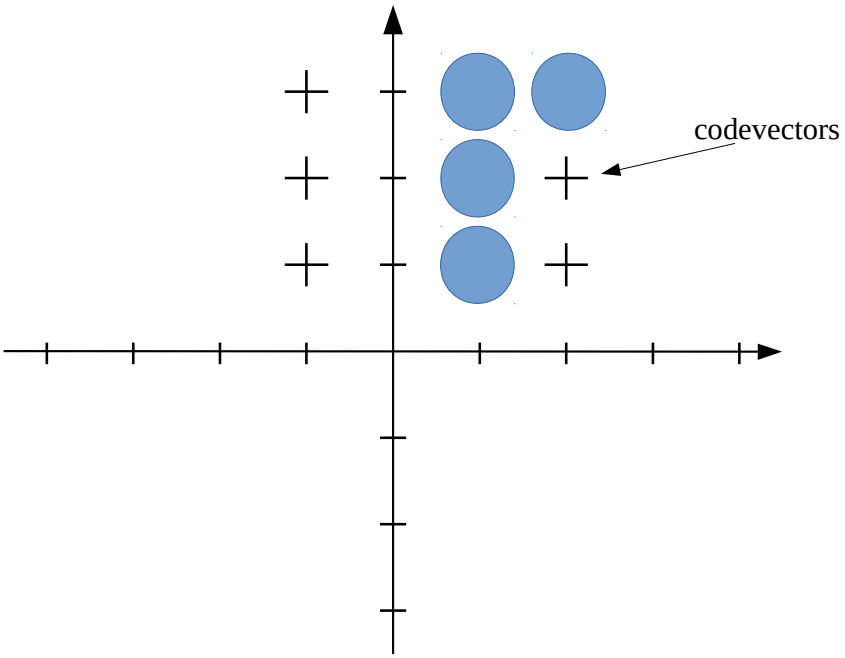
```
plot(spex[2::2], spex[1::2], '+')
```



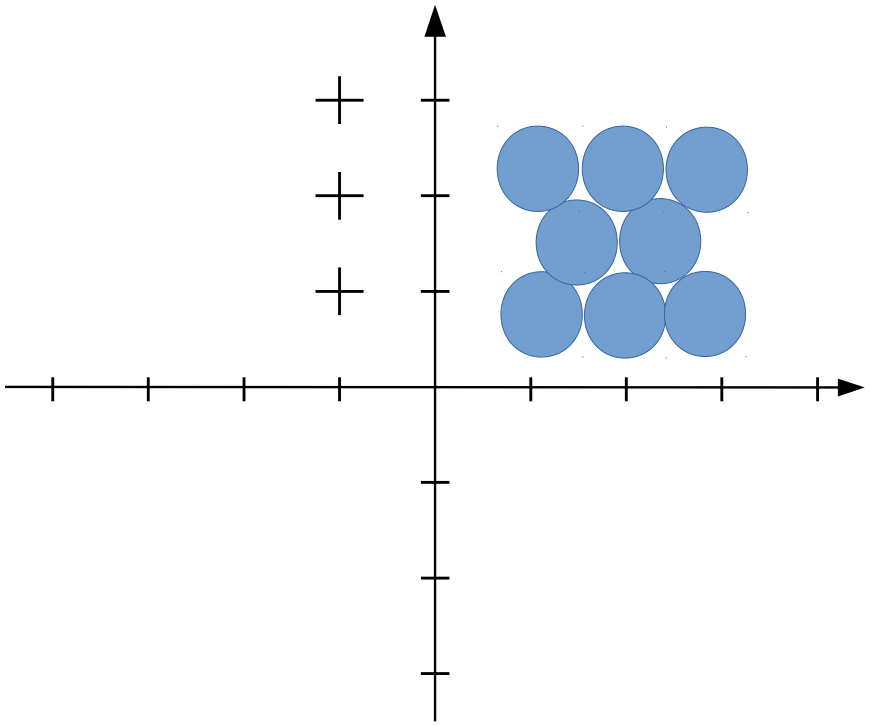
We can see: Since the odd and the even samples are similar to each other, we get a distribution of vector points near the **diagonal** of the space! Hence we also need to sample this space only near the diagonal, or more generally speaking, we should sample more densely near this diagonal.
This shows that we need **fewer** reconstruction values or **codewords** as in the 1-dimensional case, which means fewer indices for them, and hence **fewer bits** for the quantized signal.

Vector **Encoder**: Dividing our signal into signal vectors, find the nearest codeword, transmit its index to the decoder:
Vector **Decoder**: Read out the codevector from the codebook using the index from the encoder, **concatenate** the sequence of codevectors into a sample stream.

An interesting property is that vector quantizers not only give an advantage for signals with memory, but also for signals without memory. In the scalar case, we can only sample an N-dimensional space on a regular grid, which is given by the coordinate axis of this space, whereas with VQ we can use something like a densest sphere packing in this N-dimensional space, such that we reduce the distance between reconstruction vectors (the so-called **codewords**), and hence reduce the Expectation of the quantisation error even in this case.
The following image illustrates the densest sphere packing for the case of memoryless signals:
The case of a scalar quantizer:



The case of a vector quantizer with N=2:



Observe that in this way we get a denser “packing”, shifting the spheres in the gaps of the neighbouring layers, which results in a reduced reconstruction error.

How do we do the **quantisation in the N-dimensional case** in general? We choose N-dimensional reconstruction values, which we now call **codewords**. We use the **nearest neighbour** rule to map each N-dimensional **signal vector to** the **nearest codevector**. You could think of the neighbourhood as n-dimensional spheres around each codevector. Each codevector has an index and this index is then transmitted to the receiver, which uses this codevector as a reconstruction value. The collection of all codewords is called a **codebook**. The size of the codebook also determines how many bits are needed for their index. Usually codebooks are fixed, pre-

defined, but there are also adaptive codebooks, for instance in speech coding.

So how do we **obtain our codebook**, our codevectors? Basically like in the Lloyd-Max case, we are just extending it to the N-dimensional case. For N-dimensional case this is called the **Linde-Buzo-Gray (LBG)** Algorithm (see also the Book Introduction to Data Compression):

This could look like the following:

1) Start (**initialize** the iteration) with a **random** assignment of M N-dimensional **codewords**, $\boldsymbol{y}_k$ (bold-face to indicate a vector)

2)Using the codewords, compute the **decision boundary** $\boldsymbol{b}_k$ (bold face to indicate that a line or hyper-plane) as the set of **all** points with equal distance between 2 reconstruction values / codewords (the such constructed regions are also called **Voronoi-regions**), using the **nearest neighbour** rule. To assign a vector to a specific region, we use the nearest neighbour rule directly. We simply test which **codeword is closest** to the observed vector.

3) Using the pdf of our signal and the decision boundary (Voronoi region), compute new **codewords** $\boldsymbol{y}_k$ as **centroids (center of mass)** or **conditional expectation** over the quantisation areas (the Voronoi region). The

same as in 1 dimension, just here the integral is going over N dimensions
4) Go to 2) until update is sufficiently small (< epsilon)

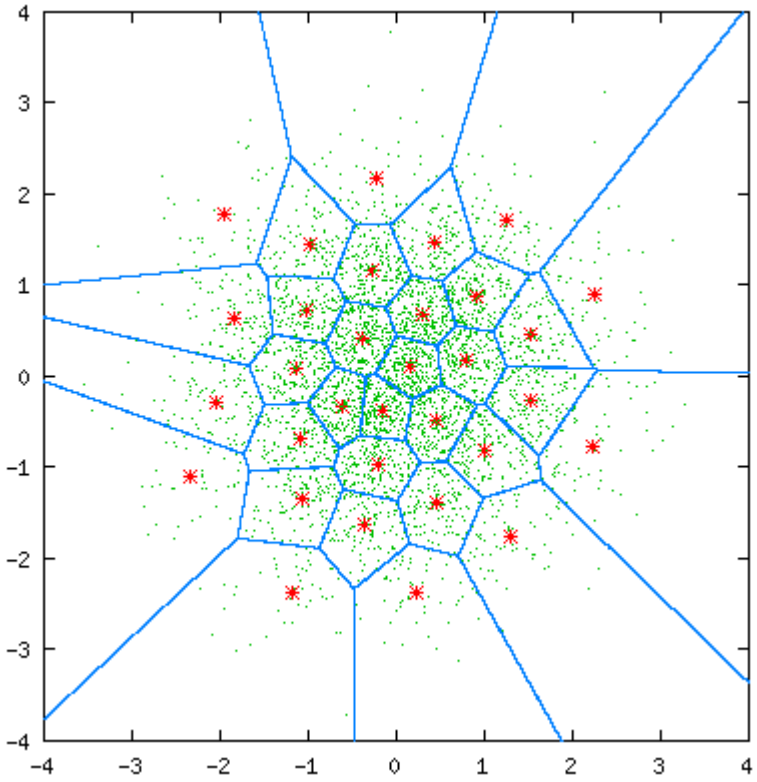
Here we assume we have a pdf of the signal. Observe that here we would need a **multi dimensional pdf or probabiltity distribution**, which is difficult to obtain, since the volume of the space increases exponentially with its dimension, but the number of vectors rather decreases. Hence we get less dense signal points in high dimensional space, which means a pdf is difficult to estimate.
Hence, instead of a multi dimensional pdf, we often only have a so-called **training set** to obtain our codebook. The training set is a set of signals which have statistics like our targeted signals, but which are **only** used to "train" (using LBG) our codebook vectors. To **test** resulting vector quantizer, we should use signals which are **not** in the training set (test set).

We can still use this same algorithm as with the pdf, we just have to compute the **centroid** or conditional expectation differently. Assume we have L samples in our Voronoi neighbourhood region, and want to compute the centroid of this Voronoi region. We could do that by assigning each signal vector **x**(k) of the training set to a probability of 1/L (we assume each vector is equally likely), and then just use the above formula for the centroid, replacing the integrals by sums. This then results in

$$y_k = \frac{\sum_{i \in Voronoi\ region\ k} x(i)}{Number\ of\ signal\ vectors \in Voronoi\ region}$$

This is simply the **average** of all **observed training vectors** in our Voronoi region. The sum contains the indices i of signal vevtors x(i) which are closest to codevector k.
This is then part of our iteration.

After training the codebook with the training set (only for training or learning to obtain the codebook), we have a (fixed) codebook which we can then use for encoding our data. Observe that the training set should be different from our data.
The resulting Vector Quantizer could look like in following image for dimension N=2, where the blue lines are the boundaries of the Voronoi regions (consisting of our $\boldsymbol{b_k}$), the red stars are the codevectors $\boldsymbol{y_k}$, and the green dots the signal vectors:



(From: http://www.data-compression.com/vq.html)

Example. Determine the codebook vectors of a LBG vector quantizer for dimension N=2, and number of codevectors M=2, after one iteration for two vectors with the given training set **x** = [3,2,4,5,7,8,8,9]. Initial codebook vectors are $\mathbf{y}_1$=[1,2], $\mathbf{y}_2$=[5,6].
The training set vectors are hence: [3,2], [4,5], [7,8], [8,9].

The solution follows the algorithm which was explained in the lecture.
1) We start with the given **randomly assigned codebook vectors $\mathbf{y}_k$**.
2)The next step is to calculate **decision boundary $\mathbf{b}_k$** using the **nearest neighbour** rule to obtain the Voronoi region.. This results in a set of midpoints. These mid-points then form the Voronoi boundary line, which is perpendicular to the connecting line of the 2 codewords. The direct mid-point is

$$\boldsymbol{b_k} = \frac{\boldsymbol{y_1} + \boldsymbol{y_2}}{2} \quad .$$

$\mathbf{b}_k$ in our case is the line going through the point [3,4], the line consisting of **all points** which have **equal distance** to the neighboring codevectors. Now we can draw the given training set vectors, codebook vectors, and the Voronoi boundaries. Observe that the computation of the **boundary line** of the mid-points is useful **only for drawing** this picture, but in a computational implementation we don't need to compute it, but use the nearest neighbour rule directly.

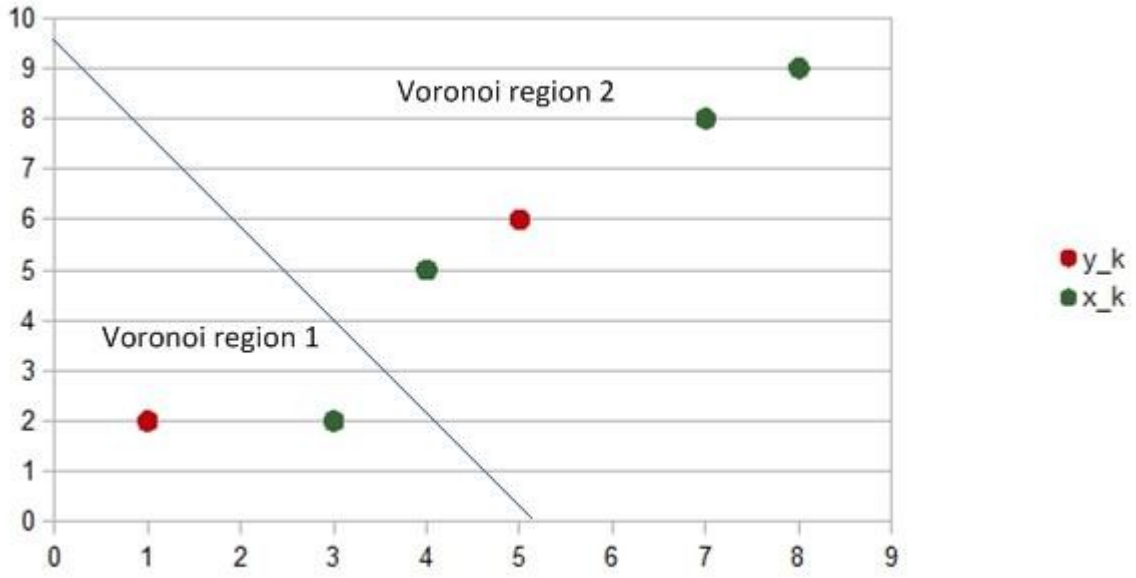


Fig. 1: Illustration of LBG.

3) The next step is to compute the new **codewords y_k** as a **centroid** or **conditional expectation** over a quantization area or Voronoi region. In order to do that, the formula for the trainings set from the lecture should be used:

$$y_k = \frac{\sum_{i \in Voronoi\ region\ k} x(i)}{Number\ of\ signal\ vectors \in Voronoi\ region}$$

In order to find out in **which Voronoi region** a vector is located, we use the nearest neighbour rule. For that we need to calculate the Euclidean **distances** between all the **training set vectors** and **codebook vectors** and then decide which training vectors are closer to which codebook vector.
For each trainings set vector we compute to which codebook vector it has the closest distance, with the Euclidean distances calculated in the following way:

- For trainigs set vector $\mathbf{x_1}$=[3,2]:
  - distance to codebook vector $\mathbf{y_1}$=[1,2]:
  - $d_1=\sqrt{(3-1)^2+(2-2)^2}=\sqrt{4}$
  - distance to codebook vector $\mathbf{y_2}$=[5,6]:
  - $d_2=\sqrt{(5-3)^2+(6-2)^2}=\sqrt{20}$
  - hence $\mathbf{y_1}$ is closer
- For trainigs set vector $\mathbf{x_2}$=[4,5]
  - $d_1=\sqrt{(4-1)^2+(5-2)^2}=\sqrt{18}$
  - $d_2=\sqrt{(5-4)^2+(6-5)^2}=\sqrt{2}$
  - Hence $\mathbf{y_2}$ is closer
- For trainigs set vector $\mathbf{x_3}$=[7,8]
  - $d_1=\sqrt{(7-1)^2+(8-2)^2}=\sqrt{72}$
  - $d_2=\sqrt{(7-5)^2+(8-6)^2}=\sqrt{8}$
  - Hence $\mathbf{y_2}$ is closer
- For trainigs set vector $\mathbf{x_4}$=[8,9]
  - $d_1=\sqrt{(8-1)^2+(9-2)^2}=\sqrt{98}$
  - $d_2=\sqrt{(8-5)^2+(9-6)^2}=\sqrt{18}$
  - Hence $\mathbf{y_2}$ is closer

Now we can compute the **centroid or conditional expectation** for each of the 2 Voronoi regions.

- **Voronoi region 1** only contains trainings set vector $\mathbf{x_1}$=[3,2], hence its centroid and new codebook vector is identical to $\mathbf{x_1}$ and we get the **new codebook vector 1** as

$$\boldsymbol{y_1}=[3,2]$$

- **Voronoi region 2** contains the remaining 3 vectors. We obtain its centroid by averaging over them, and obtain the **new codebook verctor 2** as

$$\boldsymbol{y_2}=[\frac{4+7+8}{3},\frac{5+8+9}{3}]=[6+\frac{1}{3},7+\frac{1}{3}]$$

So the updated codebook vectors will look in the following way:

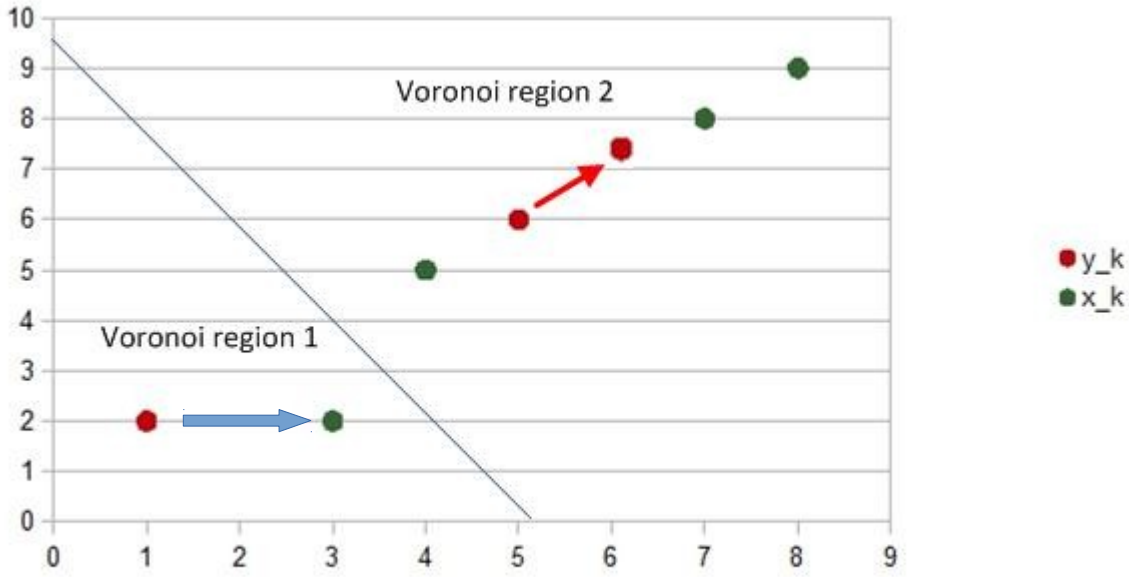


Fig. 2: Updated quantization areas.

4) Go back to step 2) and repeat the procedure until the result does not change much anymore.

Vector Quantization in an Encoder and Decoder

-Both, encoder and decoder, have the same codebook in their memory.

-In an Encoder we first convert our signal sample stream to our vectors.

-Then we map those vectors to the nearest code vectors.

-We transmit the indices of those codevectors to the decoder

-The decoder converts the indices back to the codevectors

-The codevectors are then concatenated and converted back into a stream of samples.

Example:
Encoder:
stream to vectors:
x: [3,4],[7,8],…
Vectors to codevectors:
y: [4,5],[6.7]
to indices:
k: 4, 5
Decoder:
Indices to vectors:
y: [4,5],[6,7]
to stream of samples:
xrek: 4,5,6,7