

Mines ParisTech, S1916

Digital Audio Coding

by

Sébastien Boisgérault

Sebastien.Boisgerault@mines-paristech.fr



Digital Audio Coding by Sébastien Boisgérault is licensed under a Creative Commons Attribution 3.0 Unported License : <http://creativecommons.org/licenses/by/3.0/legalcode>.

Contents

1 Coding	5
1.1 Binary Data	5
1.1.1 Bits	5
1.1.2 Bytes and Words	7
1.1.3 Integers	9
1.2 Information Theory and Variable-Length Codes	12
1.2.1 Entropy from first principles.	13
1.2.2 Alphabets, Symbol Codes, Stream Codes	16
1.2.3 Optimal Length Coding	22
1.2.4 Golomb-Rice Coding	27
2 Quantization	33
2.1 Principles of Scalar Quantization	33
2.1.1 Quantizers	33
2.1.2 Uniform Quantization	36
2.1.3 Quantization of Random Variables	36
2.1.4 Implementation of Non-Uniform Quantizers	37
2.2 Logarithmic Quantization	38
2.2.1 The μ -law Quantizer	39
2.2.2 IEEE754 Floating-Point Numbers and A -law	41
2.3 Signal-to-Noise Ratio	43
3 Linear Prediction	47
3.1 Prediction Principles	47
3.1.1 Polynomial Prediction	47
3.1.2 Optimal Linear Prediction	48
3.1.3 Finite Impulse Response (FIR) Filters	52
3.1.4 Auto-Regressive (AR) Filters	53
3.1.5 Transfer Function, Stability and Frequency Response	54
3.2 Voice Analysis and Synthesis	56
3.2.1 The TIMIT corpus	56
3.2.2 Voice Analysis and Compression	58
3.2.3 Linear Prediction Coding	65
4 Spectral Methods	67
4.1 Signal, Spectrum, Filters	67
4.1.1 Convolution and Filters	70
4.2 Finite Signals	72

4.2.1	Design of Low-Pass Filters	73
4.2.2	Spectrum Computation	74
4.3	Multirate Signal Processing	78
4.3.1	Decimation and Expansion	78
4.3.2	Downsampling and Upsampling	81
4.3.3	Ideal Filter Banks and Perfect Reconstruction	83
4.3.4	Filter Banks and Perfect Reconstruction	84
4.3.5	Cosine Modulated Filter Banks	84
4.3.6	Polyphase Representation of Filters Banks	85
4.4	Psychoacoustics - Perceptual Models	90
4.4.1	Acoustics - Physical Values	90
4.4.2	Threshold in Quiet	91
4.4.3	Simultaneous Masking	92
4.4.4	Spreading Functions	94
4.4.5	Implementation - Bit Allocation Strategies	94

Chapter 1

Coding

1.1 Binary Data

Digital audio data – stored in a file system, in a compact disc, transmitted over a network, etc. – always end up coded as binary data, that is a sequence of bits.

1.1.1 Bits

Binary Digits and Numbers

The term “bit” was coined in 1948 by the statistician John Tukey as a contraction of **binary digit**. It therefore refers to a numeral symbol in base 2: either 0 and 1. Several



Figure 1.1: **John Wilder Tukey** (June 16, 1915 – July 26, 2000) was an American statistician best known for development of the FFT algorithm and box plot - source: http://en.wikipedia.org/wiki/John_Tukey.

binary digits may be used to form **binary numbers** in order to represent arbitrary integers. In the decimal system (base 10), “42” denotes $4 \times 10^1 + 4 \times 10^0$; similarly, in base 2,

$$b_0 b_1 \cdots b_{n-1} \text{ where } b_i \in \{0, 1\}$$

represents the integer

$$b_0 \times 2^{n-1} + b_1 \times 2^{n-2} + \cdots + b_{n-1} \times 2^0.$$

As an example the number 42 (decimal representation), equal to $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$, has the binary representation 101010.

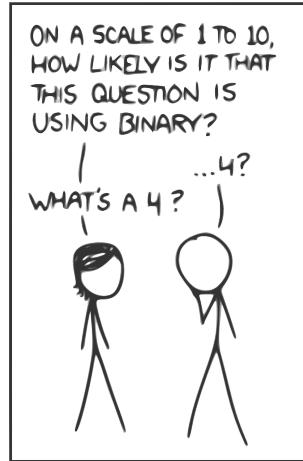


Figure 1.2: <http://xkcd.com/953/>

The term “digit” used in the construction of “bit” is somehow self-contradictory as the etymology of “digit” refers to the base ten:

The name “digit” comes from the fact that the 10 digits (ancient Latin *digita* meaning fingers) of the hands correspond to the 10 symbols of the common base 10 number system, i.e. the decimal (ancient Latin adjective *dec.* meaning ten) digits. http://en.wikipedia.org/wiki/Numerical_digit

In PYTHON, the decimal notation is obviously available to define number literals, but we also may use the prefix `0b` to define a number with its binary representation. The built-in function `bin` returns the binary representation of a number as a string:

```
>>> 42
42
>>> 0b101010
42
>>> bin(42)
'0b101010'
```

Note that there is not a unique binary representation of an integer: one can add as many leading zeros without changing the value of the integer. The function `bin` uses a *canonical* representation of the binary value that uses the minimum number of digits necessary to represent the binary value ... except for 0 which is represented as `0b0` and not `0b`! Except for this particular value, the first binary digit (after the prefix `0b`) will always be 1. However, sequences of bits beginning with (possibly) multiple 0's are valid: `0b00101010` is a valid definition of 42. Finally, note that negative integers may also be described within this system: `-42` is for example denoted as `-0b101010`. Several arithmetic operators in PYTHON (operation on numbers), for example `<<`, `>>`, `|`, `^`, `&`, are far simpler to understand when their integer arguments are in binary representation. Consider:

```
>>> print 42 << 3           >>> print bin(0b101010 << 3)
336                         0b101010000
>>> print 42 >> 2         >>> print bin(0b101010 >> 2)
10                          0b1010
>>> print 42 | 7           versus      >>> print bin(0b101010 | 0b111)
47                          0b101111
>>> print 42 ^ 7           >>> print bin(0b101010 ^ 0b111)
45                          0b101101
>>> print 42 & 7           >>> print bin(0b101010 & 0b111)
2                           0b10
```

To summarize:

- `<< n` is the **left shift by n**: bits are shifted on the left by `n` places, the holes being filled with zeros,
- `>> n` is the **right shift by n**: bits are shifted on the right by `n` places, bits in excess being dropped,
- `|` is the **bitwise or**: the bits of the two binary numbers are combined place-by-place, the resulting bit being 0 if both bits are 0, and 1 otherwise,
- `^` is the **bitwise exclusive or (xor)**: the bits of the two binary numbers are combined place-by-place, the resulting bit being 1 if exactly one of the bits is 1, and 0 otherwise,
- `&` is the **bitwise and**: the bits of the two binary numbers are combined place-by-place, the resulting bit being 1 if both bits are 1, and 0 otherwise,

1.1.2 Bytes and Words

In many application contexts, we consider binary data as sequences of *groups* of a fixed number of bits named **bytes**. Consider for example that every data in computer memory is accessed through pointers and that a pointer gives the location of a byte, not of an individual bit. All the same, filesystems store data into files that contain an entire number of bytes. Despite several historical choices of the size of the byte (due to different hardware architectures), nowadays, the *de facto* size for the byte is 8 bits, that is the byte is an **octet**. In the sequel, we will drop the octet term entirely and use byte instead.

Bytes are sometimes grouped into even larger structures, called **words**, that may group 2, 3, 4, etc. bytes. The choice of the size of a word often reflect the design of a specific hardware architecture¹.

Alternate Representations - Hexadecimal and ASCII

The hexadecimal representation of an integer uses the base 16. As a consequence, any byte may be represented by an hexadecimal number with two digit. This representation, being more compact than the binary, often makes binary data easier to interpret by human beings. The 10 first hexadecimal digits are represented by the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and the 5 following (that correspond to the decimal value 10, 11, 12, 13, 14, 15) are a, b, c, d, e, f (sometimes capitalized).

As an example, here is the representation of a few numbers in decimal, binary and hexadecimal form:

¹[http://en.wikipedia.org/wiki/Word_\(computer_architecture\)](http://en.wikipedia.org/wiki/Word_(computer_architecture))

decimal	binary	hexadecimal
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1100	B
15	11111	F
16	100000	10
42	101010	2A

Figure 1.3: decimal, binary and hexadecimal representation of several small integers.

Integer literals in hexadecimal notation are supported in PYTHON with the `0x` prefix. For example

```
>>> 0x10
16
>>> 0xff
255
>>> 0xcaffe
831486
```

In PYTHON 2.x, binary data is also naturally represented as **strings**, instances of the type `str` ; reading data from file objects or writing into them is made via a string representation. String are therefore used as the same time to describe text – even if in this case the **unicode strings** of type `unicode` are more appropriate – and binary data.

Strings are delimited with quotes. Within the quotes, a byte may be denoted a symbol (letter, digits, punctuation marks, etc.) – in which case the byte value is the ASCII code of the symbol – or by an escape sequence `\x??` where ?? is the hexadecimal representation of the byte. The latter case is handy when the symbol that would represent the byte is not printable. For example, `"S\xc3\xa9bastien"` is the string that represent the text "Sébastien" in the utf-8 text encoding: the e acute does not belong to the set of ASCII printable characters and is represented by the combination of the bytes `0xc3` and `0xa9`.

Consider the string made of all characters whose code increases from 0 to 255. We write it to a file and use the canonical output of the `hexdump` command to display to file content. The middle columns contain the hexadecimal representation of the data and the right column the character representation. The symbols outside of the printable characters range (hexadecimal 20 to 7F) are replaced with dots.

```
>>> string = "".join([chr(i) for i in range(256)])
>>> file = open("file.txt", "w")
>>> file.write(string)
>>> file.close()
>>> import os
>>> e = os.system("hexdump -C file.txt")
```

```

00000000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f |.....|
00000010 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f |.....|
00000020 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f |!"#$%&'()*+,-./|
00000030 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f |0123456789;:<>?|
00000040 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f |@ABCDEFGHIJKLMNO|
00000050 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f |PQRSTUVWXYZ[\]^_|
00000060 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f |`abcdefghijklmnopqrstuvwxyz{|}~.|
00000070 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f |pqrstuvwxyz{|}~.|
00000080 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f |.....|
00000090 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f |.....|
000000a0 a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af |.....|
000000b0 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf |.....|
000000c0 c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf |.....|
000000d0 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df |.....|
000000e0 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef |.....|
000000f0 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff |.....|

```

The character representation of most binary data that is not text is going to be meaningless, apart from text tags used to identify the nature of the file, or specific chunks of text data within a heterogeneous data content. For example, the hexdump of the start of a WAVE audio file may look like:

```

>>> e = os.system(r"hexdump -C -n160 You\ Wanna\ Have\ Babies.wav")
00000000 52 49 46 46 d8 cf 06 00 57 41 56 45 66 6d 74 20 |RIFF....WAVEfmt |
00000010 10 00 00 00 01 00 02 00 44 ac 00 00 10 b1 02 00 |.....D.....|
00000020 04 00 10 00 4c 49 53 54 18 00 00 00 49 4e 46 4f |....LIST....INFO|
00000030 49 41 52 54 0c 00 00 00 4a 6f 68 6e 20 43 6c 65 |IART....John Cle|
00000040 65 73 65 00 64 61 74 61 94 cf 06 00 28 ff 2b ff |ese.data....(.+.)|
00000050 2b ff 2f ff 34 ff 37 ff 3f ff 3e ff 46 ff 41 ff |+./.4.7.?,>.F.A.|
00000060 48 ff 43 ff 41 ff 3f ff 32 ff 30 ff 15 ff 13 ff |H.C.A.?2.0.....|
00000070 f1 fe f0 fe cf fe d1 fe bf fe c1 fe c1 fe |.....|
00000080 cd fe d0 fe de fe e4 fe f5 fe f9 fe 0c ff 0f ff |.....|
00000090 24 ff 26 ff 35 ff 35 ff 38 ff 3b ff 2c ff 32 ff |$.&.5.5.8.;,.2.|
000000a0 1c ff 22 ff 13 ff 16 ff 1b ff 1a ff 36 ff 38 ff |...".....6.8.|
000000b0 5f ff 65 ff 94 ff 96 ff ce ff ca ff ff ff 00 00 |_-e.....|
000000c0 2b 00 2e 00 4b 00 4e 00 59 00 5e 00 57 00 5b 00 |+..K.N.Y.^W.[.]|
000000d0 4a 00 4c 00 43 00 41 00 43 00 42 00 4b 00 4f 00 |J.L.C.A.C.B.K.O.|
000000e0 59 00 5a 00 64 00 62 00 63 00 64 00 5b 00 5d 00 |Y.Z.d.b.c.d.[.]|
000000f0 4f 00 4d 00 3d 00 38 00 1c 00 1b 00 f5 ff fa ff |O.M.=.8.....|
00000100 ce ff d4 ff a8 ff ae ff 7e ff 84 ff 4c ff 52 ff |.....~....L.R.|
00000110 18 ff 1b ff ee fe f0 fe d6 fe d8 fe d0 fe |.....|
00000120 cd fe cf fe c8 fe ca fe c4 fe c0 fe c8 fe c3 fe |.....|
00000130 d4 fe d3 fe e8 fe ea fe fd fe ff fe 08 ff 08 ff |.....|
00000140

```

1.1.3 Integers

NumPy provide 8 different types to describe integers: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32` and `int64`. The digits at the end of the type name denote the size in bits of the representation: all these representations are **fixed-size** and use an entire number of bytes so that they may be handled efficiently by the hardware. The type whose name start with a “u” are unsigned, they represent only non-negative integers ; the others are signed, they may represent negative integers as well.

We will explain how these types represent integers as binary data. We will use the `bitstream` library to illustrate this ; more specifically, we will only use its ability to handles bits as sequence of booleans and build on top of that the functions needed to manage the integer types.

Unsigned Integers

The non-negative integers that may be described in one byte are in the range $0, 1, \dots, 255$. Writing them in a binary stream one bit at a time may be done like that:

```
def write_uint8(stream, integers):
    integers = array(integers)
    for integer in integers:
        mask = 0b10000000
        while mask != 0:
            stream.write((integer & mask) != 0)
            mask = mask >> 1
```

The use of the statement `integers = array(integers)` ensures that single integers, lists of integers and array of integers will all be handled as arrays.

Reading unsigned 8-bit integers from a stream is even simpler:

```
def read_uint8(stream, n=None):
    if n is None:
        integer = 0
        for _ in range(8):
            integer = (integer << 1) + int(stream.read(bool))
        return integer
    else:
        return array([read_uint8(stream) for _ in range(n)], uint8)
```

This code can easily be generalized to handle 16-bits or 32-bits integers.

The `bitstream` library actually supports these types already, with quite a few extra features – error handling, efficient vectorization, and static typing for extra performance via CyTHON².

If that was not already done, we could register the above encoder and decoder via

```
bitstream.register(uint8, reader=read_uint8, writer=write_uint8)
```

The following PYTHON session demonstrates a few ways to write 8-bits unsigned to streams and read them from streams.

```
>>> stream = BitStream()
>>> stream.write(uint8(0))
>>> stream.write(15, uint8)
>>> write_uint8(stream, 255)
>>> print stream
000000000000111111111111
>>> stream.write([0, 15], uint8)
>>> stream.write([uint8(255), uint8(0)])
>>> stream.write(array([15, 255], uint8))
>>> print stream
0000000000111111111111000000000000011111111111000000000000111111111111
>>> stream.read(uint8)
0
>>> stream.read(uint8, 2)
array([ 15, 255], dtype=uint8)
>>> stream.read(uint8, inf)
array([ 0, 15, 255, 0, 15, 255], dtype=uint8)
```

The first form to write integers – `stream.write(uint8(0))` – automatically detects the type of the argument, the second – `stream.write(15, uint8)` – uses an explicit type argument and is slightly faster. The third one that calls the writer directly and bypasses the call to `stream.write` – `write_uint8(stream, 127)` – is again slightly faster. But the real way to make a difference performance-wise is to vectorize these calls and write several

²<http://cython.org/>

integers at once, either lists of integers – as in the calls to `stream.write([0, 15], uint8)` or `stream.write([uint8(255), uint8(0)])` – or better yet write NumPy arrays with data type `uint8` as in the call `stream.write([[15, 255], uint8])`.

Reading such integers from a stream may be done one integer at a time – as in the call `stream.read(uint8)` – or many at the same time, the data being packed into NumPy arrays – as in `stream.read(uint8, 2)`, the second argument being the number of integers. The number may be `numpy.inf` to read integers until the end of the stream.

Signed Integers

A first approach to the coding of 8-bit signed integers would be to use the first bit to code the sign and the remaining 7 bits to code the absolute value of the integer.

```
def write_int8(stream, integers):
    integers = array(integers)
    for integer in integers:
        stream.write(integer < 0)
        integer = abs(integer)
        mask = 0b1000000
        while mask != 0:
            stream.write((integer & mask) != 0)
            mask = mask >> 1
    stream.write
```

In this scheme, the code `1000000` is never used (it would correspond to -0), and therefore we could code 255 different integers, from -127 to 127 . By shifting the negative values by 1 we could use all 8-bit codes and therefore encode the -128 to 127 range. The modification would be:

```
def write_int8(stream, integers):
    integers = array(integers)
    for integer in integers:
        if integer < 0:
            negative = True
            integer = - integer - 1
        else:
            negative = False
        stream.write(negative)
        mask = 0b1000000
        while mask != 0:
            stream.write((integer & mask) != 0)
            mask = mask >> 1
```

The *actual* standard coding of 8-bit signed integers uses a slightly different scheme called **two's complement**: conceptually, we encode the integer as in the previous scheme but as a last step, the bits other than the sign bits are inverted – 0 for 1, 1, for 0 – for the negative numbers. A careful examination of the code below shows that it implements effectively the desired scheme ; the line `2**8 - integer - 1` actually explains the name of this scheme:

```
def write_int8(stream, integers):
    integers = array(integers)
    for integer in integers:
        if integer < 0:
            integer = 2**8 - integer - 1
        mask = 0b1000000
        while mask != 0:
            stream.write((integer & mask) != 0)
            mask = mask >> 1
```

The motivation behind two's complement schemes is that addition correctly works between signed integers, without any special handling of the bit sign³. Let's see the result of this encoding on a few examples:

```
>>> BitStream(0, int8)
00000000
>>> BitStream(127, int8)
01111111
>>> BitStream(-128, int8)
10000000
>>> BitStream(-127, int8)
10000001
>>> BitStream(-3, int8)
11111101
>>> BitStream(-2, int8)
11111110
>>> BitStream(-1, int8)
11111111
```

Network Order

Consider the integer 3882 ; we need at least two bytes to encode this number in an unsigned fixed multi-byte scheme. The `bitstream` module represents this integer as `0000111100101010`: the integer 3882 is equal to $15 \times 2^8 + 42$. We have encoded the 8-bit unsigned integer 15 – the **most significant bits** `00001111` – *first* and then the **least significant bits** `00101010`. This scheme is called the **big endian** ordering and this is what the `bitstream` module support by default. The opposite scheme – the **little endian** ordering – would encode 42 first and then 15, and then the bit content would be `0010101000001111`.

NUMPY provides a handy method on its multi-byte integer types: `newbyteorder`. It allows to switch from a little endian representation to a big endian and reciprocally. See for example:

```
>>> BitStream(uint16(42))
00000000010101
>>> BitStream(uint16(42).newbyteorder())
0010101000000000
>>> BitStream(uint32(42))
0000000000000000000000000000101010
>>> BitStream(uint32(42).newbyteorder())
00101010000000000000000000000000
```

So reading an integer encoded in little endian representation is done in two steps: read it (as if it was encoded with the big endian representation) and then use the method `newbyteorder` on the result.

1.2 Information Theory and Variable-Length Codes

In this section, we introduce the modelling of **sources** or **channels** as random generators of symbols and measure the quantity of information they have. We'll see later – as anyone can guess – that such a measure directly influences the size of the binary data necessary to encode such sources.

³see for example http://en.wikipedia.org/wiki/Two's_complement

1.2.1 Entropy from first principles.

The (Shannon) information content of an event E is:

$$I(E) = -\log_2 P(E) \quad (1.1)$$

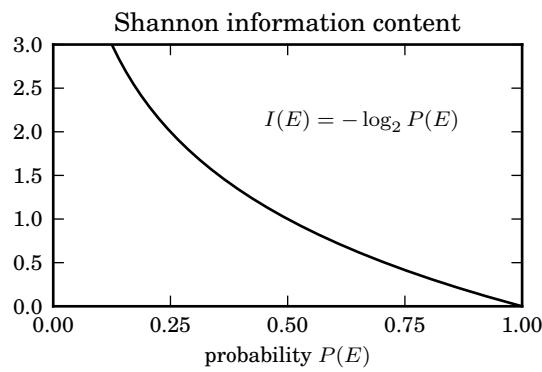
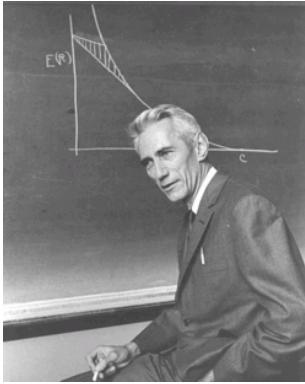


Figure 1.4: Claude E. Shannon (April 30, 1916 – February 24, 2001) was an American mathematician, electronic engineer, and cryptographer known as “the father of information theory”.

This expression may actually be derived from a simple set of axioms:

- **Positivity:** the information content has non-negative – finite or infinite – values,
- **Neutrality:** the information content of an event only depends on the probability of the event,
- **Normalization:** the information content of an event with probability $1/2$ is 1 ,
- **Additivity:** the information content of a pair of independent events is the sum of the information content of the events,

The positivity of the information content is a natural assumption. The value $+\infty$ has to be allowed for a solution to the set of axioms to exist: it does correspond to events whose probability is 0 ; they are so unlikely that their occurrence brings an infinite amount of information. More generally, it's easy to see, given the expression of (1.1) that the less likely an event is, the bigger its information content is: The additivity axiom is also quite natural but one shall emphasize the necessity of the independence assumption. On the contrary, consider two events E_1 and E_2 that are totally dependent: if the first occurs, we know for sure that the second also will. Then $P(E_1 \wedge E_2) = P(E_1) \times P(E_2|E_1) = P(E_1)$ and therefore, the information content $I(E_1)$ is equal to $I(E_1 \wedge E_2)$: the occurrence of the second event brings no further information.

The normalization axiom is somehow arbitrary: we could have selected any finite positive value instead of 1 , but this convention is the most convenient in the context of binary data. This axiom basically says that the probability that a random bit – whose values 0 and 1 are equally likely – is 1 (or 0 for that matter) is 1 . So if we imagine a

memory block of n bits whose values is random and independent, for any sequence of n values in $\{0, 1\}$, the probability that the memory block has that precise state is $n \times 1 = n$. With this convention, the information content unit corresponds to the number of bits ! Therefore, it makes sense to use the word “bit” as a basic measure of information content ; Shannon use the word with this meaning as soon as 1948, the year Tukey invented it.

Let's now prove that the first four fundamental axioms imply that the information content of an event is given by the formula (1.1).

Proof. Let $f : [0, 1] \rightarrow [0, +\infty]$ be such that for any event E , we have $I(E) = f(P(E))$. Let E_1 and E_2 be independent events with $p_1 = P(E_1)$ and $p_2 = P(E_2)$. As $P(E_1 \wedge E_2) = p_1 \times p_2$, we have

$$f(p_1 \times p_2) = I(E_1 \wedge E_2) = I(E_1) + I(E_2) = f(p_1) + f(p_2).$$

Hence the function $g = f \circ \exp$ is additive: for any $x, y \in [-\infty, 0]$, $g(x + y) = g(x) + g(y)$. As g takes non-negative values, g is non-increasing. Moreover, as $g(-\log 2) = f(1/2) = 1$, it is finite on $(-\infty, 0]$ and satisfies $g(-\infty) = +\infty$ and $g(0) = 0$. It is also continuous on $(-\infty, 0]$. As $g(-p/q) = p/q \times g(-1)$ for any $(p, q) \in \mathbb{N} \times \mathbb{N}^*$, by continuity, $g(x) = -xg(-1)$: g is homogeneous. Precisely, $g(x) = -(x/\log 2)g(-\log 2) = -x/\log 2$ and therefore $f(p) = g \circ \log(p) = -\log p/\log 2 = -\log_2 p$. ■

Let X be a discrete random variable. The **entropy** of X is the mean information content among all possible outcomes

$$H(X) = \mathbb{E}[x \mapsto I(X = x)], \quad (1.2)$$

or explicitly

$$H(X) = - \sum_x P(X = x) \log_2 P(X = x) \quad (1.3)$$

If X takes N possible different values x_1 to x_N , we can show that the entropy is maximal if

$$P(X = x_1) = \dots = P(X = x_N) = 1/N$$

that is, if the variable X is “totally random”, all of its values being equally likely. In this case, we have

$$H(X) = \log_2 N$$

In particular, if the random variable X with values in $\{0, 1, \dots, 2^n - 1\}$ is a model for the state of a memory block of n bits where all states are equally likely, then

$$H(X) = n$$

so once again, it makes sense to attach to the entropy a unit named “bits”.

Password Strength measured by Entropy

Given that entropy measures the quantity of information of a random symbol source, it is an interesting tool to measure the strength of a password. Let's measure the entropy attached to the first password generation method described in figure 1.5; the algorithm generates passwords such as

```
$ ./password.py 5
Ablativa18'
Horseplay>4
greff0tome1*
beelzebub)4
Containments5.
```

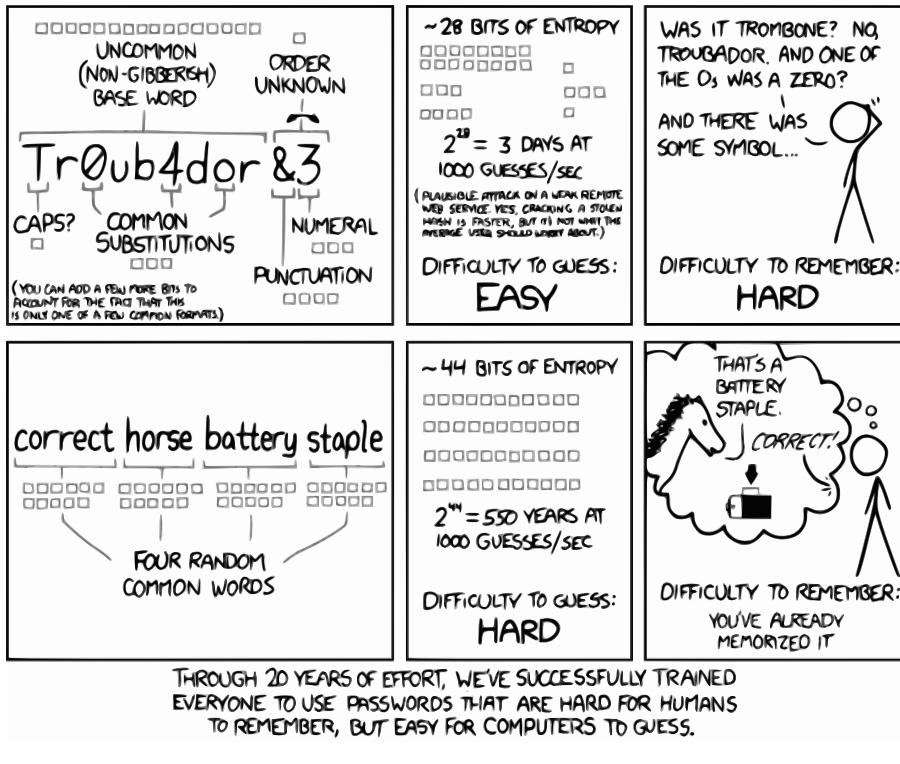


Figure 1.5: "Tr0ub4dor&3" or "correct horse battery staple" ? (<http://xkcd.com/936/>)

Here are the steps that are reproduced:

- **long and uncommon base word.** Pick at random an english word of at least 9 letters. The words.py utility gives us the number of such words:

```
$ ./words.py "len(word) >= 9"
67502
```

At this stage, your pick may be:

"troubador"

If every word in the dictionary has the same chance to be picked, the entropy of your word generator so far is:

```
>>> log2(67502)
16.042642627599378
```

So far our estimate closely matches the one from fig. 1.5 (16 grey “bit boxes” attached to the base random word).

- **caps ?** Capitalize the word – or don’t – and that randomly (each strategy is equally likely and independent of the previous pick).

You password at this stage may be:

”Troubador”

and the corresponding entropy:

```
>>> log2(67502) + 1
17.042642627599378
```

- **common substitutions.** Let’s pretend that there is a flavor of the “leetspeak” language (see <http://en.wikipedia.org/wiki/Leet>) where every letter is uniquely represented by an alternate symbol that is unique and *not* a letter. For example a → 4, b → 8, c → (, e → 3, l → 1, o → 0, i → !, t → 7, x → %, etc. so that leet for example is represented as 1337. Replace letters in position 3, 6 and 8 by their corresponding leetspeak number – or don’t do it – randomly, the two options being equally likely and independent of the previous steps.

You password at this stage may be:

”Tr0ub4dor”

and the corresponding entropy:

```
>>> log2(67502) + 1 + 3
20.042642627599378
```

- **punctuation and numeral.** Pick at random a non-letter, non-digit symbol among printable characters in the US-ASCII set. There are 127 US-ASCII characters, 95 of them are printable (codes 0x20 to 0x7E, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters), that makes 95 minus 10 digits minus 2×26 letters – lower and upper case – that is 33 possible symbols. Add a random digit symbol and optionally swap the punctuation and digit symbols at random.

You password at this stage may be:

”Tr0ub4dor&3”

and the corresponding entropy:

```
>>> log2(67502) + 1 + 3 + log2(33) + log2(10) + 1
29.408964841845194
```

1.2.2 Alphabets, Symbol Codes, Stream Codes

An **alphabet** is a countable set of distinct symbols, meant to represent information. For example, the letters ”a” to ”z”, all unicode glyphs, the words of the English language, the non-negative integers, the binary values 0 and 1, etc.

Let \mathcal{A} be an alphabet and $n \in \mathbb{N}$. We denote \mathcal{A}^n the set of all n -uples with elements in \mathcal{A} , \mathcal{A}^+ the set of all such non-empty n -uples when n varies and \mathcal{A}^* the set of all such n -uples, including the (empty) 0-uple denoted ϵ .

$$\mathcal{A}^n = \overbrace{\mathcal{A} \times \cdots \times \mathcal{A}}^{n \text{ terms}}, \quad \mathcal{A}^+ = \bigcup_{n=1}^{+\infty} \mathcal{A}^n, \quad \mathcal{A}^* = \{\epsilon\} \cup \mathcal{A}^+ \quad (1.4)$$

Elements of \mathcal{A}^n are most of the time denoted without the commas and parentheses, by simple juxtaposition of the symbols of the tuple as in

$$a_0 a_1 \cdots a_{n-1} \in \mathcal{A}^n \quad (1.5)$$

We also talk about **sequences** and **(finite) streams** when we refer to such n -uples. The **length** of a stream $a_0 \cdots a_n$ is the number of symbols that it contains:

$$|a_0 \cdots a_{n-1}| = n \quad (1.6)$$

A **(variable-length) (binary) symbol code** c is a mapping from an alphabet \mathcal{A} to the set of non-empty finite binary streams $\{0, 1\}^+$. The code is of fixed length n if $c(a)$ has length n for any $a \in \mathcal{A}$. We usually require the code to be injective, sometimes using the term **non-ambiguous code** to emphasize this property. The elements of $\{0, 1\}^+$ are named **(binary) codes**, and **valid codes** if they belong to the range of c . Such mappings are characterized by:

$$c : \mathcal{A} \rightarrow \{0, 1\}^+ \text{ such that } c(a) = c(b) \implies a = b \quad (1.7)$$

The non-ambiguity assumption makes the code uniquely decodable: given $c(a)$, a is identified uniquely and may be recovered. Symbol codes are usually defined to form **stream codes**: the mapping c is extended from \mathcal{A} to \mathcal{A}^+ by

$$c(a_0 a_1 \cdots a_{n-1}) = c(a_0) c(a_1) \cdots c(a_{n-1}) \quad (1.8)$$

The resulting mapping is still a code – with symbols in the alphabet \mathcal{A}^+ instead of \mathcal{A} – only if the extended mapping is still non-ambiguous or bf self-delimiting, that is, the unique decodability is preserved by the extension to streams. In order for the stream code is non-ambiguous, the lengths of the valid codes have to satisfy the Kraft inequality:

$$K(c) = \sum_{a \in \mathcal{A}} 2^{-|c(a)|} \leq 1 \quad (1.9)$$

A converse statement also holds, see 1.2.1.

Proof. Let c be a symbol code for \mathcal{A} whose stream code is non-ambiguous. For any integer n , we have

$$K(c)^n = \sum_{a_0 \in \mathcal{A}} \cdots \sum_{a_{n-1} \in \mathcal{A}} 2^{-(|c(a_0)| + \cdots + |c(a_{n-1})|)} = \sum_{a \in \mathcal{A}^n} 2^{-|c(a)|}$$

Assume than \mathcal{A} is finite and let L be an upper bound for the code length of symbols in \mathcal{A} . For any integer l , there are at most 2^l distinct binary stream codes whose length is l and therefore

$$K(c)^n = \sum_{l=n}^{nL} \sum_{a \in \mathcal{A}^n, |c(a)|=l} 2^{-l} \leq \sum_{l=n}^{nL} 2^l \times 2^{-l} = n(L-1) + 1$$

Passing to the limit on n yields $K(c) \leq 1$. If \mathcal{A} is countable and $K(c) > 1$, there is a finite subset \mathcal{A}' of \mathcal{A} such that $c' = c|_{\mathcal{A}'}$ satisfies $K(c') > 1$. Hence, c' is ambiguous as a stream code and therefore c also is. \blacksquare

Prefix Codes

Unique decodability is necessary for a code to be useful. Still it does not make the code *easy* to use. Consider for example the alphabet $\mathcal{A} = \{0, 1, 2, 3\}$ and the mapping $\mathcal{A} \rightarrow \{0, 1\}^+$

$$0 \rightarrow 0, 1 \rightarrow 01, 2 \rightarrow 011, 3 \rightarrow 0111 \quad (1.10)$$

It is a code and we can convince ourselves that it is uniquely decodable. However decoding a stream requires lookahead: consider the stream that starts by 010110... and imagine a process that lets you discover the symbols one by one. At the first step, 0, you don't know what the original first symbol is: it could be 0, or it could be the partial code for any of the original symbols. So, you have to pick the second symbol, and you can rule out 0 but you still can't decide, the first symbol could be 1, or 2 or 3. Only the third symbol gives you the solution: the stream starts with 010 and it can be the case only if the first symbol was 1 (with code 01).

Given a possible symbol, here, with one step at most we can decide if a code is complete or partial but generally we can build uniquely decodable codes for which this limit is arbitrarily large. Consider on the contrary the code

$$0 \rightarrow 0, 1 \rightarrow 10, 2 \rightarrow 110, 3 \rightarrow 1110 \quad (1.11)$$

As soon as you receive a 0, the code is complete and you know that you can decode a symbol. Conversely, if you receive a 1, you know that you have to keep on reading the bit stream to decode a new symbol. So the decoding process is easy.

Codes that require no look-ahead and are therefore easy to decode are called **prefix codes**: there is no valid code, that, completed with extra bits, would form another valid code.

So prefix codes make streaming non-ambiguous *and* easy. However are they general enough? The answer is yes: if the Kraft inequality is satisfied for a code length function, there is non-ambiguous code that correspond to these lengths, and *this code can always be selected to be prefix code*. We'll prove this result later after we have studied the suitable representation of prefix codes.

Example of variable-length code: Unicode and utf-8.

Unicode is a computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems. The Unicode Consortium, the nonprofit organization that coordinates Unicode's development, has the ambitious goal of eventually replacing existing character encoding schemes with Unicode and its standard Unicode Transformation Format (UTF) schemes, as many of the existing schemes are limited in size and scope and are incompatible with multilingual environments. <http://en.wikipedia.org/wiki/Unicode>

Unicode was developed in conjunction with the Universal Character Set standard and published in book form as The Unicode Standard. As of 2011, the latest version of Unicode is 6.0 and consists of an alphabet of 249,031 characters or graphemes among 1,114,112 possible **code points**. Code points are designated by the symbol $U+X$ where X is the hexadecimal representation of an integer between 0 to 10ffff. Unicode can be implemented by different character encodings with the most commonly used encoding being UTF-8. UTF-8 uses one byte for any ASCII characters whose code point is between U+00 and U+7f, and up to four bytes for other characters. The coding

process is the following: first, if the code point is in the range U+0 – U+7f, the UTF-8 code is 0 followed by the code point binary representation. Otherwise, the code point binary representation requires more than 7 bits. The UTF-8 code then begins with the unary representation of the number of bytes used to represent the code point given that all bits of the first byte not used by the unary coding may be used and that beyond the first byte, every byte start with 10 which leaves 6 usable bits. That is, if the code point binary representation needs n bits, the number of bytes N uses by the UTF-8 encoding is

$$N = 1 \text{ if } n \leq 7, \left\lceil \frac{n-1}{5} \right\rceil \text{ if } 8 \leq n \leq 31. \quad (1.12)$$

The following table summarize the utf-8 code layout.

Range	Code Format
U+0 – U+7f	0xxxxxx
U+80 – U+7ff	110xxxxx 10xxxxxx
U+800 – U+ffff	1110xxxx 10xxxxxx 10xxxxxx
U+10000 – U+1fffff	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U+20000 – U+3fffffff	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U+4000000 – U+7fffffff	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

As an example, let's consider the code point U+2203 that represents the character ∃. It is an element of the mathematical operators whose range is U+2200–U+22ff (see <http://www.unicode.org/charts/PDF/U2200.pdf>). Being in the range U+0800 – U+ffff, it needs 3 bytes to be represented in utf-8. The binary representation of 2203 – as a hexadecimal number – is

00100010 00000011

We should fit those 16 bits into the $16 \times$ slots of the following pattern. No zero padding (adding zeros on the left) is necessary.

1110xxxx 10xxxxxx 10xxxxxx

So, we split the binary representation into

0010 001000 000011

and intertwine the result to obtain

11100010 10001000 10000011

or in hexadecimal

e2 88 83

This is confirmed by the output of the following PYTHON interactive session:

```
>>> exists = u"\u2203"
>>> print exists
∃
>>> exists.encode("utf-8")
'\xe2\x88\x83'
```

The utf-8 coding may be used in music format to store text information. But given that it really applies to integers, the code point interpretation is sometimes dropped entirely. For example, the FLAC format (<http://flac.sourceforge.net/format.html>) uses a “utf-8” coding to store sample number and frame number for variable block sizes.

Prefix Codes Representation

Prefix codes may seem to be hard to design. But as a matter of fact suitable representations of such structures make the process quite easy: binary trees and arithmetic representations are here the key structures.

Binary Trees. Binary trees are sets of nodes organised in a hierarchical structure where each node has at most two children. Formally, a **binary tree** is a pair $(\mathfrak{N}, \downarrow)$ where \mathfrak{N} is a set of elements called **nodes** and \downarrow is the **children mapping**, a partial application

$$\cdot \downarrow \cdot : \text{dom}(\downarrow) \subset \mathfrak{N} \times \{0, 1\} \rightarrow \mathfrak{N} \quad (1.13)$$

such that there exist a unique **root node** r

$$\exists ! r \in \mathfrak{N}, \forall n \in \mathfrak{N}, \forall i \in \{0, 1\}, r \neq n \downarrow i, \quad (1.14)$$

nodes are not shared

$$n \downarrow i = n' \downarrow i' \implies i = i' \wedge b = b' \quad (1.15)$$

and there is no cycle

$$n_0 \downarrow i_0 = n_1, \dots, n_{p-1} \downarrow i_{p-1} = n_p \implies (n_i = n_j \implies i = j) \quad (1.16)$$

A node n is **terminal** (or is a **leaf node**) if it has no children:

$$(n, 0) \notin \text{dom}(\downarrow) \text{ and } (n, 1) \notin \text{dom}(\downarrow) \quad (1.17)$$

The **depth** $|n|$ of a node $n \in \mathfrak{N}$ is the unique non-negative integer d defined by

$$\exists (b_0, \dots, b_{|d|-1}) \in \{0, 1\}^{|d|}, r \downarrow b_0 \downarrow \dots \downarrow b_{|d|-1} = n. \quad (1.18)$$

Codes are Binary Trees. Let \mathcal{A} be a *finite* alphabet and c a binary code on \mathcal{A} . Let \mathfrak{N} be the subset of $\{0, 1\}^*$ of all prefixes of valid codes, that is

$$\mathfrak{N} = \{b \in \{0, 1\}^*, \exists b' \in \{0, 1\}^*, \exists a \in \mathcal{A}, c(a) = bb'\} \quad (1.19)$$

We define the children mapping $\cdot \downarrow \cdot$ by:

$$b_0 \dots b_{n-1} \downarrow b_n = b_0 \dots b_{n-1} b_n \quad (1.20)$$

provided that $b_0 \dots b_{n-1}$ and $b_0 \dots b_{n-1} b_n$ both belong to \mathfrak{N} . With this definition, $(\mathfrak{N}, \downarrow)$ is a binary tree whose root is ϵ and the depth of a node is the length of the bit sequence so that the notation used in both contexts $|b_0 \dots b_{n-1}|$ is not ambiguous. We also define a partial **label mapping** S on \mathfrak{N} with values in \mathcal{A} by

$$S(b_0 \dots b_{n-1}) = a \text{ if } c(a) = b_0 \dots b_{n-1} \quad (1.21)$$

S is a one-to-one mapping from its domain to \mathcal{A} .

Conversely, given a tree $(\mathfrak{N}, \downarrow)$ and a one-to-one partial label mapping S with values in \mathcal{A} , we may define uniquely a code: let r be the root node ; for any $a \in \mathcal{A}$, there is a unique sequence $b_0 \dots b_{n-1}$ such that

$$S(r \downarrow b_0 \downarrow b_1 \downarrow \dots \downarrow b_{n-1}) = a, \quad (1.22)$$

so that we may set

$$c(a) = b_0 \cdots b_{n-1} \quad (1.23)$$

In other words, $c(a)$ is the path from the root node to the node whose label is a .

Binary trees used in this context are usually trimmed down so that all terminal nodes in \mathfrak{N} are labelled – belong to the domain of S ; extra nodes are useless to define the code c . We say that those trees are **compact** with respect to S and that $(\mathfrak{N}, \downarrow, S)$ is a (**compact, binary, labelled**) **tree representation** of the code c . Among such trees, the ones that correspond to prefix codes are easy to spot: only their terminal nodes are labelled.

Arithmetic representation. Let $b_0 \cdots b_{n-1} \in \{0, 1\}^n$. The (**binary**) **fraction representation** – denoted $0.b_0 \cdots b_{n-1}$ – of this bit sequence is

$$0.b_0 \cdots b_{n-1} = \sum_{i=0}^{n-1} b_i \times 2^{-i+1} \quad (1.24)$$

To any such bit sequence b we may associate an interval in $[0, 1]$ by:

$$b = b_0 \cdots b_{n-1} \rightarrow I(b) = [0.b_0 \cdots b_{n-1}, 0.b_0 \cdots b_{n-1}1] \quad (1.25)$$

We may notice that $I(b)$ is the only interval that contains all binary fractions (whose denominator is a power of two) whose first bits as a binary fraction are b_0, \dots, b_{n-1} . In other words, it contains the fraction representation of all codes of which $b_0 \cdots b_{n-1}$ is a prefix. As a consequence, for any prefix code c on \mathcal{A} the intervals $I(c(a))$, $a \in \mathcal{A}$ are non-overlapping:

$$\forall (a, a') \in \mathcal{A}^2, a \neq a' \implies I(a) \cap I(a') = \emptyset \quad (1.26)$$

This property is actually necessary and sufficient for codes to be prefix codes. We use that crucial property to proof the converse of Kraft's theorem, namely:

Proposition 1.2.1 *Given an alphabet \mathcal{A} and a list of code lengths (l_a) , $a \in \mathcal{A}$ such that*

$$K = \sum_{a \in \mathcal{A}} 2^{-l_a} \leq 1$$

there is a prefix code c such that

$$\forall a \in \mathcal{A}, |c(a)| = l_a$$

Proof. Let's order all symbols in \mathcal{A} in a sequence a_0, a_1, \dots . Given the lengths l_a , $a \in \mathcal{A}$, we define the sequence of binary fractions and intervals $x_{a_0} = 0$, $x_{a_n} = x_{a_{n-1}} + 2^{-l_{a_{n-1}}}$ and $I(a_n) = [x_{a_n}, x_{a_{n+1}}]$. At any step n of this process, $x_{a_n} = \sum_{i=0}^{n-1} 2^{-l_{a_i}} \leq 1$ because Kraft's inequality holds. We may therefore define the code c by

$$c(a_n) = b_0 \cdots b_{p-1} \text{ with } I(a_n) = [0.b_0 \cdots b_{p-1}, 0.b_0 \cdots b_{p-1}1] \quad (1.27)$$

By construction these intervals are non-overlapping and consequently what we have build is a prefix code. \blacksquare

1.2.3 Optimal Length Coding

Given an alphabet and random symbol in it, we explain in this section what bounds exist on codes average bit length and how we can build a code that is optimal for this criteria.

Average Bit Length – Bounds

Let \mathcal{A} be an alphabet, A a random symbol in \mathcal{A} and c a code on \mathcal{A} . We define the **(average) (bit) length** of the code as:

$$\mathbb{E}|c(A)| = \sum_{a \in \mathcal{A}} p(a)|c(a)| \quad (1.28)$$

where as usual $p(a) = P(A = a)$. Let $(\mathfrak{N}, \downarrow, S)$ is a tree representation of c and define the **weight** $w(n) = p(c^{-1}(n))$. The **weighted tree** $t = (\mathfrak{N}, \downarrow, S, w)$ has a **(weighted) depth** $|t|$ given by

$$|t| = \sum_{n \in \text{dom } S} w(n)|n| \quad (1.29)$$

that is equal to $\mathbb{E}|c(A)|$.

We know from the previous section that we can restrict our search to prefix codes. For such codes, the entropy $H(A)$ provides bounds on the expected length. Specifically, we have:

Proposition 1.2.2 *Every prefix code c for \mathcal{A} satisfies*

$$H(A) \leq \mathbb{E}|c(A)|. \quad (1.30)$$

Moreover there exist a prefix code c for \mathcal{A} such that

$$\mathbb{E}|c(A)| < H(A) + 1. \quad (1.31)$$

The inequality (1.30) may usefully be detailed. First, to the code c we have to associate a set of **implicit probabilities** $q(a)$, $a \in \mathcal{A}$, by

$$q(a) \propto 2^{-|c(a)|}, \quad \sum_{a \in \mathcal{A}} q(a) = 1 \quad (1.32)$$

Let $p(a) = P(A = a)$. We define the **(Kullback-Leibler) divergence** of the probability distributions p and q by:

$$D_{\text{KL}}(p||q) = \sum_{a \in \mathcal{A}} p(a) \log_2 \frac{p(a)}{q(a)} \geq 0 \quad (1.33)$$

with the two conventions that if there is a $a \in \mathcal{A}$ such that $q(a) = 0$ and $p(a) \neq 0$ then $D_{\text{KL}}(p||q) = +\infty$ and that $p(a) \log_2 \frac{p(a)}{q(a)} = 0$ when $p(a) = 0$. The positivity of the divergence is called the **Gibbs inequality**. It results from a convexity argument⁴ and moreover

$$D_{\text{KL}}(p||q) = 0 \text{ iff } \forall a \in \mathcal{A}, p(a) = q(a) \quad (1.34)$$

⁴The function $x \mapsto \log_2 x$ being strictly concave, we have

$$D_{\text{KL}}(p||q) = \sum_{a \in \mathcal{A}, p(a) \neq 0} p(a) \left(-\log_2 \frac{q(a)}{p(a)} \right) \geq -\log_2 \left(\sum_{a \in \mathcal{A}, p(a) \neq 0} p(a) \frac{q(a)}{p(a)} \right) = 0$$

with equality only when $p(a) = q(a)$ for every a .

These terms being given, we have the equality (proved below)

$$H(A) = \mathbb{E}|c(A)| - D_{\text{KL}}(p||q) - \log_2 K(c) \quad (1.35)$$

where $K(c) \leq 1$ is defined in the Kraft inequality (1.9). The equality case in the inequality (1.30) happens if and only if

$$\forall a \in \mathcal{A}, |c(a)| = -\log_2 p(a) \quad (1.36)$$

a condition for which we may find a code c if and only if the probability p satisfies

$$\forall a \in \mathcal{A}, p(a) \in 2^{-\mathbb{N}} \quad (1.37)$$

Proof. The mean code length satisfies

$$\mathbb{E}|c(A)| = \sum_{a \in \mathcal{A}} p(a)|c(a)| = - \sum_{a \in \mathcal{A}} p(a) \log_2 2^{-|c(a)|}$$

Given the definition of the implicit probability

$$q(a) = \frac{2^{-|c(a)|}}{\sum_{a \in \mathcal{A}} 2^{-|c(a)|}} = \frac{2^{-|c(a)|}}{K(c)},$$

we end up with

$$\begin{aligned} \mathbb{E}|c(A)| &= - \sum_{a \in \mathcal{A}} p(a) \log_2 q(a) - \log_2 K(c) \\ &= - \sum_{a \in \mathcal{A}} p(a) \log_2 p(a) - \sum_{a \in \mathcal{A}} p(a) \log_2 \frac{q(a)}{p(a)} - \log_2 K(c) \\ &= H(A) + D_{\text{KL}}(p||q) - \log_2 K(c) \geq H(A) \end{aligned}$$

For any $a \in \mathcal{A}$, we define $l(a) = \lceil -\log_2 p(a) \rceil$. As we have

$$\sum_{a \in \mathcal{A}} 2^{-l(a)} \leq \sum_{a \in \mathcal{A}} 2^{\log_2 p(a)} = \sum_{a \in \mathcal{A}} p(a) = 1$$

there is a prefix code c that satisfies $|c(a)| = l(a)$ for any $a \in \mathcal{A}$. Moreover

$$\mathbb{E}|c(A)| = \sum_{a \in \mathcal{A}} p(a)l(a) < \sum_{a \in \mathcal{A}} p(a)(-\log_2 p(a) + 1) = H(A) + 1.$$

■

Algorithm and Implementation of Huffman Coding

We represent (finite) alphabets and the probability distribution of a random symbol as a single Python dictionary, for example

$$\mathcal{A} = \{'a', 'b', 'c'\} \text{ and } p('a') = 0.5, p('b') = 0.3, p('c') = 0.2$$

as

```
{'a': 0.5, 'b': 0.3, 'c': 0.2}
```

Probabilities will be used relative **weights**: instead we could define the dictionary as

```
{'a': 5, 'b': 3, 'c': 2}
```

and the code resulting from the Huffman algorithm would be the same. The algorithm we implement uses **weighted binary trees** where:

- terminal nodes are symbol/weight pairs such as ('a', 1.0),
- non-terminal nodes are children/weight pairs where children is a list of (terminal or non-terminal) nodes, such as [(('c': 0.2), ('a': 0.5)], 0.7).

Handling of these structures is made through a small set of helper functions:

```
class Node(object):
    "Function helpers to manage nodes as (symbol, weight) pairs"

    @staticmethod
    def symbol(node):
        return node[0]

    @staticmethod
    def weight(node):
        return node[1]

    @staticmethod
    def is_terminal(node):
        return not isinstance(Node.symbol(node), list)
```

The **Huffman algorithm** is the following: we create a list of symbol/weight nodes from the initial dictionary and repeat the following steps:

1. pick up two nodes with the least weights, remove them from the list,
2. insert into the list a non-terminal nodes with those two nodes as children and a weight that is the sum of their weights,,
3. stop when there is a single node in the list: this is the root of the binary tree

Note that this algorithm is not entirely deterministic if at any step, the lowest weight correspond to three nodes or more.

Here is the corresponding implementation in `make_binary_tree`; note that in the other methods of the `huffman` class, we also create a symbol to code dictionary (`self.table`) from the binary tree to simplify the coding and decoding of symbols – not all functions are given in this chunk of code ; the `symbol_encoder` and `symbol_decoder` create (symbol) coders and decoders from the symbol to code tables and `stream_encoder` and `stream_decoder` create stream coder and encoder from those.

```
class huffman(object):
    def __init__(self, weighted_alphabet):
        self.weighted_alphabet = weighted_alphabet
        self.tree = huffman.make_binary_tree(weighted_alphabet)
        self.table = huffman.make_table(self.tree)
        self.encoder = stream_encoder(symbol_encoder(self.table))
        self.decoder = stream_decoder(symbol_decoder(self.table))

    @staticmethod
```

```

def make_binary_tree(weighted_alphabet):
    nodes = weighted_alphabet.items()
    while len(nodes) > 1:
        nodes.sort(key=Node.weight)
        node1, node2 = nodes.pop(0), nodes.pop(0)
        node = ([node1, node2], Node.weight(node1) + Node.weight(node2))
        nodes.insert(0, node)
    return nodes[0]

@staticmethod
def make_table(root, table=None, prefix=None):
    if prefix is None:
        prefix = BitStream()
    if table is None:
        table = {}
    if not Node.is_terminal(root):
        for index in (0, 1):
            new_prefix = prefix.copy()
            new_prefix.write(bool(index))
            new_root = Node.symbol(root)[index]
            huffman.make_table(new_root, table, new_prefix)
    else:
        table[Node.symbol(root)] = prefix
    return table

```

As usual, the coder and decoder are registered for use with the `BitStream` instances.

```
bitstream.register(huffman, reader=lambda h: h.decoder, writer=lambda h: h.encoder)
```

Optimality of the Huffman algorithm

We prove in this section that the Huffman algorithm creates a code whose average bit length is minimal among all prefix codes (and therefore all non-ambiguous stream codes).

Switching nodes. Consider a weighted tree $t = (\mathfrak{N}, \downarrow, S, w)$ and two of its terminal nodes n_1 and n_2 . Switch the position of these nodes and call the resulting tree t' . A simple computation shows that the weighted bit length of t' is given by

$$|t'| = |t| + (|n_1|_t - |n_2|_t)(w(n_2) - w(n_1)) \quad (1.38)$$

Consequently, a tree that has a terminal node with a high probability at a depth larger than another node with low probability higher in the tree can't be optimal, because switching the two nodes would decrease the code average bit length. The same formula also shows that given two terminal nodes n_1 and n_2 among the ones with the lowest probability, we can always find an optimal tree with them as siblings at the greatest depth: at least one optimal tree exist for combinatorial reasons ; for such a tree, either a node is a leaf, or it has two children nodes; now get at the bottom of the tree: there are two terminal nodes. If they are not the desired nodes, switches that are neutral to the average bit length will create a new optimal tree with the desired property.

Length optimality. The proof that the Huffman coding algorithm is optimal is made by induction on the number of symbols. The check for the basis assumption is straightforward. Now assume that the result holds for N symbols and pick an alphabet with $N + 1$ symbols. Let t be the tree that results from the application of the Huffman algorithm. Now consider the initial set of symbols and replace the two with the lowest weight (nodes n_1 and n_2) with a single one with a weight equal to the sum of the original symbol weights. Let t' be the result of the Huffman algorithm

on these N symbols. The average bit length of both trees are related by $|t'| = |t| + (w(n_1) + w(n_2))(|n_1|_t - 1) - w(n_1)|n_1|_t - w(n_2)|n_2|_t = |t| - w(n_1) - w(n_2)$. Now consider an optimal rearrangement of t_* of t that would have n_1 and n_2 as siblings at the lowest level (possible by the previous section). Grouping n_1 and n_2 leads to a t'_* whose length is $|t'_*| = |t_*| - w(n_1) - w(n_2)$ and such t'_* is a rearrangement of t' . Hence, as $|t| = |t_*| - (|t'_*| - |t'|)$ and because by the induction hypothesis t' is optimal, $|t'_*| - |t'| \geq 0$, we have $|t| \leq |t^*|$ and t is optimal.

Example: Brainfuck, Spoon and Fork.

Brainfuck⁵ is an eight-instruction Turing-complete programming language. The eight instructions are represented by the characters:

```
> < + - . , [ ]
```

The language is organised around a single byte pointer p . As an exemple, the instruction ‘+’ corresponds to the C code fragment “ $(*p)++;$ ” and ‘.’ to “ $\text{putchar}(*p);$ ”. The classic “Hello World!” program in Brainfuck is the sequence:

```
++++++++[>++++++>++++++>+++><<<-]
>++.>+.+++++.+++.>+.<<+++++++
>.+++.-----.-----.>+.>.
```

Spoon⁶ is a variant of Brainfuck created in 1998 by Steven Goodwin. It uses binary encoding of the eight original symbols according to the mapping

$>$	\rightarrow	010	$ $	$<$	\rightarrow	011	$ $	$+$	\rightarrow	1	$ $	$-$	\rightarrow	000
.	\rightarrow	001010	$ $,	\rightarrow	0010110	$ $	[\rightarrow	00100	$ $]	\rightarrow	0011

The code is a prefix code that was actually designed using Huffman encoding, with probabilities of each command computed from a set of example programs. You may notice that the code appears NOT to be optimal as ‘,’ has the longest code and has no sibling: it could be shortened to 001011. But that’s only because the *actual* Spoon variant of Brainfuck introduces two extra reserved codes, 00101110 to dump the memory (for debug purposes) and 00101111 to stop execution. The full binary tree of the code – where the corresponding symbols are * and ! – and the arithmetic representation are displayed in figure 1.6.

The Spoon encoding of the “Hello World!” program is a sequence of 245 bits:

```
11111111100100010111111010111111111010111010101
10110110110000110101100101001010010101111100101
000101011100101001011001010011011111111111111100
10100100010101110010100000000000000000000000010100000
00000000000000000000000000000000101001010010001010.
```

As the commands are exactly 8, it is also tempting to define a 3-bit fixed-length encoding of the original symbols and let’s call it Fork ! We adopt the mapping:

$>$	\rightarrow	000	$ $	$<$	\rightarrow	001	$ $	$+$	\rightarrow	010	$ $	$-$	\rightarrow	011
.	\rightarrow	100	$ $,	\rightarrow	101	$ $	[\rightarrow	110	$ $]	\rightarrow	111

⁵<http://www.muppetlabs.com/~breadbox/bf>

⁶<http://web.archive.org/web/20100226062821/http://progopedia.com/dialect/spoon/>

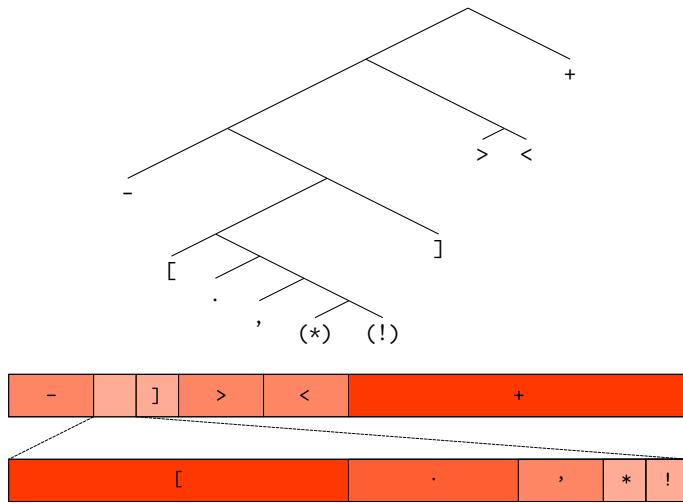


Figure 1.6: Spoon coding tree and arithmetic display.

The encoding with Fork of the “Hello World!” program is

```

01001001001001001001001001001001001001001001001001001001001001001
00100100000100100100100100100100100100100000100100100100000100100
1000001000100100100101111000010010100000010100010
0100100100100100100100100100100100100100100100100000
10010100100100100100100100100100100100100100100100101
00000100010010010001101101101101101100011011011
0110110110110110000010100000100,

```

a stream of 333 binary digits, about 36 % less space-efficient than spoon.

1.2.4 Golomb-Rice Coding

Optimality of Unary Coding – Reduced Sources.

Consider the alphabet \mathbb{N} of the non-negative integers and a random symbol n in \mathbb{N} such that for any integer i

$$P(n = i) = P(n > i) \quad (1.39)$$

Normalization of this probability distribution $p(i) = P(n = i)$ yields

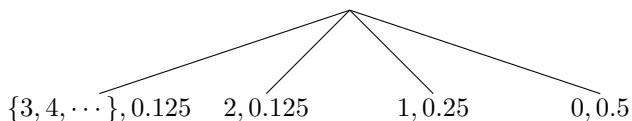
$$\forall i \in \mathbb{N}, p(i) = 2^{-i-1} \quad (1.40)$$

What is the optimal coding of such a random symbol? An infinite number of symbols have a non-zero probability, therefore we can't apply directly Huffman coding. However, we can get a flavor of what the optimal coding can be by reducing the random symbol to a finite alphabet. To do so, we select a threshold m and group together all the outcomes of n greater or equal to m ; the new random symbol based on n has values in the finite alphabet whose symbols are $0, 1, 2, \dots, m-1$ and the set $\{m, m+1, \dots\}$ of values above m . The probability of the m first m symbols i is

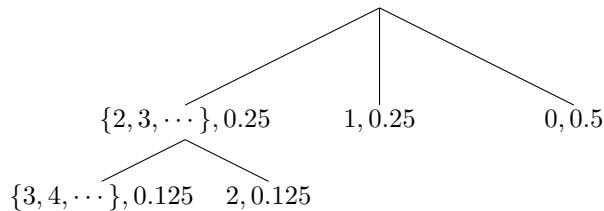
$p_m(i) = P(n = i) = 2^{-i-1}$ and the probability of the last one is

$$p_m(i) = P(n \geq m) = \sum_{i=m}^{+\infty} P(n = i) = \sum_{i=m}^{+\infty} 2^{-i-1} = 2^{-m}$$

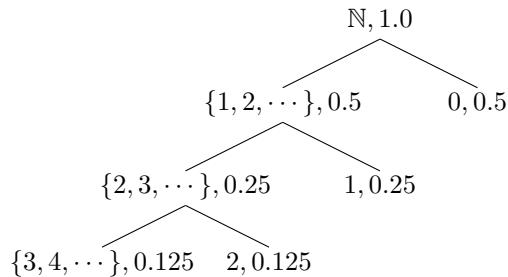
Let's apply the Huffman algorithm to the resulting symbol, say for $m = 2$. Here are the steps that build the Huffman binary tree: the initial list of symbols, sorted by increasing probability is:



The nodes $\{3, 4, \dots\}$ and 2 have the lowest probability of occurrence and therefore shall be grouped. But their probability is the same – 0.125 – and we need to make the a decision about which one will be taken as the first node of the group. We decide to go for the infinite set of symbols $\{3, 4, \dots\}$ first. Somehow it feels the right thing to do, to have the only compound symbol on the left and all the others on the right ...



This group has a cumulative probability of 0.25, the same as the symbol 1. There is only one extra symbol, 0 and its probability is higher, therefore no reordering is necessary. We therefore group $\{2, 3, \dots\}$ (first) and 1.



There are only two symbols left, so the algorithm has completed. Let's look at the results in terms of coding: we ended up with the code

$$0 \rightarrow 1, 1 \rightarrow 01, 2 \rightarrow 001, \{3, 4, \dots\} \rightarrow 000$$

From this we can guess what the optimal coding is for the original source, without the reduction attached to the threshold : it is the unary coding of the integer, or more precisely here, the variant that uses 0 for the length and 1 as an end symbol.

$$0 \rightarrow 1, 1 \rightarrow 01, 2 \rightarrow 001, 3 \rightarrow 0001, 4 \rightarrow 00001, 5 \rightarrow 000001, \dots$$

The choice of the variant has no impact on the optimality of the code: unary coding is the optimal solution for the coding of non-negative integers that occur with a probability $p(i) = 2^{-i-1}$. Here is an implementation of unary coding of integer symbols that uses 0 as an end delimiter instead:

```
def unary_symbol_encoder(stream, symbol):
    return stream.write(symbol * [True] + [False], bool)

def unary_symbol_decoder(stream):
    count = 0
    while stream.read(bool) is True:
        count += 1
    return count
```

Given those symbol encoder and decoder function, the higher-order function `stream_encoder` and `stream_decoder` from the `coding` module generate a stream encoder and decoder respectively.

```
unary_encoder = stream_encoder(unary_symbol_encoder)
unary_decoder = stream_decoder(unary_symbol_decoder)
```

Finally, we define an empty class `unary` as a type used to register the encoder and decoder in the `bitstream` module.

```
class unary(object):
    pass
bitstream.register(unary, reader=unary_decoder, writer=unary_encoder)
```

After that last step, using unary coding and decoding is as simple as:

```
>>> stream = BitStream()
>>> stream.write([0,1,2,3], unary)
>>> print stream
0101101110
>>> stream.read(unary, 4)
[0, 1, 2, 3]
```

Optimal Coding of Geometric Distribution - Rice Coding

The same method of source reduction may be applied to analyze optimal code for more complex distribution, for example the **(one-sided) geometric distribution**. Such a distribution is defined by $p(i) \propto \theta^i$ for a $\theta \in (0, 1)$; note that the distribution of the previous section was the special case of $\theta = 0.5$. The normalization of this distribution leads to

$$p(i) = (1 - \theta)\theta^i \quad (1.41)$$

The method of reduced source illustrated in the previous section may be used to derive an optimal coding in the general case. Let's summarize the findings of [GvV75]; first, let l be the unique integer such that

$$\theta^l + \theta^{l+1} \leq 1 < \theta^{l-1} + \theta^l. \quad (1.42)$$

The optimal coding of the non-negative integer i is made of two codes:

- the unary coding of $\lfloor i/l \rfloor$,
- the Huffman coding of $i \bmod l$.

The probability distribution needed to perform the second part of the encoding is:

$$P(i \bmod l) = \frac{(1 - \theta)\theta^i}{1 - \theta^l}.$$

Note that this distribution is quite flat with respect to the original one. For that reason, the length of the optimal coding are almost constant for the Huffman part. Precisely, the length of the coding of $0 \leq i < l$ is $\lfloor \log_2 l \rfloor$ (if $i < 2^{\lfloor \log_2 l \rfloor} - l$) or $\lfloor \log_2 l \rfloor + 1$ (otherwise). In particular, if l is a power of two, we have $2^{\lfloor \log_2 l \rfloor} - l = l$ is therefore every $0 \leq i < l$ is coded with the same length of $\log_2 l$.

Rice Coding

Rice coding (or **Golomb-Power-Of-Two (GPO2) coding**) uses this remark to simplify the coding at the price of a usually negligible suboptimality: instead of the “true” integer l , solution of (1.42), we select an approximation of it $l' = 2^n$ that is a power of two, then perform the coding using this value instead of l . As a consequence, the Huffman coding of the second part is replaced by a fixed-length encoding of an integer on n bits.

The remaining issue is to determine a good selection of the **Golomb parameter** n ; this issue is described in details in [Kie04]. Initially, we need an estimate of θ – that is a priori unknown – from experimental data; we can usually derive it from the mean m of the available values : as the expectation of a one-sided geometric random variable with parameter θ has an expectation of $\theta/(1 - \theta)$, it makes sense to select

$$\theta = \frac{m}{1 + m} \tag{1.43}$$

Given the golden ratio

$$\phi = \frac{1 + \sqrt{5}}{2}, \tag{1.44}$$

we select as the number of bits dedicated to the fixed-length coding the value:

$$n = \max \left[0, 1 + \left\lfloor \log_2 \left(\frac{\log(\phi - 1)}{\log \theta} \right) \right\rfloor \right]. \tag{1.45}$$

Implementation

We begin with the definition of a `rice` class that holds the parameters of a given Rice encoding and also provide a method for the selection of the optimal parameter. The use of this method is optional: given that Rice coding is very often applied to distributions of integers that are not geometric, there is no guarantee in general that the Golomb parameter of this method will be the most efficient selection.

The parameter `n` in the `rice` constructor is the Golomb parameter. The optional `signed` option may be used to enable the coding of negative integers.

```
class rice(object):
    def __init__(self, n, signed=False):
        self.n = n
        self.signed = signed

    @staticmethod
    def select_parameter(mean):
        golden_ratio = 0.5 * (1.0 + numpy.sqrt(5))
```

```

theta = mean / (mean + 1.0)
log_ratio = log(golden_ratio - 1.0) / log(theta)
return int(maximum(0, 1 + floor(log2(log_ratio))))

```

The following encoder and decoder implementations demonstrate how we deal with signed values: by prefixing the code stream with a bit sign (0 for +, 1 for −) before we encode the absolute value of the integer. More complex schemes – that intertwine negative and positive values – are possible and may be useful to deal with two-sided geometric distribution that are not centered around 0 (see for example [MSW00]). We then encode the fixed-length part of the code and follow with the unary code.

In the following code, `options` is meant to be an instance of the `rice` class, `stream` is an instance of `BitStream` and `symbol` an integer.

```

def rice_symbol_encoder(options):
    def encoder(stream, symbol):
        if options.signed:
            stream.write(symbol < 0)
        symbol = abs(symbol)
        remain, fixed = divmod(symbol, 2 ** options.n)
        fixed_bits = []
        for _ in range(options.n):
            fixed_bits.insert(0, bool(fixed % 2))
            fixed = fixed // 2
        stream.write(fixed_bits)
        stream.write(remain, unary)
    return encoder

def rice_symbol_decoder(options):
    def decoder(stream):
        if options.signed and stream.read(bool):
            sign = -1
        else:
            sign = 1
        fixed_number = 0
        for _ in range(options.n):
            fixed_number = (fixed_number << 1) + int(stream.read(bool))
        remain_number = 2 ** options.n * stream.read(unary)
        return sign * (fixed_number + remain_number)
    return decoder

```

Finally, the `rice` class, its encoder and decoder are registered for integration with the `BitStream` instances.

```

rice_encoder = lambda r: stream_encoder(rice_symbol_encoder(r))
rice_decoder = lambda r: stream_decoder(rice_symbol_decoder(r))
bitstream.register(rice, reader=rice_decoder, writer=rice_encoder)

```

The following PYTHON session demonstrates the basic usage:

```

>>> data = [0, 8, 0, 8, 16, 0, 32, 0, 16, 8, 0, 8]
>>> rice.select_parameter(mean(data))
3
>>> stream = BitStream()
>>> stream.write(data, rice(3))
>>> stream
000000010000000010000110000000011110000000011000010000000010
>>> stream.read(rice(3), 12)
[0, 8, 0, 8, 16, 0, 32, 0, 16, 8, 0, 8]

```

On this particular data, the selection of the Golomb parameter was effective if we consider the following test of possible parameter values between 0 (unary coding) and 6.

```
>>> for i in range(7):
...     stream = BitStream(data, rice(i))
...     print "rice n={0}: {1} bits".format(i, len(stream))
...
rice n=0: 108 bits
rice n=1: 72 bits
rice n=2: 60 bits
rice n=3: 60 bits
rice n=4: 64 bits
rice n=5: 73 bits
rice n=6: 84 bits
```

Chapter 2

Quantization

Quantization is a process that maps a continuous or discrete set of values into approximations that belong to a smaller set. Quantization is a lossy: some information about the original data is lost in the process. The key to a successful quantization is therefore the selection of an error criterion – such as entropy and signal-to-noise ratio – and the development of optimal quantizers for this criterion.

2.1 Principles of Scalar Quantization

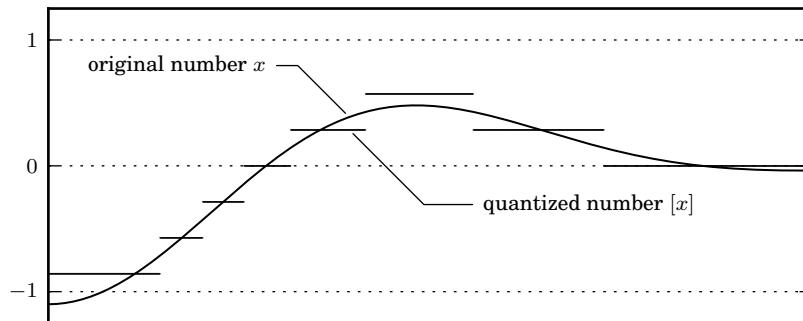


Figure 2.1: quantization of a time-varying value by a 4-bit midtread uniform quantization on $[-1.0, 1.0]$

2.1.1 Quantizers

A scalar quantizer $[\cdot]$ is an idempotent mapping from \mathbb{R} to a countable subset of \mathbb{R} :

$$|\{[x], x \in \mathbb{R}\}| \leq |\mathbb{N}| \text{ and } \forall x \in \mathbb{R}, [[x]] = [x] \quad (2.1)$$

This definition should be taken with a grain of salt as variants of the real line are often used, including the extended real line $\mathbb{R} \cup \{-\infty, +\infty\}$, the real line with signed

zeros $\mathbb{R} \cup \{0^-, 0^+\}$, the real line plus the undefined symbol \perp , or a combination thereof.

The countability assumption is what makes the quantizer useful as an attempt to approximate a continuous value by a discrete set that can be encoded as an integer. A quantizer is meant to be split into a **forward** and **inverse quantizer**: the forward quantizer builds from x an integer code that refers to $[x]$ without ambiguity and the inverse quantizer builds the approximation $[x]$ back from the code.

Formally, a forward quantizer for $[\cdot]$ is a mapping $i[\cdot] : \mathbb{R} \rightarrow \mathbb{Z}$ such that $[x] = [y]$ implies $i[x] = i[y]$. Because of this property, $i[\cdot]$ may be factored into $i[\cdot] = i \circ [\cdot]$ where $i : \text{ran}[\cdot] \rightarrow \mathbb{Z}$. The notation for the forward quantizer is therefore consistent with the use as $f[x]$ as a shortcut for $f([x])$. The associated inverse quantizer, denoted i^{-1} , is a left inverse of i : a mapping whose domain is a subset of \mathbb{Z} that contains $\text{ran } i$ and such that

$$\forall x \in \mathbb{R}, (i^{-1} \circ i)[x] = [x] \quad (2.2)$$

The first step of this quantizer composition partitions the real line into the family of sets $(I_n)_n$ with

$$I_n = \{x \in \mathbb{R}, i[x] = n\}, n \in \text{ran } i$$

The second step associates to any set into this partition a unique representative element. In every practical case we will encounter, the sets I_n are – possibly unbounded – intervals, either open, half-open or closed. In this context, we associate to x the decision values $[x]^-$ and $[x]^+$ to be

$$[x]^- = \inf \{y \in \mathbb{R}, [x] = [y]\} \text{ and } [x]^+ = \sup \{y \in \mathbb{R}, [x] = [y]\} \quad (2.3)$$

and the step of the quantization at point x is

$$\Delta(x) = [x]^+ - [x]^- \quad (2.4)$$

EXAMPLE - integer rounding. The floor function $\lfloor \cdot \rfloor$ is a scalar quantizer that maps a real number to the largest previous integer:

$$\forall x \in \mathbb{R}, \lfloor x \rfloor \in \mathbb{Z} \text{ and } \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 \quad (2.5)$$

A natural forward quantizer for $\lfloor \cdot \rfloor$ is ... itself! The identity $n \mapsto n$ is the corresponding inverse mapping. This quantizer partitions the real-line into the half-open intervals $I_n = [n, n + 1)$ for any $i \in \mathbb{Z}$.

The `floor` function of NumPy is an finite-precision implementation of this function. Its argument and return value are (arrays of) 64-bits floating-point numbers.

To obtain a (forward) quantizer with a finite range indexable on 32 bits, we may modify the initial quantizer specification so that the data outside of the range $[-2^{31}, 2^{31} - 1]$ – the range of 32-bit signed integers – is clipped:

$$\lfloor x \rfloor_{32} = \begin{cases} -2^{31} & \text{if } x \leq -2^{31} \\ 2^{31} - 1 & \text{if } x \geq 2^{31} - 1 \\ \lfloor x \rfloor & \text{otherwise.} \end{cases}$$

Given those modifications, a suitable finite implementation of the forward and inverse quantizers is the following code/decode pair:

```
from numpy import *

def encode(x):
    n = floor(x)
    n = clip(n, -2**31, 2**31 - 1)
    return int32(n)

def decode(n):
    return float64(n)

def quantize(x):
    return decode(encode(x))
```

The step function Δ of this quantization is defined by:

$$\Delta(x) = \begin{cases} +\infty & \text{if } x < -2^{31} + 1 \\ 1 & \text{if } -2^{31} + 1 \leq x < 2^{31} - 1 \\ +\infty & \text{if } 2^{31} - 1 \leq x \end{cases}$$

Other rounding functions may serve as the basis for similar schemes: the ceiling function $\lceil \cdot \rceil$ (NumPy function ceil) defined by:

$$\forall x \in \mathbb{R}, \lceil x \rceil \in \mathbb{Z} \text{ and } \lceil x \rceil - 1 < x \leq \lceil x \rceil \quad (2.6)$$

Instead of selecting the lower or upper integer approximation of x we may also select the nearest:

$$\forall x \in \mathbb{R}, |x - \lceil x \rceil| = \min \{|x - n|, n \in \mathbb{Z}\} \quad (2.7)$$

The value $\lceil x \rceil$ is not defined by this relation when $x = n + 1/2$, n being an integer. The NumPy function round_ rounds for example such real number to the nearest even integer.

This example suggests a general interface for quantizers. Such objects would provide an encode method for the forward quantization, a decode method for the inverse quantization and would be callable so that quantizer(x) would apply both steps to the data x . Such objects could inherit the following Quantizer base class:

```
class Quantizer(object):
    "Quantizers Base Class."
    def encode(self, data):
        raise NotImplementedError("undefined forward quantizer")

    def decode(self, data):
        raise NotImplementedError("undefined inverse quantizer")

    def __call__(self, data):
        return self.decode(self.encode(data))
```

We can then rewrite the above integer approximation quantizer as:

```
class RoundingQuantizer(Quantizer):
    def __init__(self, rounding=floor, integer_type=int32):
        self.rounding = rounding
        self.integer_type

    def encode(self, x):
        x = array(x)
        n = self.rounding(x)
        n = clip(n, -2**31, 2**31 - 1)
        return n.astype(self.integer_type)
```

```

def decode(self, n):
    n = array(n)
    return n.astype(float64)

rounding_quantizer = RoundingQuantizer()

```

Note that this version of the quantizer is also vectorized: several values grouped in a NumPy array may be used as arguments to `encode` and `decode`. This is an implicit requirement that we expect all quantizer classes to follow for convenience.

2.1.2 Uniform Quantization

A quantizer is uniform in an interval with lower bound a and higher bound b if its step function is constant in the interval. The size of the step is then directly connected to the width of the interval and the number N of distinct values of $[x]$ by

$$\Delta(x) = \frac{b - a}{N} \quad (2.8)$$

The final option that characterizes the quantizer is the choice of the base rounding function. A reference implementation is then given by:

```

class Uniform(Quantizer):
    def __init__(self, low=0.0, high=1.0, N=2**8, rounding=round_):
        self.low = float(low)
        self.high = float(high)
        self.N = N
        self.delta = (high - low) / self.N
        self.rounding = rounding

    def encode(self, data):
        low, high, delta = self.low, self.high, self.delta
        data = clip(data, low + delta/2.0, high - delta/2)
        flints = self.rounding((data - low) / delta - 0.5)
        return array(flints, dtype=long)

    def decode(self, i):
        return self.low + (i + 0.5) * self.delta

```

Note that if the default value of N is selected – or more generally any even value – $[0] \neq 0$: the approximation error for 0 is not zero. When this property may be an issue, odd values of N may be selected – for example $2^8 - 1$ so that 0 is correctly approximated ; such a quantizer is called a **midtread** quantizer – opposed to the original **midrise** quantizer.

2.1.3 Quantization of Random Variables

Consider a random variable X with values $x \in \mathbb{R}$ and a density of probability $p(x)$. For any $[x]$, we may consider the event $[X] = [x]$ whose probability is given by

$$P([X] = [x]) = \int_{\{y \in \mathbb{R}, [y]=[x]\}} p(y) dy = \int_{[x]^-}^{[x]^+} p(y) dy \quad (2.9)$$

If the density p is constant on every interval associated to the quantization, this equation may be simplified into:

$$P([X] = [x]) = p(x) \times \Delta(x) \quad (2.10)$$

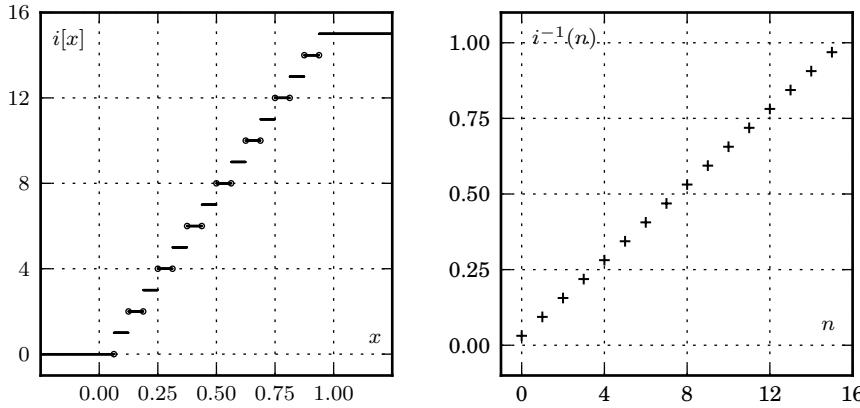


Figure 2.2: a 4-bit uniform quantizer on $(0, 1)$: forward quantization on the left, inverse quantization on the right.

More generally, if the quantizer values $[x]$ are dense enough – we say that the **high resolution assumption** is satisfied – then this relation holds approximately

The entropy attached to this collection of events is maximal when every event is equally likely, that is, under this approximation, when the step $\Delta(x)$ is proportional to the inverse of $p(x)$

$$\Delta(x) \propto \frac{1}{p(x)} \quad (2.11)$$

2.1.4 Implementation of Non-Uniform Quantizers

Non-uniform quantizers may be – at least conceptually – simply generated from uniform quantizers and non-linear transformations. If $[\cdot]$ denotes a uniform quantizer and f is an increasing mapping, the function $[\cdot]_f$ defined by the equation

$$[x]_f = (f^{-1} \circ [\cdot] \circ f)(x) \quad (2.12)$$

and displayed in figure 2.3 is a nonlinear quantizer. The function f is called the **characteristic function** of the quantizer. Depending on the selected range for the uniform quantizer, it is determined up to an affine transformation. Note that if f is linear or



Figure 2.3: nonlinear quantizer implementation.

affine, that is $f(x) = ax + b$, the quantizer $[\cdot]_f$ is still uniform – that's a reason why uniform quantizers are sometimes called **linear quantizers**.

Let Δ be the step of the uniform quantizer et let's determine what quantization step $\Delta_f(x)$ is attached to this scheme. For every value of x , the decision values at-

tached to $y = f(x)$ by the uniform quantizer are $[y]^-$ and $[y]^+$. Hence, the decision values for x and the non-linear quantization are

$$[x]_f^- = f^{-1}([y]^-) \text{ et } [x]_f^+ = f^{-1}([y]^+)$$

and if the high resolution assumption is satisfied the step $\Delta_f(x)$ is :

$$\Delta_f(x) = f^{-1}([y]^- + \Delta) - f^{-1}([y]^-) \simeq (f^{-1})'(f(x))\Delta = \frac{\Delta}{f'(x)}$$

something that is remembered as

$$\boxed{\Delta_f(x) \propto \frac{1}{f'(x)}}$$

(2.13)

The proportionaly constant may be easily recovered by noting that when $f(x) = x$, $[\cdot]_f = [\cdot]$ and therefore $\Delta(x) = \Delta$. If we impose moreover $f(0) = 0$, we find

$$f(x) \propto \int_0^x \frac{ds}{\Delta(s)} \quad (2.14)$$

If the quantizer is to maximize the entropy for the random variable X with density $p(x)$ we obtain

$$f(x) \propto \int_0^x p(y) dy \quad (2.15)$$

EXAMPLE. Let's consider the digital audio signal displayed in figure 2.4.

The uniform quantization on $(-1, 1)$ with step $\Delta = 10^{-1}$ is dense enough so that the associated histogram may be considered as a continuous function of the parameter x . We observe in figure 2.5 that this partition generates – for a large range of values of x – a counting measure $n(x)$ of a few thousands. The ration $n(x)/n$ where n is the total number of samples should therefore generate a good approximation of the density of the signal, considered as a sequence of independent and identically distributed values.

The logarithm of the histogram is similar to a function of the type $-a|x| + b$, $a > 0$ (cf fig. 2.6). We therefore select $p(x) \propto \exp(-a|x|)$. The optimal quantization – for the entropy criterion – and the corresponding characteristic function f such that $f(0) = 0$ are therefore given by:

$$\Delta(x) \propto e^{a|x|} \text{ and } f(x) \propto \operatorname{sgn}(x)(1 - e^{-a|x|}) \quad (2.16)$$

2.2 Logarithmic Quantization

We consider in this section several related quantizers whose characteristic function is – roughly speaking – the logarithm of their argument.

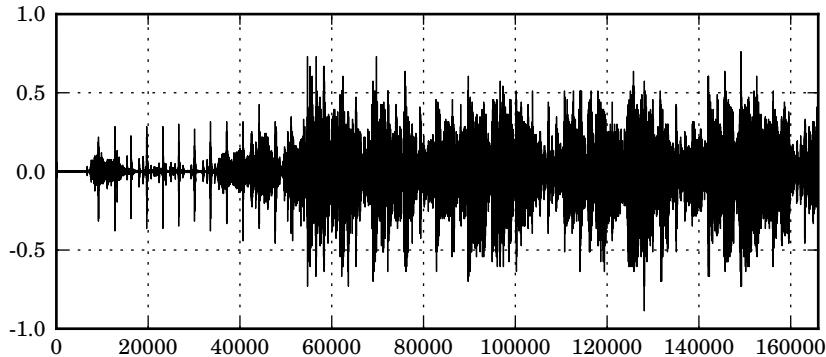


Figure 2.4: around 20 seconds of audio data (source format: NeXT/Sun .au)

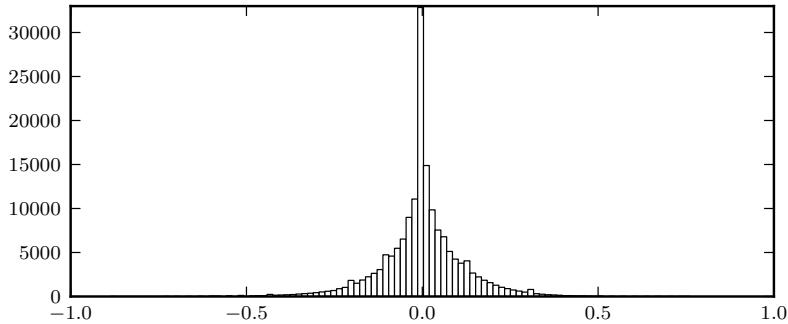


Figure 2.5: audio data histogram

2.2.1 The μ -law Quantizer.

Consider the probability law

$$p(x) \propto \begin{cases} \frac{1}{1 + \mu|x|/A} & \text{if } |x| \leq A, \\ 0 & \text{otherwise.} \end{cases} \quad (2.17)$$

The threshold A is necessary as otherwise the right-hand side of the equation would not be summable. The parameter μ controls directly the relative probability of low and high amplitude values as $p(\pm A)/p(0) = 1/(1 + \mu)$. In the limit case $\mu = 0$, we end up with a uniform probability distribution on $[-A, A]$.

The optimal quantizer for the entropy criterion satisfies (2.15) and therefore the characteristic function f such that $f(0) = 0$ satisfies

$$f(x) \propto \operatorname{sgn}(x) \ln \left(1 + \mu \frac{x}{A} \right). \quad (2.18)$$

If we limit the range of the quantizer to $[-1, 1]$ (we set $A = 1$) and enforce the con-

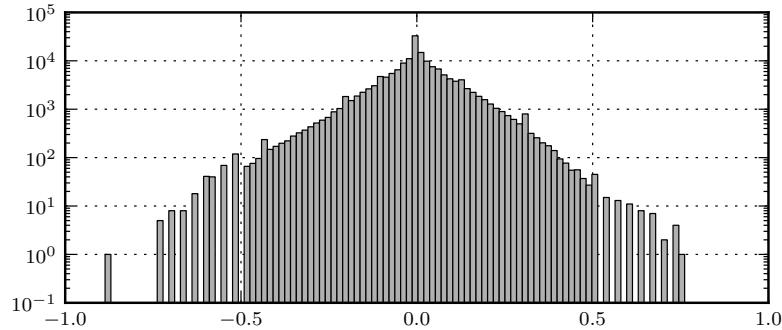


Figure 2.6: Log plot of the audio data histogram

straint $f([-1, 1]) = [-1, 1]$, we end up with

$$f(x) = \text{sgn}(x) \frac{\log(1 + \mu|x|)}{\log(1 + \mu)} \quad (2.19)$$

This quantization scheme is called μ -law and is for example used in the NeXT/SUN AU audio file format (files with extension .au or .snd). The actual implementation of the law, specified in the ITU-T G.711 standard – differs slightly from the theoretical formulas. A reference implementation is given in the code below:

```

class MuLaw(Quantizer):
    """
    Mu-law quantizer
    """
    scale = 32768
    iscale = 1.0 / scale
    bias = 132
    clip = 32635
    etab = array([0, 132, 396, 924, 1980, 4092, 8316, 16764])

    @staticmethod
    def sign(data):
        """
        Sign function such that sign(+0) = 1 and sign(-0) = -1
        """
        data = array(data, dtype=float)
        s = numpy_sign(data)
        i = where(s==0)[0]
        s[i] = numpy_sign(1.0 / data[i])
        return s

    def encode(self, data):
        data = array(data)
        s = MuLaw.scale * data
        s = minimum(abs(s), MuLaw.clip)
        [f,e] = frexp(s + MuLaw.bias)

        step = floor(32*f) - 16 # 4 bits
        chord = e - 8 # 3 bits
        sgn = (MuLaw.sign(data) == 1) # 1 bit

        mu = 16 * chord + step # 7-bit coding

```

```

mu = 127 - mu # bits inversion
mu = 128 * sgn + mu # final 8-bit coding

return array(mu, dtype=uint8)

def decode(self, i):
    i = array(i)
    i = 255 - i
    sgn = i > 127
    e = array(floor(i / 16.0) - 8 * sgn + 1, dtype=uint8)
    f = i % 16
    data = ldexp(f, e + 2)
    e = MuLaw.etab[e-1]
    data = MuLaw.iscale * (1 - 2 * sgn) * (e + data)

    return data

mulaw = MuLaw()

```

Note that this code is applied to values between -1 and 1 and uses 8 bits. The most significant bit encodes the sign; the amplitude of the signal is coded by the 7 remaining bits. The effective value of μ is approximately 250 but instead of using the expression $\log(1 + \mu|x|)$, we prefer a piecewise affine approximation of it (see fig 2.7). The values $[x]$ are then all multiples of 2^{-13} which limits the additional quantization error when the original signal is initially encoded with a uniform law using 14 bits or more. To ease the error correction when transmitted the bits other than the sign bit are finally inverted.

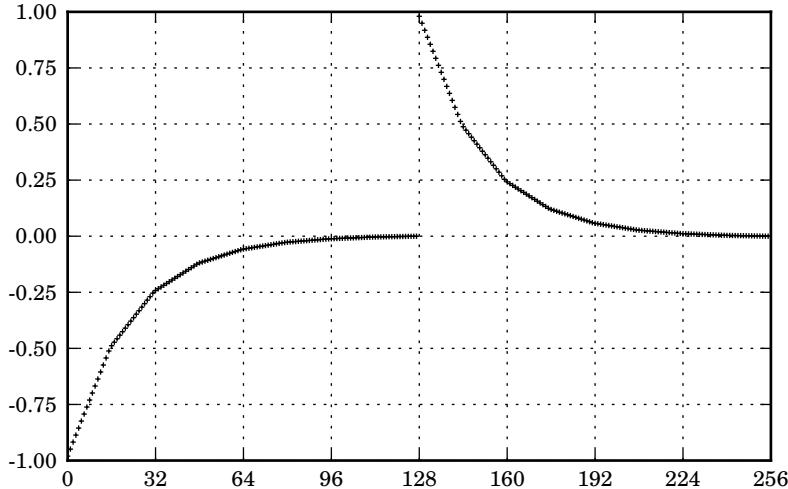
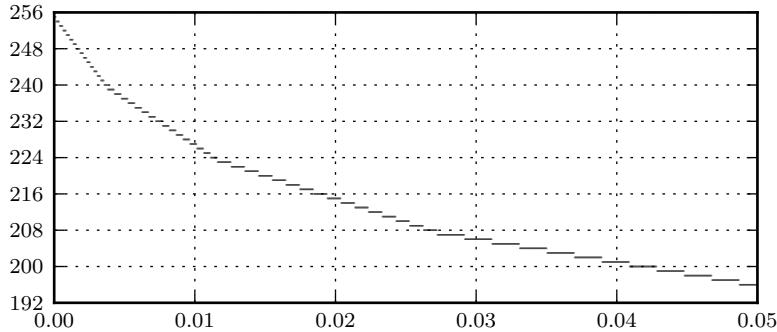


Figure 2.7: μ -law forward quantizer

2.2.2 IEEE754 Floating-Point Numbers and A -law

All scientific computing applications use implicitly a quantizer: the quantizer that represents approximation of real numbers in the floating-point arithmetic. The de-

Figure 2.8: μ -law inverse quantizer (partial view)

scription of two types of numbers – `single` and `double` (or rather, `single` and `double-precision numbers`) – is detailed in the IEEE 754 standard. In both cases, 1 bit is allocated to code the sign of the number, m bits for the exponent part and n bits for the fraction part,

$$s \in \{0, 1\}, e \in \{0, \dots, 2^m - 1\}, f \in \{0, \dots, 2^n - 1\} \quad (2.20)$$

consequently any real number is represented by an integer in $\{0, \dots, 2^{m+n+1}\}$ according to:

$$n = s \times 2^{m+n} + e \times 2^n + f \in \{0, \dots, 2^{m+n+1}\} \quad (2.21)$$

The `single` type is defined by $(m, n) = (8, 23)$ and the `double` type by $(m, n) = (11, 52)$; they are respectively coded on 32 and 64 bits.

We define

$$e_0 = 2^{m-1} - 1 \quad (2.22)$$

so that the value of the actual exponent $e - e_0$ range (almost symmetrically) from 2^{m-1} to $-2^{m-1} + 1$. The inverse quantizer attached to the standard floating point number representation is defined as follows: for an integer n , $[x] = i^{-1}(n)$ is given by

$$[x] = \begin{cases} \text{NaN} & \text{if } e = 2^m - 1 \text{ and } f \neq 0 \\ (-1)^s \infty & \text{if } e = 2^m - 1 \text{ and } f = 0 \\ (-1)^s (1 + f/2^n) \times 2^{e-e_0} & \text{if } 0 < e < 2^m - 1 \\ (-1)^s (f/2^n) \times 2^{1-e_0} & \text{if } e = 0 \end{cases} \quad (2.23)$$

The structure of these inverse quantizers are displayed in the figure 2.2.2; they are piecewise affine approximation of an exponential with a base of 2, except in the range $e = 0$ (the so-called **denormalized numbers**) where the graph is linear.

The A -law is a variant of the μ -law that has a structure similar the `single` and `double` types of floating point arithmetic but with a base different from 2. Given a value of A (often 87.7), the inverse of its characteristic function is defined on $[-1, 1]$ by

$$f^{-1}(x) = \text{sgn}(x) \times \begin{cases} (1 + \ln A)|x|/A & \text{if } |x| < \frac{1}{1+\ln A} \\ \exp(x(1 + \ln A) - 1)/A & \text{otherwise.} \end{cases} \quad (2.24)$$

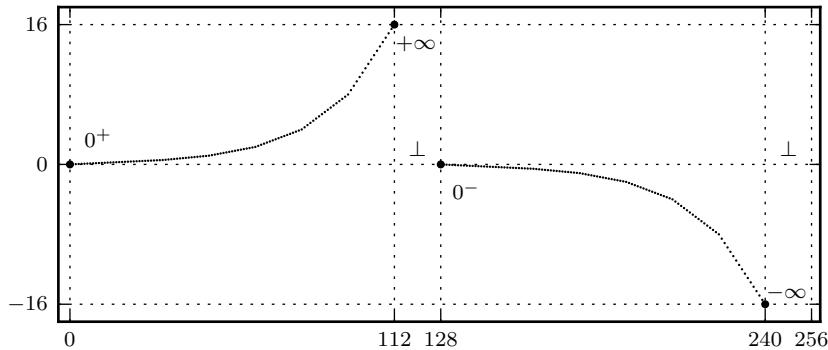


Figure 2.9: graph of the inverse quantizer for a floating point representation such that $(m, n) = (4, 3)$.

2.3 Signal-to-Noise Ratio

Computation of the signal-to-noise ratio. For a given sequence of k values x_n , the output $[x_n]$ of a quantizer may be interpreted as the sum of the original value and a perturbation sequence $b_n = [x_n] - x_n$ called a **noise**. The square of the signal-to-noise ratio – or SNR – is simply the ratio between the energies of those two values:

$$\text{SNR}^2 = \frac{\mathbb{E} \left(\sum_{n=0}^{k-1} x_n^2 \right)}{\mathbb{E} \left(\sum_{n=0}^{k-1} b_n^2 \right)} \quad (2.25)$$

The SNR is often measured in **decibels** (dB):

$$\text{SNR [dB]} = 20 \log_{10} \text{SNR} = 10 \log_{10} \text{SNR}^2 \quad (2.26)$$

When the values x_n are independent and follow the same probability law $p(x)$, this energy is given by

$$\mathbb{E} \left(\sum_{n=0}^{p-1} x_n^2 \right) = k \mathbb{E} (x_n^2) = k \int_{-\infty}^{+\infty} x^2 p(x) dx$$

and under a high resolution assumption we have

$$\begin{aligned}
\mathbb{E}(b_n^2) &= \int_{-\infty}^{+\infty} ([x] - x)^2 p(x) dx \\
&= \sum_y \int_y^{y+\Delta(y)} ([x] - x)^2 p(x) dx \\
&\simeq \sum_y p(y) \int_y^{y+\Delta(y)} (y + \Delta(y)/2 - x)^2 dx \\
&= \sum_y p(y) \frac{\Delta(y)^3}{12} \\
&\simeq \sum_y \int_y^{y+\Delta(y)} \frac{\Delta(x)^2}{12} p(x) dx \\
&= \int_{-\infty}^{+\infty} \frac{\Delta(x)^2}{12} p(x) dx \\
&= \frac{1}{12} \mathbb{E}(\Delta(x_n)^2)
\end{aligned}$$

Finally

$$\text{SNR}^2 = 12 \frac{\mathbb{E}(x_n^2)}{\mathbb{E}(\Delta(x_n)^2)} = 12 \frac{\int_{\mathbb{R}} x^2 p(x) dx}{\int_{\mathbb{R}} \Delta(x)^2 p(x) dx} \quad (2.27)$$

In the typical case where the probability density of the signal is uniform on $[-A, A]$ and the quantization is uniform on this range with a step Δ , we end up with

$$\text{SNR} = 2A/\Delta \quad (2.28)$$

Maximization of the SNR. For a given density of probability, how can we select the quantization scheme so that the SNR is maximal ? Formulated like that, this problem is not well-posed because the quantization noise may be made a small as possible with a decrease of the quantization step. The significant problem is to solve this problem under a constant bit budget. Without any loss of generality, we may assume that the signal has values in $[-1, 1]$ and that the characteristic function of the searched quantization satisfies $f([-1, 1]) = [-1, 1]$. If we allocate N bits to the quantization scheme, the step $\Delta(x)$ is determined by

$$\Delta(x) = \frac{2^{-N+1}}{f'(x)}$$

The SNR then takes the form

$$\text{SNR} = \kappa 2^N$$

where the value of κ depend only from the probability law of the signal and of the choice of f . In decibels, this equation is written as

$\text{SNR [dB]} \simeq 6.02 \times N + \kappa'$

(2.29)

that is, every extra bit increase the SNR by approximately 6 dB. To maximize the SNR, we then have to solve

$$\min_{f'} \int_{-1}^1 \frac{1}{f'(x)^2} p(x) dx \text{ subject to } f(1) - f(-1) = 2$$

or even, with $\psi = f'$

$$\min_{\psi} J(\psi) = \int_{-1}^1 \frac{1}{\psi(x)^2} p(x) dx \text{ with } K(\psi) = \int_{-1}^1 \psi(x) dx = 2$$

At the optimum, there is a $\lambda \in \mathbb{R}$ such that the lagrangian $L(\psi) = J(\psi) + \lambda K(\psi)$ satisfies $dL(\psi) = 0$, that is

$$\text{for all } \delta\psi : [-1, 1] \rightarrow \mathbb{R}, \int_{-1}^1 \left(-\frac{2}{\psi(x)^3} p(x) + \lambda \right) (\delta\psi)(x) dx = 0$$

and that implies

$$-\frac{2}{\psi(x)^3} p(x) + \lambda = 0$$

and hence

$$f'(x) \propto (p(x))^{\frac{1}{3}} \quad (2.30)$$

Chapter 3

Linear Prediction

3.1 Prediction Principles

Prediction relies on the signal past and current values to estimate its future values. Such process relies on a given class of models, supposed to rule the behavior of the signal whose parameters shall be identified. This step being achieved, we may compute the **prediction error** or **residual**, the difference between the actual signal values and the predicted values. In the context of data compression, and if the model used for prediction is accurate, the prediction error has a much smaller range than the original values and therefore may be coded more efficiently.

3.1.1 Polynomial Prediction

Polynomial prediction is one of the simplest fixed-parameter prediction schemes. Given m sample values x_0, x_1, \dots, x_{m-1} , we identify the unique polynomial P of order at most $m - 1$ such that

$$\forall n \in \{0, 1, \dots, m - 1\}, P(n) = x_n$$

and with it, provide a prediction \hat{x}_m for the value x_m :

$$\hat{x}_m = P(m)$$

The polynomial P , given by

$$P(n) = \sum_{n=0}^{m-1} a_n j^n$$

is determined by the matrix equality

$$\begin{bmatrix} 1 & 0^1 & 0^2 & \dots & 0^{n-1} \\ 1 & 1^1 & 1^2 & \dots & 1^{n-1} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & (n-1)^1 & (n-1)^2 & \dots & (n-1)^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

The matrix on the left-hand side is an invertible Vandermonde matrix, therefore the polynomial coefficients may be obtained from the signal values x_0, x_1, \dots, x_{n-1} and \hat{x}_n can be computed.

The prediction error $e_n = x_n - \hat{x}_n$ may be computed efficiently. Consider a signal x_n whose values are stored in the NumPy array x , and let's begin with a polynomial prediction of order 0 or **difference coding**; our prediction model is that the signal is constant. We may compute at once all the prediction errors $e_n = x_n - x_{n-1}$ for the signal with

```
e = diff(x) = [x[1] - x[0], x[2] - x[1], ...]
```

However, we end up with a vector with only $\text{len}(x) - 1$ values : e_0 is undefined and $e[0]$ would be e_1 ; we would have no information about the first value of the signal $x[0]$ in e . We therefore add $x[0]$ as the first value of e . This is the same as taking into account a supposedly zero value $x[-1]$ of the signal, and may by adding 0 to the beginning of x before applying the difference operator:

```
e = diff(r_[0, x])
```

Reconstruction of x from the residual e can be done by computing the cumulative sum $x_n = \sum_{i=0}^n e_i$:

```
x = cumsum(e)
```

What about first-order polynomial prediction then ? The formula for \hat{x}_n is $\hat{x}_n = x_{n-1} + (x_{n-1} - x_{n-2})$ and the corresponding residual is

$$e_n = x_n - \hat{x}_n = x_n - 2x_{n-1} + x_{n-2} = (x_n - x_{n-1}) - (x_{n-1} - x_{n-2}).$$

This residual may therefore be computed as :

```
e_0 = diff(r_[0, x])
e = diff(r_[0, e_0])
```

and reconstruction is given as

```
e_0 = cumsum(e)
x = cumsum(e_0)
```

This scheme may be generalized to a polynomial prediction of arbitrary order.

3.1.2 Optimal Linear Prediction

Consider the following problem: given a sequence $\{x_n\}$, get the best linear approximation \hat{x}_n of x_n as a linear combinations of the m previous samples:

$$\hat{x}_n = a_1 x_{n-1} + \cdots + a_m x_{n-m}.$$

Let's be more precise: if the values x_0, x_1, \dots, x_{n-1} are available, we can predict $n - m$ values and therefore measure the prediction error by the quadratic criterion:

$$j(a) = \sum_{i=m}^{n-1} (x_i - a_1 x_{i-1} - \cdots - a_m x_{i-m})^2 \quad (3.1)$$

The process that produces the estimates \hat{x}_n is known as a **Wiener(-Hopf) filter**. The vectors $a = (a_1, \dots, a_m)$ that minimize the quadratic error are therefore solution of

$$a = \arg \min_x \|e\|^2, \quad \text{with } e = Ax - b \quad (3.2)$$

where

$$A = \begin{bmatrix} x_{m-1} & x_{m-2} & \dots & x_0 \\ x_m & x_{m-1} & \dots & x_1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n-2} & x_{n-3} & \dots & x_{n-m-1} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} x_m \\ x_{m+1} \\ \vdots \\ x_{n-1} \end{bmatrix} \quad (3.3)$$

The analysis of this problem shows that there is a unique solution $a = x$ if A is into that is if the square matrix $A^t A$ is full-rank (m) and the solution is

$$a = [A^t A]^{-1} A^t b \quad (3.4)$$

Indeed, the function $j(x) = \|Ax - b\|^2$ to minimize is quadratic in x : $j : x \mapsto 1/2x^t Qx + Lx + c$. The Taylor decomposition at the point a yields $j(x) = j(a) + \nabla j(a)^t(x-a) + 1/2(x-a)^t \nabla^2 j(a)(x-a)$. Any a such that $\nabla j(a) = 0$ (here, with the full rank assumption, there is a unique solution) is a global minimum. A geometrical analysis would also have worked: a solution a to the minimum problem has to be such that for any x , the error vector $e = b - Ax$ and Ax are orthogonal ; this also yields the condition (3.4). The same geometrical analysis – or a direct computation – yields the error measure as by the Pythagorean Theorem, we have $\|b\|^2 = \|Aa\|^2 + \|Aa - b\|^2$

$$\|e\|^2 = \|b\|^2 - \|Aa\|^2 \quad (3.5)$$

The full-rank assumption is not a problem in practice: it just means that the signal data is rich enough to discriminate a unique optimal candidate x . If that's not the case, instead of $[A^t A]^{-1} A^t$ we could use the pseudo-inverse of A^\dagger of A , defined as

$$A^\dagger = \lim_{\epsilon \rightarrow 0} [A^t A + \epsilon I]^{-1} A^t \quad (3.6)$$

such that $a = A^\dagger b$ provides among the solutions x of the minimisation problem the one with the smallest norm.

Instead of implementing ourselves a solution of the minimization problem, we provide a reference implementation of the linear prediction problem that uses the NUMPY function `linalg.lstsq` that solves this least-square (quadratic) minimization problem:

```
def lp(signal, m):
    """Wiener predictor coefficients"""
    signal = ravel(signal)
    n = len(signal)

    A = array([signal[m - arange(1, m + 1) + i] for i in range(n-m)])
    b = signal[m:n]
    a = linalg.lstsq(A, b)[0]

    return a
```

Estimation of the parameter a as a solution of (3.2 + 3.3) is called the **covariance method**. We present now a variant of this process, called **autocorrelation method**, that is amenable to faster implementations and also has more pleasant properties, such as the stability of the inverse of error filters (see section (3.1.5)).

Consider the following change: add m zeros at the start of $\{x_n\}$, add m zeros at the end, then apply the autocorrelation method. What we are trying to achieve is to predict ALL the values of x_n (even when we don't have all prior values) and conversely, for symmetry reasons that will be clearer in a moment, predict the trailing zeros from significant data as long as there is one usable sample.

The implementation of a linear predictor solver that support both methods is simple:

```
def lp(signal, m, zero_padding=False):
    """Wiener predictor coefficients"""
    signal = ravel(signal)
    if zero_padding: # select autocorrelation method instead of covariance
        signal = r_[zeros(m), signal, zeros(m)]
    n = len(signal)

    A = array([signal[m - arange(1, m + 1) + i] for i in range(n-m)])
    b = signal[m:n]
    a = linalg.lstsq(A, b)[0]

    return a
```

Note that in the covariance methods, the new A and b are:

$$A = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ x_0 & 0 & \dots & 0 & 0 \\ x_1 & x_0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n-2} & \dots & \dots & \dots & x_{n-m-1} \\ x_{n-1} & \dots & \dots & \dots & x_{n-m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & x_{n-1} & x_{n-2} \\ 0 & \dots & \dots & 0 & x_{n-1} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad (3.7)$$

Set $x_i = 0$ if $i < 0$ or $i \geq n$ and

$$c_j = \sum_{i=-\infty}^{+\infty} x_i x_{i-j} \quad (3.8)$$

We now have

$$C(m) = A^t A = \begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_{m-1} \\ c_1 & c_0 & c_1 & \dots & c_{m-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{m-1} & c_{m-2} & \dots & c_1 & c_0 \end{bmatrix} \quad \text{and} \quad A^t b = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} \quad (3.9)$$

The correlation matrix $C(m) = A^t A$ is now symmetric but also **Toeplitz** (or **diagonal constant**) and therefore efficient algorithms to solve the least-square problem exist. Note also that we could add MORE zeros before or after the data and that it wouldn't change a thing: A and b change but $A^t A$ and $A^t b$ are the same. In this context, the set of scalare equations

$$[A^t A]a = A^t b \quad (3.10)$$

are called **Wiener-Hopf** equatons, **Yule-Walker** equations or **normal equations**.

So the autocorrelation method naturally fits into the "infinite signals" point of view, strongly related to the convolution operator (see section 4.10). To be more precise, consider the (causal) signal $\{x_n\}$ defined for ALL n (by setting 0 when not defined) and consider the signal $\{a_n\}$ defined in the same way (in particular, $a_0 = 0$). Then, the (possibly) non-zero coeffs of $\{a_n\} * \{x_n\}$ and $\{x_n\}$ correspond to the following vectors:

$$\{a_n\} * \{x_n\} \rightarrow Aa \quad \text{and} \quad \{x_n\} \rightarrow b$$

so that the minimisation problem we are trying to solve really is:

$$\{a_n\} = \arg \min \|\{x_n\} - \{h_n\} * \{x_n\}\|^2$$

among all strictly causal filters h_n with $h_i = 0$ if $i > m$. Or if we introduce the prediction error filter $b_0 = 1$ and $b_n = -a_n$,

$$\{r_n\} = \arg \min \|\{h_n\} * \{x_n\}\|^2$$

among causal filters with $h_0 = 1$ and length less or equal to $m + 1$. This is a causal deconvolution problem.

Note however, the completion of the signal by 0's even if the result is questionable: if the real signal has values outside the window, they are probably not 0. It does not matter much when the length of the window is big with respect to the prediction order, but otherwise a the covariance method is probably more accurate.

Additional Properties of the Autocorrelation Method.

Let $r_n = (1, -a_1, \dots, -a_m)$ be the coeffs of the prediction error filter. We are going to prove that:

$$\begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_m \\ c_1 & c_0 & c_1 & \dots & c_{m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_m & c_{m-1} & \dots & c_1 & c_0 \end{bmatrix} \begin{bmatrix} 1 \\ -a_1 \\ \vdots \\ -a_{m-1} \\ -a_m \end{bmatrix} = \begin{bmatrix} \|\{e_n\}\|^2 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad (3.11)$$

By the way, that gives us a new way to get the a_n : get $C(m+1)^{-1}[1, 0, \dots, 0]^t$ and normalize the result w.r.t. the first coefficient (whose meaning is interesting: the energy of the residual!). In the sequel, we denote $\sigma_r = \|\{e_n\}\|$.

All the 0 of the equations are a direct consequence of $AA^t a = A^t b$. The first coeff is equal to $c_0 - [c_1, \dots, c_m] \cdot a = c_0 - (A^t b) \cdot a = c_0 - b \cdot (Aa) = \|\{x_n\}\|^2 - \{x_n\} \cdot (\{a_n\} * \{x_n\}) = \{x_n\} \cdot \{e_n\}$. But by the orthogonality condition, this is equal to $\{e_n\} \cdot \{e_n\} = \sigma_r^2$.

Linear Prediction of Unlimited Order - White Noise

Let x_0, \dots, x_{n-1} be a finite sequence of real values. We may extend the definition of x_i for arbitrary values of the index by setting $x_i = 0$ if i does not belong to the original index range. Now we may try to solve the linear prediction problem of unlimited order by minimizing over all *infinite* sequences of prediction coefficients a_i the quadratic sum of the prediction error e_i :

$$\sum_{i=-\infty}^{+\infty} e_i^2 \text{ where } e_i = x_i - \sum_{j=1}^{+\infty} a_j x_{i-j} \quad (3.12)$$

Any solution to this problem satisfies

$$\forall j > 0, \sum_{i=0}^{+\infty} e_i e_{i-j} = 0 \quad (3.13)$$

The prediction error of the unlimited order linear prediction problem is not correlated at all – it is a **white noise**.

Proof. Let $(x * y)_i = \sum_{j=-\infty}^{+\infty} x_j y_{i-j}$, $\langle x, y \rangle = \sum_{i=-\infty}^{+\infty} x_i y_i$ and $\|x\| = \sqrt{\langle x, x \rangle}$. We denote by $L^2(\mathbb{Z})$ the set of infinite sequences x such that $\|x\| < +\infty$ and if $I \subset \mathbb{Z}$, by $L^2(I)$ the set of sequences x in $L^2(\mathbb{Z})$ such that $x_i = 0$ if $i \notin I$. Our minimization problem may be formalized as

$$\min_{a \in A} \|x - a * x\|^2 \text{ with } A = L^2(\mathbb{N}^*) \quad (3.14)$$

Let $e = x - a * x$ be the prediction error ; any solution a is a solution of (3.14) satisfies $\forall \delta \in L^2(\mathbb{N}^*)$, $\langle \delta * x, e \rangle = 0$. Let \bar{x} be the infinite sequence such that $\bar{x}_i = x_{-i}$. We have $\langle \delta * x, e \rangle = \langle \delta, \bar{x} * e \rangle$ and $(\bar{x} * e)_j = \sum_{i=-\infty}^{+\infty} x_{i-j} e_i$. Therefore $\forall j > 0$, $\sum_{i=-\infty}^{+\infty} x_{i-j} e_i = 0$. As any e_i is a linear combination of the previous values of x , this equality yields

$$\forall j > 0, \sum_{i=0}^{+\infty} e_i e_{i-j} = 0$$

■

3.1.3 Finite Impulse Response (FIR) Filters

The Wiener-Hopf prediction that produces the sequence of estimates \hat{x}_n from the x_n or error filter that outputs $e_n = x_n - \hat{x}_n$ are special cases of **finite impulse response (FIR) filters** : they associate to an input sequence u_n an output sequence y_n related by:

$$y_n = a_0 u_n + a_1 u_{n-1} + \cdots + a_{N-1} u_{n-N+1} \quad (3.15)$$

A core, real-time implementation for such system is given by:

```
class FIR(Filter):
    def __call__(self, input):
        if shape(input):
            inputs = ravel(input)
            return array([self(input) for input in inputs])
        else:
            output = self._a[0] * input + dot(self._a[1:], self.state)
            if len(self.state):
                self.state = r_[input, self.state[:-1]]
            return output
```

where some features, such as the initialization and changes of a , the management of the filter state, common between finite impulse response filters and auto-regressive filters (see section 3.1.4) are implemented in the base class `Filter`. We talk about a *real-time* implementation of a FIR because instances of `FIR` produce the value y_n as soon as u_n is available. To do this, they need to store a state that contains at the time n the $N - 1$ past values $u_{n-1}, \dots, u_{n-N+1}$ of the input.

Consider as an example the 4-point moving average filter:

$$y_n = \frac{1}{4}(u_n + u_{n-1} + u_{n-2} + u_{n-3})$$

Such a filter may be defined and used by the following code:

```
>>> ma = FIR([0.25, 0.25, 0.25, 0.25])
>>> ma.state
array([ 0.,  0.,  0.])
>>> ma(1.0)
0.25
>>> ma(2.0)
0.75
>>> ma(3.0)
1.5
>>> ma(4.0)
2.5
>>> ma([5.0, 6.0, 7.0, 8.0, 9.0, 10.0])
array([ 3.5, 4.5, 5.5, 6.5, 7.5, 8.5])
>>> ma.state
array([ 10.,  9.,  8.])
```

Once the filter `ma` is initialized (by default with a zero state), every call to `ma` shall give one or several new input values and as many output values are produced.

Note that if we start with a zero state and input a single non-zero value before sending a sequence of zeros, the filter will output a finite number of (possibly) non-zero and will then output only zeros: this is actually a defining property of finite impulse response filters.

```
>>> ma.state = [0.0, 0.0, 0.0]
>>> ma([1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
array([ 0.25, 0.25, 0.25, 0.25, 0. , 0. , 0. , 0. ])
```

In the context of linear prediction, here is how the prediction coefficients `a` produced by `lp` may be used to build the predictor filters and error filters.

```
>>> a = lp(x, ...)
>>> predictor = FIR(r_[0.0, a])
>>> error = FIR(r_[1.0, -a])
```

3.1.4 Auto-Regressive (AR) Filters

Autoregressive (AR) filters are – at least formally – inverses of FIR filters. Consider the equation of an FIR error filter whose input is x_n and output e_n

$$e_n = 1.0 \cdot x_n - a_1 x_{n-1} - \cdots - a_m x_{n-m}$$

If this equation holds for every value of n , the inverse system that has e_n as an input and x_n as an output is ruled by:

$$x_n = a_1 x_{n-1} + \cdots + a_m x_{n-m} + e_n$$

Therefore we consider the class of autoregressive systems with inputs u_n and outputs y_n ruled by

$$y_n = a_1 y_{n-1} + \cdots + a_N y_{n-N} + u_n \quad (3.16)$$

A core implementation is given as

```
class AR(Filter):
    def __call__(self, input):
        if shape(input):
            inputs = ravel(input)
            return array([self(input) for input in inputs])
        else:
```

```

output = dot(self.a, self.state) + input
self.state[-1] = output
self.state = roll(self.state, 1)
return output

```

The state of such an AR instance is the sequence of N previous values of y_n . The usage of the class `AR` is similar to `FIR`. For example, the filter:

$$y_n = 0.5 \cdot y_{n-1} + u_n$$

may be defined and used by

```

>>> ar = AR([0.5])
>>> ar.state = [1.0]
>>> ar(0.0)
0.5
>>> ar(0.0)
0.25
>>> ar(0.0)
0.125
>>> ar(0.0)
0.0625
>>> ar([1.0, 1.0, 1.0, 1.0])
array([ 1.03125 , 1.515625 , 1.7578125 , 1.87890625])
>>> ar.state
array([ 1.87890625])

```

3.1.5 Transfer Function, Stability and Frequency Response

Transfer Function

The **transfer function** of a (linear, time-invariant, single-input single output) system is a (partial) function $H : \mathbb{C} \rightarrow \mathbb{C}$ defined in the following way: given $z \in \mathbb{C}$ and a complex-valued input signal $u_n = uz^n$ the corresponding output having the structure $y_n = yz^n$, if it exists, satisfies:

$$y = H(z)u \quad (3.17)$$

For example, the FIR filter defined by the equation (3.15) has the transfer function

$$H(z) = a_0 + a_1 z^{-1} + \cdots + a_{N-1} z^{-N+1} \quad (3.18)$$

and the AR filter defined by the equation (3.16) has the transfer function

$$H(z) = \frac{1}{1 - a_1 z^{-1} - \cdots - a_N z^{-N}} \quad (3.19)$$

Stability

A filter is **(input-output) stable** if all bounded input signals result in bounded outputs. Stability of filters whose transfer function is rational – such as FIR and AR filters – is conditioned by the location of their **poles**, the roots of their transfer functions. Precisely, such a filter is stable if and only if all its poles have a negative real part.

The classes `FIR` and `AR` have a method that return their poles ; its implementation is based on the `numpy.lib.polynomial.roots` function that computes the roots of a polynomial.

For FIR filters, the situation is simple: as

$$H(z) = a_0 + a_1 z^{-1} + \cdots + a_{N-1} z^{-N+1} = \frac{a_0 z^{N-1} + a_1 z^{N-2} + \cdots + a_{N-1}}{z^{N-1}},$$

all N poles are 0 and therefore all FIR filters are stable.

```
class FIR(Filter):
    ...
    def poles(self):
        return zeros(len(self.a))
```

For AR filters,

$$H(z) = \frac{1}{1 - a_1 z^{-1} - \cdots - a_N z^{-N}} = \frac{z^N}{z^N - a_1 z^{N-1} - \cdots - a_N}$$

and therefore the poles are the solution of the polynomial $P(z) = z^N - a_1 z^{N-1} - \cdots - a_N$.

```
class AR(Filter):
    ...
    def poles(self):
        return roots(r_[1.0, -self.a])
```

As an example, consider the two auto-regressive filters ruled by:

$$y_n = 0.5 \cdot y_{n-1} - 0.5 \cdot y_{n-2} + u_n \quad \text{and} \quad y_n = y_{n-1} + y_{n-2} + y_{n-3} + y_{n-4} + u_n$$

The first one is stable but the second one is unstable:

```
>>> ar = AR([0.5, -0.5])
>>> ar.poles()
array([ 0.25+0.66143783j,  0.25-0.66143783j])
>>> max(abs(pole) for pole in ar.poles())
0.70710678118654757
>>> ar = AR([1.0])
>>> ar = AR([1.0, 1.0, 1.0, 1.0])
>>> ar.poles()
array([ 1.92756198+0.j , -0.77480411+0.j ,
       -0.07637893+0.81470365j, -0.07637893-0.81470365j])
>>> max(abs(pole) for pole in ar.poles())
1.9275619754829254
```

As a matter of fact, we will deal in the next sections with AR filters that are inverses of FIR prediction error filters provided by linear prediction. Such filters are always stable when the autocorrelation method is used but may be unstable with the covariance method.

Frequency Response

When a filter is stable, it makes sense to ask what output corresponds to a cosine input with frequency f , amplitude A and phase ϕ . If the input sequence is scheduled to produce a new value every Δt seconds, we have $u_n = A \cos 2\pi f n \Delta t + \phi$ and therefore

$$u_n = A/2 \cdot e^{i\phi} (e^{i2\pi f \Delta t})^n + A/2 e^{-i\phi} \cdot (e^{-i2\pi f \Delta t})^n.$$

By the definition of the transfer function, we have the corresponding output y_n :

$$\begin{aligned} y_n &= A/2 \cdot e^{i\phi} H(2\pi f \Delta t) (e^{i2\pi f \Delta t})^n + A/2 \cdot e^{-i\phi} H(-2\pi f \Delta t) (e^{-i2\pi f \Delta t})^n \\ &= \Re [H(2\pi f \Delta t) A e^{i\phi} e^{i2\pi f \Delta t n + \phi}] \end{aligned}$$

So if we consider the polar decomposition

$$H(2\pi f \Delta t) A e^{i\phi} = A' e^{i\phi'}$$

then the cosine output of the filter is

$$y_n = A' \cos 2\pi f n \Delta t + \phi'$$

The function

$$f \mapsto H(2\pi f \Delta t) \quad (3.20)$$

that relates input and ouput amplitude and phase at the frequency f is called the filter frequency response. We often consider separately

$$|H(2\pi f \Delta t)| \text{ and } \angle H(2\pi f \Delta t),$$

the frequency response gain and phase.

The implementation of transfer functions for FIR and AR filters relies on the computation of signal spectrum or Fourier transform, provided by the function `F` of the `spectrum` module, see section 4.2.2.

```
from spectrum import F

class FIR(Filter):
    ...
    def __F__(self, **kwargs):
        dt = kwargs.get("dt") or 1.0
        return F(self.a / dt, dt=dt)

class AR(Filter):
    ...
    def __F__(self, **kwargs):
        dt = kwargs.get("dt") or 1.0
        FIR_spectrum = F(FIR(a=r_[1.0, -self.a]), dt=dt)
        def AR_spectrum(f):
            return 1.0 / FIR_spectrum(f)
        return AR_spectrum
```

The function `F` is generally used to get the frequential representation of and object, signal or filter, or something else. Apart from signals, for which we directly compute the spectrum, the objects are supposed to know what their spectral representation is and encode this information in the special method `__F__`; for filters, we return the frequency response. Those methods being defined for FIR and AR filters, we may use them like that:

```
>>> ma = FIR([0.5, 0.5])
>>> tf = F(ma, dt=1.0)
>>> tf([0.0, 0.1, 0.2, 0.3, 0.4, 0.5])
array([-1.0000000 +0.0000000e+00j,  0.9045085 -2.93892626e-01j,
       0.6545085 -4.75528258e-01j,  0.3454915 -4.75528258e-01j,
       0.0954915 -2.93892626e-01j,  0.0000000 -6.12303177e-17j])
```

3.2 Voice Analysis and Synthesis

3.2.1 The TIMIT corpus

The TIMIT corpus is a collection of read speech data that includes for each utterance 16-bit 16 kHz waveforms as well as time-aligned orthographic, phonetic and word

transcriptions. It was designed – as a joint effort among the Massachusetts Institute of Technology (MIT), SRI International (SRI) and Texas Instruments, Inc. (TI) – for acoustic-phonetic studies and for the development and evaluation of automatic speech recognition systems.

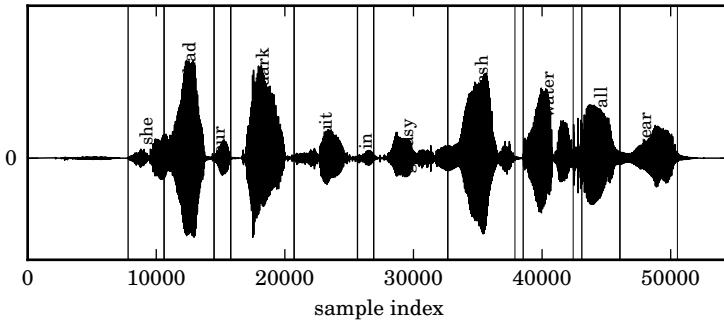


Figure 3.1: Waveform of the TIMIT utterance 'dr1-fvmh0/sa1' and display of its segmentation into words.

The **PYTHON** library NLTK – for Natural Language Toolkit – is an open source collection of modules that provides linguistic data and documentation for research and development in natural language processing and text analytics (<http://www.nltk.org/>). As a part of the distribution, is a small sample of the TIMIT corpus is made available.

The samples from TIMIT are designated by ids whose list is given by the `utteranceids` method:

```
>>> import nltk
>>> timit = nltk.corpus.timit
>>> timit.utteranceids()
['dr1-fvmh0/sa1', 'dr1-fvmh0/sa2', 'dr1-fvmh0/si1466', 'dr1-fvmh0/si2096',
 ...
 'dr8-mbcg0/sx237', 'dr8-mbcg0/sx327', 'dr8-mbcg0/sx417', 'dr8-mbcg0/sx57']
>>> uid = timit.utteranceids()[0]
'dr1-fvmh0/sa1'
```

The corpus provides a detailed decomposition of the utterances in words as well as **phones**¹ – speech segments that have distinct properties. Those decompositions are timed, the numbers being sample indices.

```
>>> timit.words(uid)
['she', 'had', 'your', 'dark', 'suit', 'in', 'greasy', 'wash', 'water', 'all', 'year']
>>> timit.word_times(uid)
[('she', 7812, 10610), ('had', 10610, 14496), ('your', 14496, 15791),
 ('dark', 15791, 20720), ('suit', 20720, 25647), ('in', 25647, 26906),
 ('greasy', 26906, 32668), ('wash', 32668, 37890), ('water', 38531, 42417),
 ('all', 43091, 46052), ('year', 46052, 50522)]
>>> timit.transcription_dict()["she"]
['sh', 'iy1']
>>> timit.phones(uid)
['h#', 'sh', 'iy', 'hv', 'ae', 'dcl', 'y', 'ix', 'dcl', 'd', 'aa', 'kcl', 's',
```

¹not to be confused with **phonemes**, set of phones that are cognitively equivalent (<http://en.wikipedia.org/wiki/Phoneme>).

```
'ux', 'tcl', 'en', 'gcl', 'g', 'r', 'iy', 's', 'iy', 'w', 'aa', 'sh', 'epi',
'w', 'aa', 'dx', 'ax', 'q', 'ao', 'l', 'y', 'ih', 'ax', 'h#']
>>> timit.phone_times(uid)
[('h#', 0, 7812), ('sh', 7812, 9507), ('iy', 9507, 10610), ('hv', 10610, 11697),
...
('ih', 47848, 49561), ('ax', 49561, 50522), ('h#', 50522, 54682)]
```

The `audiodata` method, combined with the `bitstream` module, provide the waveform as a single-dimensional NumPy array `data`:

```
>>> str_data = timit.audiodata(uid)
>>> data = BitStream(str_data).read(int16, inf).newbyteorder()
```

3.2.2 Voice Analysis and Compression

The knowledge that the audio data that we are willing to compress is a voice signal can go a long way in the reduction of bit rate. Consider for example the G.711 PCM speech codec: defined in 1972, it is based on a 8 kHz sampling time and a quite generic method of non-linear quantization (8-bit μ -law or A -law). It achieves a data rate of 64 kb/s. A more specific technology developed in the early 90's, and based on linear prediction, the full-rate GSM, has a 13 kbps bit rate. More recent efforts in this direction have achieved a quality similar to the G.711 codec at 6.4 kbps, or with a lesser quality go as low 2.4 kbps (see [HSW01]).

In the sequel, we'll assume that the data we consider is sampled at 8 khz ; this is a standard assumption in fixed telephony that takes into account the fact that most voice audio content is in the 300-3400 Hz band. Applications that require more accurate descriptions of voice data may use **wideband** and use a 16 kHz sampling instead for a higher accuracy – all the audio data in the TIMIT data base uses this sampling frequency for example.

Short-Term Prediction

Beyond the selection of an appropriate sampling rate, the key to achieve significant compression rate is to recognize that voice has a local – say on a 20 ms frame – stationary structure that can therefore be described by a small numbers of parameters. This property is clearly visible in the figure 3.2.

The figure 3.3 displays two 20-ms voice fragments sampled at 8 kHz and the corresponding residuals after a prediction of order . The first one clearly has achieved its goal: the residual appears to be left without structure and is a good approximation of a white noise. For those kind of data, the short-term prediction provides a simple production model: an AR synthesis filter whose input is a white noise. For the second type of signals, for which the residual is clearly not random, we need a more complex production model that complements the short-term prediction with a long term prediction (see sections 3.2.2 and 3.2.2).

Spectral Analysis

The spectrum of a voice segment $x(t)$ may be estimated classically, with the formula

$$x(f) = \Delta t \sum_{t \in \mathbb{Z} \Delta t} x(t) \exp(-i2\pi ft),$$

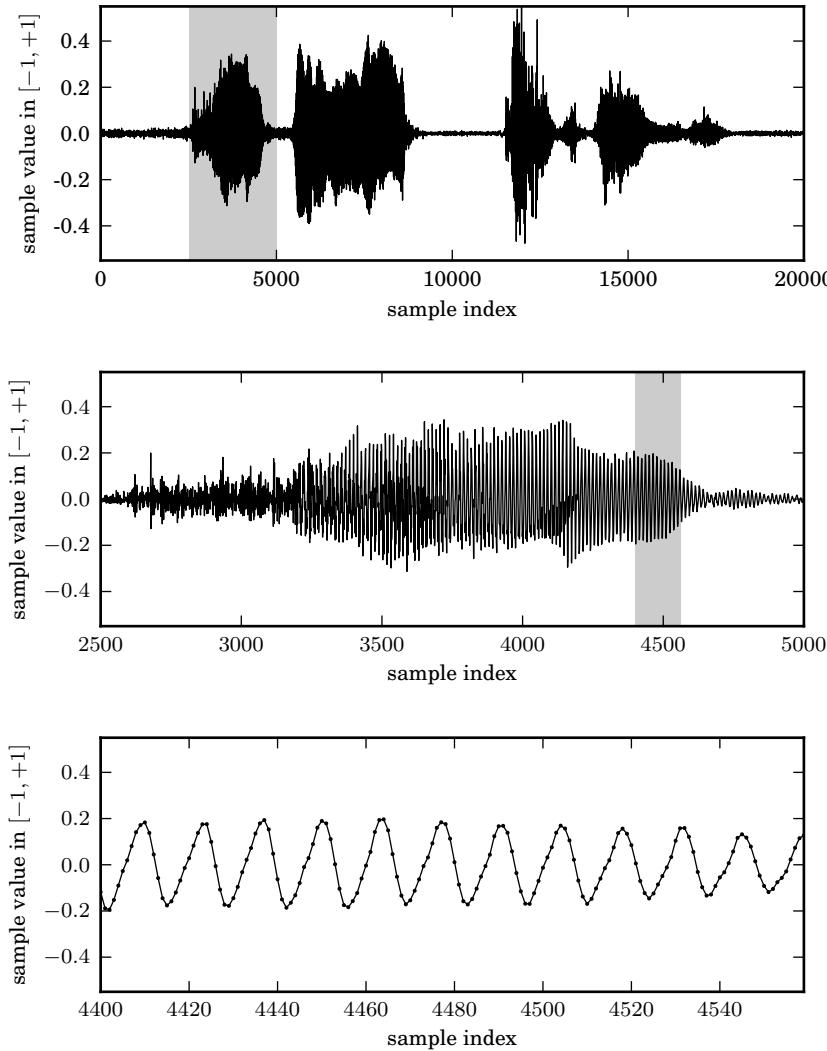


Figure 3.2: **Voice patterns.** A voice signal sampled at 8 kHz displays complex and non-stationary patterns on a scale of 2.5 s (top). When we zoom to a 300 ms scale (middle), and then further to a 20 ms scale (bottom), we see that locally, the signal appears to be almost periodic.

but there is another way: if we have performed a successful prediction of the data that leads to a synthesis filter with frequency response $\frac{1}{1-A(f)}$, the prediction error should be almost white and its spectrum $e(f)$ should be approximately constant. As the signal data $x(t)$ is related to $e(t)$ by

$$x(f) = \frac{1}{1-A(f)} e(f),$$

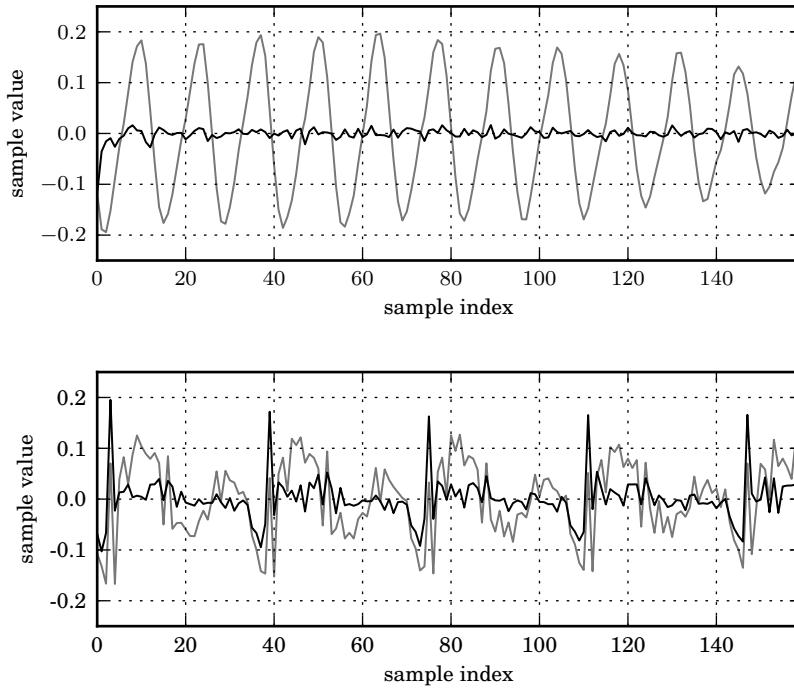


Figure 3.3: **short-term prediction error.** top: a frame of 160 samples within a 8 kHz signal (grey) and the corresponding prediction error (black) for a linear prediction of order 16 (covariance method). The residual show little remaining structure. bottom: the prediction error (black) of the voice signal (grey) still exhibits a periodic structure, made of regularly spaced spikes, characteristic of voiced segments.

the frequency response of the synthesis filter provides a (parametric) estimate of the signal spectrum. Both kind of methods are illustrated in figure 3.2.2.

Models of the Vocal Tract

Continuous Modelling. A simple model of vocal tract is the **horn**: a tube whose cross-sectional area A is a function of the position x in the tube. Let ϕ denote the air flow, positive by convention if the air travel towards the right, p the pressure, ρ the air density and K its bulk modulus. Newton's second law of motion yields

$$\frac{d}{dt} \rho \phi = - \frac{dpA}{dx}.$$

We approximate this equation by:

$$\rho \frac{\partial \phi}{\partial t} = -A \frac{\partial p}{\partial x} \quad (3.21)$$

On the other hand, as the bulk modulus relates changes in the pressure p and in the volume by $Kdv + vdp = 0$, we also have

$$K \frac{\partial \phi}{\partial x} = -A \frac{\partial p}{\partial t} \quad (3.22)$$

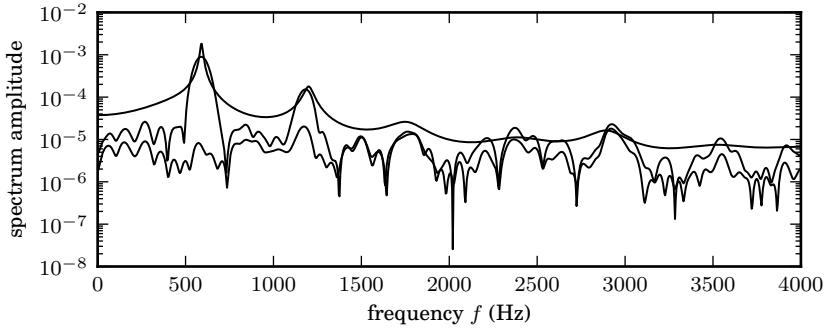


Figure 3.4: **Spectral View.** spectrum of the signal of the figure 3.3, estimated by non-parametric (fft) method and by the frequency response of the synthesis filter. The spectrum of the prediction error is also displayed.

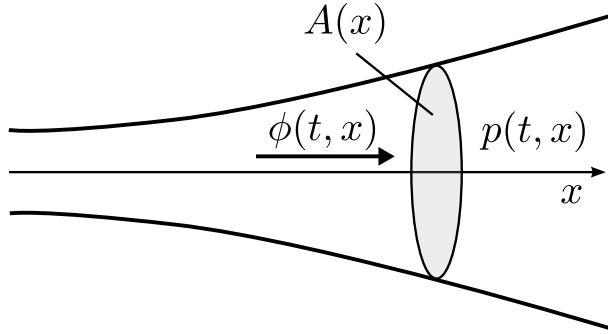


Figure 3.5: the vocal tract: horn model.

The combination of equations (3.21) and (3.22) yield **Webster's Equation**

$$\boxed{\frac{1}{c^2} \frac{\partial^2 p}{\partial t^2} - \frac{1}{A} \frac{dA}{dx} \frac{\partial p}{\partial x} - \frac{\partial^2 p}{\partial x^2} = 0} \quad (3.23)$$

where c , the wave velocity in the media is given by:

$$c = \sqrt{\frac{K}{\rho}} \quad (3.24)$$

Discrete Modelling. A common simplification of the horn model is to trade the continuous change in the cross-sectional area $A(x)$ for a tube made of a finite number of cylindrical sections of equal length L whose cross-sectional area A_k is a function of the section index k . Consider the pressure p_k in the section k as the superposition of right and left-travelling waves $p_k(t, x) = p_k^+(x - ct) - p_k^-(x + ct)$. Stating that the pressure is continuous at the section boundary x leads to the system of equations

$$\begin{aligned} p_{k+1}^+(x - ct) &= (1 - r_k^+) p_k^+(x - ct) + r_{k+1}^- p_{k+1}^-(x + ct) \\ p_k^-(x + ct) &= (1 - r_{k+1}^-) p_{k+1}^-(x + ct) + r_k^+ p_k^+(x - ct) \end{aligned} \quad (3.25)$$

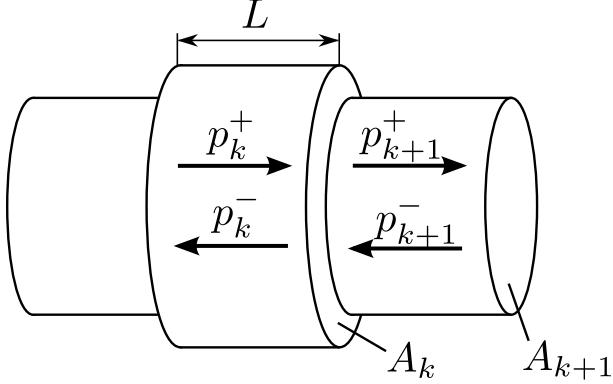


Figure 3.6: the vocal tract: discrete tube model.

where r_k^+ and r_{k+1}^- are **reflection coefficients**. The air flow $\phi_k(t, x) = \phi_k^+(x - ct) - \phi_k^-(x + ct)$ is also continuous at the section boundary. A Fourier decomposition of the waves and the use of equation (3.21) show that it is related to the pressure by

$$\frac{p_k^\pm}{\phi_k^\pm} = \pm Z_k$$

where Z_k is the **impedance**, given in each section by

$$Z_k = \frac{c\rho}{A_k} = \frac{\sqrt{K\rho}}{A_k} \quad (3.26)$$

The continuity of the air flow at the position x provides for all time the equations

$$A_k p_k^+(x - ct) + A_k p_k^-(x + ct) = A_{k+1} p_{k+1}^+(x - ct) + A_{k+1} p_{k+1}^-(x + ct)$$

which, coupled with the system of equations (3.25) leads to

$$r_k^+ = -r_{k+1}^- = \frac{A_{k+1} - A_k}{A_{k+1} + A_k}. \quad (3.27)$$

Ladder and Lattice Filters. In a given tube section, the pressure waves travel unchanged at the speed c . Given that the tube section is of length L , the time needed to go from one boundary of the section to the other is L/c . As a consequence, the transformation between the values of p^+ and p^- from the left of one section boundary to the left of the next section boundary on the right may be modelled as the junction depicted on the left of figure 3.2.2.

Now, if we want to follow what happens to the pressure wave p^+ travelling to the right, we may introduce a variable \tilde{p}^+ that compensates for the delay in the wave propagation as well as the attenuation (or amplification) at the sections boundary. We apply the same treatment to \tilde{p}^- so that

$$\tilde{p}_{k+1}^\pm(t) = \frac{1}{1 - r_k^\pm} p_{k+1}^\pm(t + L/c) \quad (3.28)$$

Straightforward computations show that the equations satisfied by the corresponding variables are described by the lattice junction depicted on the right of the figure 3.2.2.

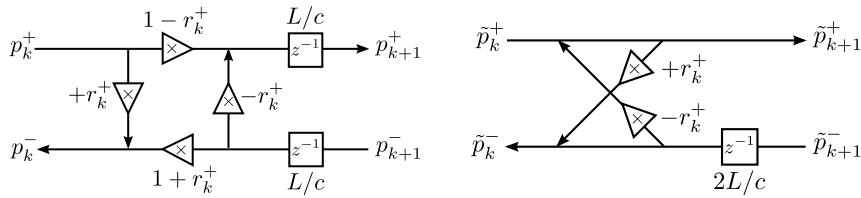


Figure 3.7: Kelly-Lochbaum junction: ladder form (left) and lattice form (right)

Lattice Filters in Linear Predictive Coding. When it comes to the implementation of synthesis filters that model the vocal tract, lattice filters – implemented as a serial connexions of lattice junctions – are often preferred to classic (register-based) implementations of the autoregressive filters. Their parameters – the reflection coefficients – are easy to interpret and also, when the synthesis filter is determined by the auto-correlation method, the Levenson-Durbin or Schur algorithm may be used to compute them directly instead of the linear regression coefficients a_i . Moreover, these algorithms are recursive and have a $\mathcal{O}(m^2)$ complexity where m is the predicton order, better than the typical $\mathcal{O}(m^3)$ of the least-square resolution needed to compute the a_i .

Finally, lattice filters are stable as long as the reflexion coefficients are between -1 and 1 . As a consequence, we can easily perform a quantization of these coefficients that will preserve the stability of the synthesis filter. A classic choice is the logarithmic quantization of the area-ratio A_{k+1}/A_k , that is, because of the equation (3.27), the uniform quantization of

$$\text{LAR}_k = \log \frac{1 + r_k^+}{1 - r_k^+}. \quad (3.29)$$

Pitch Analysis

The prediction error of the voice fragment displayed at the bottom of figure 3.3 still displays some structure : a white noise plus a quasi-periodic sequence of impulses. As the short-term prediction has inverted the vocal tract filter, what we are looking at is actually the sequence of **glottal pulses**. The duration between two pulses is the voice **pitch period**, ts inverse is the speech **fundamental frequency**. When this periodic structure is present after short-term predicton, the speech fragment is said to be **voiced** and when it's not, it is **unvoiced**.

The simplest kind voiced/unvoiced classifier is based on the **autocorrelation** of the short-term prediction error (see for example [Mar72]). In a given data frame, we select a subframe, typically at the end, and compare it with all the subframes of equal size within the frame by computing the normalized scalar product between the two vectors. Values of (the modulus of) the correlation near 1 correspond to two subframes that are – up to a gain – almost equal.

```
def ACF(data, frame_length):
    frame = data[-frame_length:]
    frame = frame / norm(frame)
    past_length = len(data) - frame_length
    correl = zeros(past_length + 1)
    for offset, _ in enumerate(correl):
        past_frame = data[past_length-offset:past_length-offset+frame_length]
        past_frame = past_frame / norm(past_frame)
```

```
correl[offset] = dot(past_frame, frame)
return correl
```

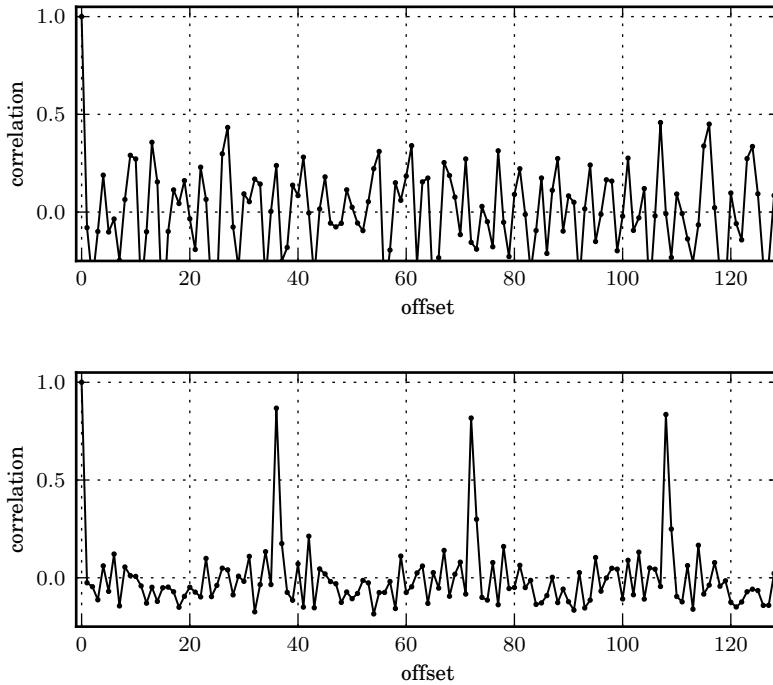


Figure 3.8: normalized autocorrelation of the prediction errors for the speech fragment of figure 3.3 with a reference window of 32 samples. The top graph corresponds to an unvoiced signal and the bottom one to a voiced signal with a pitch period of 36 samples.

These kind of method will therefore rely on the selection of autocorrelation threshold to distinguish between voiced and unvoiced signals and localization of autocorrelation maxima to estimate the pitch period. Care must be taken not to select a multiple of the pitch period instead.

Long-Term Prediction

Given a reference subframe y and a subframe x offsetted by the pitch period p we can compute the best linear approximation of y in terms of x , that is, the gain k , solution of $k = \arg \min_{\kappa} \|y - \kappa x\|^2$. It is given by

$$k = \frac{x^t y}{\|x\|^2} \quad (3.30)$$

Once again, what we have done is a prediction, but a long-term prediction, applied to the residual of the short-term prediction. If x_n denotes the error of the short-term prediction, the error e_n after the additional long-term prediction is given by

$$x_n = kx_{n-p} + e_n \quad (3.31)$$

This equation models an auto-regressive synthesis filter whose diagram is given in figure 3.2.2

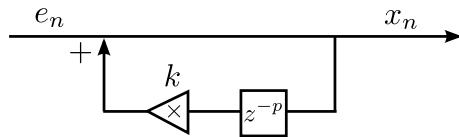


Figure 3.9: LTP synthesis filter

3.2.3 Linear Prediction Coding

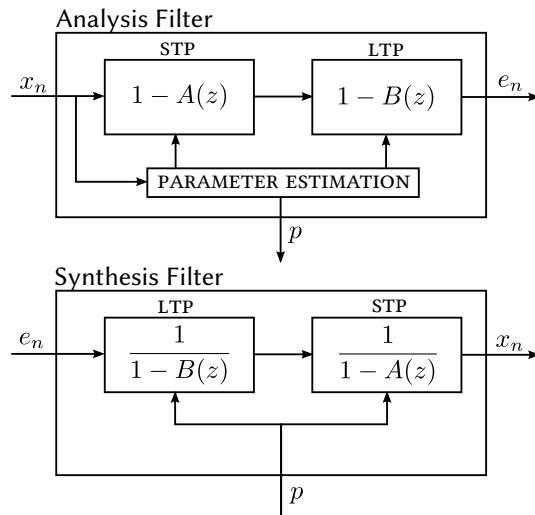


Figure 3.10: LPC analysis and synthesis filters diagram

The use of short-term and long-term linear prediction method may be used in several ways to compress voice information. The algorithms that follow this path are generally referred to as **Linear Predictive Coding (LPC)**. “Pure” LPC algorithms encode the prediction parameters and the residual power but do not keep any extra information on the prediction residual ; this approach is consistent with the belief that a good prediction produces a residual which is a white noise. The voice is reconstructed by the injection of a synthetic white noise into the synthesis filter.

Adaptive Predictive Coding (APC) is also called **Residual-Excited Linear Prediction (RELP)** : in order to have a reconstructed voice with a higher quality, the residual information is not discarded but quantized and transmitted along with the prediction parameters.

This kind of approach has a major drawback: the quantization typically aims at the minimization of the quantization error *of the residual*, a quantity that has little to do with the error induced on the voice itself. The **Code-Excited Linear Prediction (CELP)**

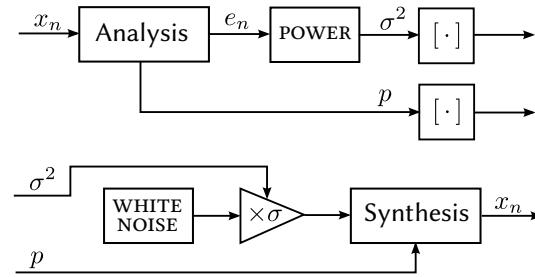


Figure 3.11: “Pure” LPC analysis and synthesis diagrams

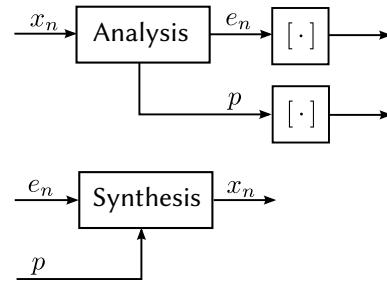


Figure 3.12: APC/REPL analysis and synthesis diagrams

approach solves that issue by discarding the residual entirely and by trying instead several excitation signals among a finite **codebook**, apply to them the synthesis filter, and look for the output that matches the more closely the voice data.

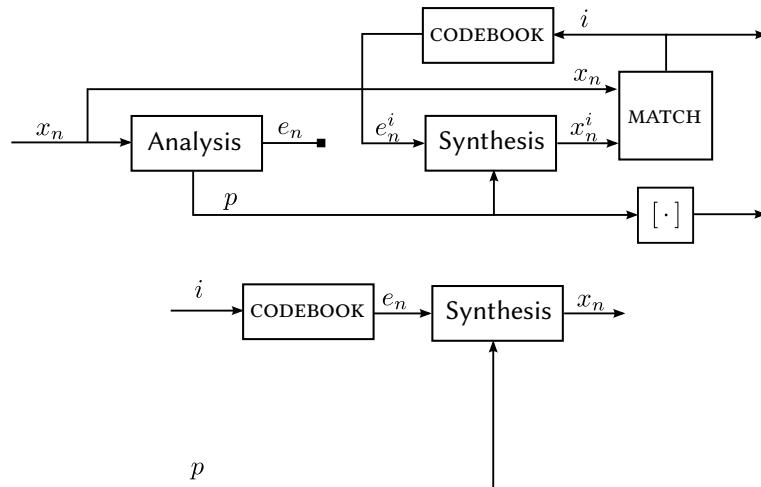


Figure 3.13: APC/REPL analysis and synthesis diagrams

Chapter 4

Spectral Methods

In the context of audio signal processing, spectral methods refer to algorithms that rely on the representation of signals as superposition of sinusoids. Such a decomposition – the spectrum of the signal – is obtained with the Fourier transform ; efficient computations of the spectrum are possible with the fast Fourier transform algorithms.

Spectral methods are crucial in the study of filters such as the finite impulse response filters or autoregressive filters and more generally to understand any transformation based on a convolution ; they are also key in multirate systems that achieve compression through data rate reduction.

4.1 Signal, Spectrum, Filters

A discrete-time signal x with sample time Δt is a function defined on

$$\mathbb{Z}\Delta t = \{k\Delta t, k \in \mathbb{Z}\}. \quad (4.1)$$

This definition is nothing but a convenient packaging of the sequence of values (x_n) , $n \in \mathbb{Z}$, with the sample time Δt into a unique mathematical object.

We investigate in this section the representation of a discrete-time signal as a superposition of sinusoids. Let's start the search for such a spectral representation with a real-valued discrete-time signal x . Given a non-negative frequency f , a sinusoid is determined uniquely by its amplitude $a(f) \geq 0$ and – provided that $a(f) \neq 0$ – its phase $\phi(f) \in [-\pi, \pi]$. We therefore search for a pair of functions a and ϕ – subject to the above constraints – such that

$$\forall t \in \mathbb{Z}\Delta t, x(t) = \int_0^{+\infty} a(f) \cos(2\pi ft + \phi(f)) df \quad (4.2)$$

Alternatively, we may use complex exponentials instead of sinusoids: decompose the cos in the previous equation and set

$$x(f) = \begin{cases} 1/2 \times a(f)e^{i\phi(f)} & \text{if } f \geq 0 \\ 1/2 \times a(-f)e^{-i\phi(-f)} & \text{otherwise.} \end{cases}, \text{ or equivalently } \begin{aligned} a(f) &= 2|x(f)| \\ \phi(f) &= \angle x(f) \end{aligned}$$

The equation (4.2) becomes

$$x(t) = \int_{-\infty}^{+\infty} x(f) \exp(i2\pi ft) df \quad (4.3)$$

and the only constraint that holds on the complex-valued function $x(f)$, defined for any real frequency f , is the symmetry constraint

$$x(-f) = \overline{x(f)} \quad (4.4)$$

This relation specifically ensures that the complex exponentials in (4.3) always combine to produce a real-valued signal $x(t)$, so that the values of $x(f)$ for negative frequency hold no extra information and are merely an artifact of the complex exponential representation.

However we can drop this symmetry constraint if we allow complex-valued signals $x(t)$ in the first place and then these negative frequencies values are no longer redundant. At the same time, we notice that (4.3) still makes sense if we consider vector-valued signals $x(t) \in \mathbb{C}^p$, so given this higher generality, and also the better mathematical tractability of (4.3), we will stick to this formulation of the problem.

At this stage we clearly search for summable functions – that is $x(f) \in L^1(\mathbb{R}, \mathbb{C}^p)$ – so that the right-hand side of the equation makes sense. Still, the problem of finding a solution $x(f)$ to (4.3) is not well posed: let Δf be the signal sampling frequency, defined by

$$\boxed{\Delta f \times \Delta t = 1.} \quad (4.5)$$

If $x(f)$ is a solution to the equation, so is $f \mapsto x(f - k\Delta f)$ for any $k \in \mathbb{Z}$ ¹: the spectral content of $x(t)$ is only determined up to frequency shifts that are multiples of the sampling frequency Δf . A way to remove this ambiguity in $x(f)$ is to reassign to any spectral component at the frequency f the smallest frequency $f - k\Delta t$ that is the nearest from 0 among any possible values of $k \in \mathbb{Z}$ – that frequency has to be in $[-\Delta f/2, +\Delta f/2]$. In other words, for any spectral component of the signal, we make a low-frequency interpretation. Mathematically, that means that we replace $x(f)$ with²

$$x(f) \rightarrow x'(f) = \begin{cases} \sum_{k \in \mathbb{Z}} x(f - k/\Delta t) & \text{if } f \in [-\Delta f/2, +\Delta f/2], \\ 0 & \text{otherwise.} \end{cases}$$

and therefore if we rename $x(f)$ this particular solution $x'(f)$, we end up with the search for an integrable function $x(f) : [-\Delta f/2, +\Delta f/2] \rightarrow \mathbb{C}^p$ solution of the

¹Indeed, we have

$$\begin{aligned} \int_{-\infty}^{+\infty} x(f + k\Delta f) \exp(i2\pi ft) df &= \int_{-\infty}^{+\infty} x(f) \exp(i2\pi(f - k\Delta f)t) df \\ &= \int_{-\infty}^{+\infty} x(f) \exp(i2\pi ft) \exp(-i2\pi kt\Delta f) df \end{aligned}$$

As $t = n\Delta t$ for a $n \in \mathbb{Z}$, $\exp(-i2\pi kt\Delta f) = \exp(-i2\pi kn) = 1$ and consequently

$$\int_{-\infty}^{+\infty} x(f + k\Delta f) \exp(i2\pi ft) df = \int_{-\infty}^{+\infty} x(f) \exp(i2\pi ft) df$$

²this is legitimate because for any $t \in \mathbb{Z}\Delta t$

$$\int_{-\infty}^{+\infty} x(f) \exp(i2\pi ft) df = \int_{-\Delta f/2}^{+\Delta f/2} \left[\sum_{k \in \mathbb{Z}} x(f - k/\Delta t) \right] \exp(i2\pi ft) df$$

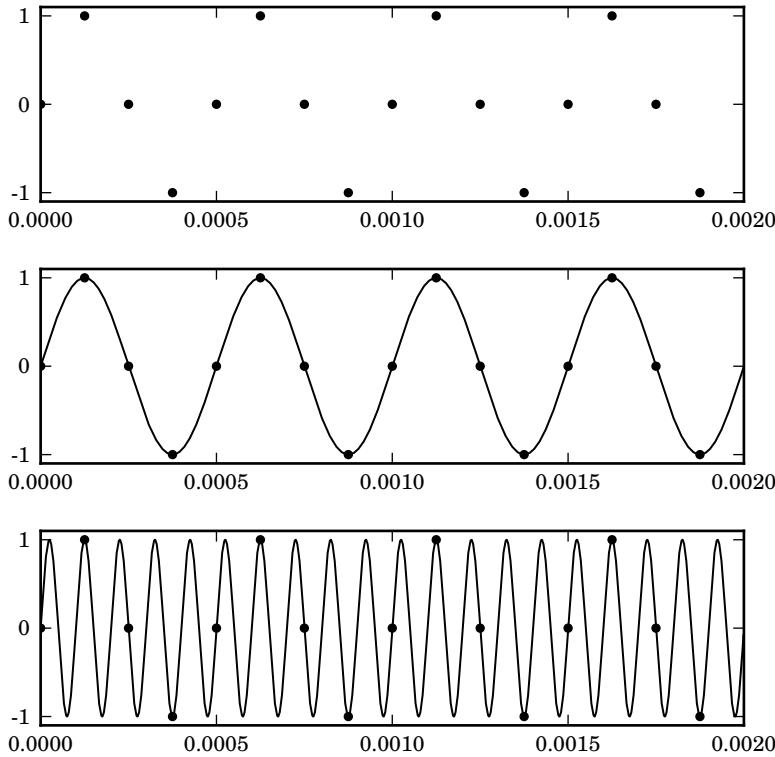


Figure 4.1: A signal sampled at 8kHz (top) and two valid interpretation of its spectral content : either a pure 2 kHz sine (mid) or 10 kHz sine (bottom).

equation

$$\forall t \in \mathbb{Z}\Delta t, x(t) = \int_{-\Delta f/2}^{+\Delta f/2} x(f) \exp(i2\pi ft) df \quad (4.6)$$

The highest frequency value in the integration interval, $\Delta f/2$ is called the **Nyquist frequency** associated to the sample time Δt .

We notice at this stage that equation (4.6) defines $x(n\Delta t)$ as the n -th Fourier coefficient of the Fourier serie associated to $x(f)$. As a consequence, if we make the assumption that the signal $x(t)$ is of finite energy, that is mathematically $x(t) \in L^2(\mathbb{Z}\Delta t, \mathbb{C}^p)$ or

$$\sum_{n \in \mathbb{Z}\Delta t} |x(t)|^2 < +\infty$$

then the function $x(f)$ is uniquely defined (almost everywhere). It also belongs to $L^2([-\Delta f/2, +\Delta f/2], \mathbb{C}^n)$

$$\int_{-\Delta f/2}^{\Delta f/2} |x(f)|^2 df < +\infty$$

and satisfies³

$$x(f) = \Delta t \sum_{n \in \mathbb{Z}\Delta t} x(t) \exp(-2i\pi f t) \quad (4.7)$$

The transform – denoted \mathcal{F} – that maps a signal time-domain representation $x(t)$ to its frequency-domain representation or **spectrum** $x(f)$ is **(discrete-time) Fourier transform (DTFT)**.

Parseval's theorem also yields

$$\int_{-\Delta f/2}^{+\Delta f/2} |x(f)|^2 df = \Delta t \sum_{t \in \mathbb{Z}\Delta t} |x(t)|^2 \quad (4.8)$$

which means that we may measure the energy of the signal by summing either the energy of each sample in the time domain, or the energy density of all signal spectral components.

The category of finite energy signals is sufficient most of the time but still does not encompass every signal we'd like to consider ... and to begin with, pure tones! To perform the spectral decomposition of signals that have an infinite energy, we need to go beyond Δf -periodic (locally) integrable functions of the frequency f and consider instead Δf -periodic (vector-valued complex) measures. For such a measure $x(f)$, $x(t)$ is represented by the integral

$$\forall t \in \mathbb{Z}\Delta t, x(t) = \int_{(-\Delta f/2)^-}^{(+\Delta f/2)^-} \exp(i2\pi f t) dx(f) \quad (4.9)$$

In practice, we don't need measures with singular parts which means that every measure spectra we need to consider has on the interval $[-\Delta f/2, +\Delta f/2]$ the form

$$x(f) = x_1(f) + \sum_i a_i \delta(f - f_i) \text{ where } \begin{cases} x_1(f) \in L^1([-\Delta f/2, \Delta f/2], \mathbb{C}^n) \\ \sum_i |a_i| < +\infty \end{cases}$$

And then, every dirac component in the frequency domain represents a pure tone as

$$\int_{(-\Delta f/2)^-}^{(+\Delta f/2)^-} \exp(i2\pi f t) d\delta(f - f_i) = \exp(i2\pi f_i t)$$

Therefore the equation (4.9) reduces to

$$x(t) = x_1(t) + \sum_i a_i \exp(i2\pi f_i t)$$

4.1.1 Convolution and Filters

Consider two scalar discrete-time signals x and y with a common sample time Δt . We assume for convenience that there is a $t_0 \in \mathbb{Z}\Delta t$ such that $x(t) = y(t) = 0$ for

³as a limit in $L^2([-\Delta f/2, +\Delta f/2], \mathbb{C}^n)$ or pointwise but only almost anywhere (Carleson). Stronger convergence (such as uniform convergence) may be obtained under the assumption that $x(f)$ is continuously differentiable.

any $t \leq t_0$. We define the **convolution** between x and y as the discrete-time signal $x * y$ with sample time Δt such that

$$(x * y)(t) = \Delta t \sum_{t' \in \mathbb{Z}\Delta t} x(t')y(t - t') \quad (4.10)$$

The assumptions made on the signals x and y ensure that for every value of t , the sum in the right-hand side of (4.10) has only a finite number of non-zero values. These assumptions may be relaxed in several ways ; for example we may assume that x and y belong to $L^2(\mathbb{Z}\Delta t, \mathbb{C})$ and define $x * y$ as a bounded signal.

We notice that the convolution is an associative and commutative operation. Moreover, the spectrum of the convolution between two signals is the product of the signal spectra:

$$(x * y)(f) = x(f)y(f) \quad (4.11)$$

Proof. The discrete-time Fourier transform of $x * y$ satisfies

$$(x * y)(f) = \Delta t \sum_{t \in \mathbb{Z}\Delta t} \left[\Delta t \sum_{t' \in \mathbb{Z}\Delta t} x(t')y(t - t') \right] \exp(-2i\pi ft)$$

Notice that $\exp(-2i\pi ft) = \exp(-2i\pi t' f) \exp(-2i\pi f(t - t'))$, set $\tau = t - t'$ and conclude with

$$(x * y)(f) = \left[\Delta t \sum_{t' \in \mathbb{Z}\Delta t} x(t') \exp(-2i\pi ft') \right] \left[\Delta t \sum_{\tau \in \mathbb{Z}\Delta t} y(\tau) \exp(-2i\pi f\tau) \right]$$

■

A **filter** is a convolution operator $u \mapsto y$ with kernel h :

$$u \mapsto y = h * u \quad (4.12)$$

or equivalently in the frequency domain:

$$y(f) = h(f)u(f) \quad (4.13)$$

The function $h(f)$ is the **frequency response** of the filter. We define the **(unit) impulse** as the signal $\delta : \mathbb{Z}\Delta t \rightarrow \mathbb{C}$:

$$\delta(t) = \begin{cases} 1/\Delta t & \text{if } t = 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.14)$$

The **(unit) impulse** is a unit – in the algebraic sense – for the convolution operator : for any signal x , $x * \delta = \delta * x = x$. For this reason, when $u = \delta$, the output of the filter (4.12) is $y = h$ and h is called the **filter impulse response**.

Convolution operators are a very general class of signal transformations. Consider an operator L that maps any finite discrete-time signal u with sample time Δt to a signal with the same sample time and such that for any finite input signals u and v :

$$\forall \lambda, \mu \in \mathbb{C}, L(\lambda u + \mu v) = \lambda L(u) + \mu L(v) \quad (4.15)$$

$$\forall T \in \mathbb{Z}\Delta t, L(t \mapsto u(t - T)) = t \mapsto L(u)(t - T) \quad (4.16)$$

Such a **linear and time-invariant(LTI)** operator is a convolution operator. Indeed, we have:

$$L(u) = L\left(\sum_{t' \in \Delta t \mathbb{Z}} u(t')\delta(t - t')\right) = \sum_{t' \in \Delta t \mathbb{Z}} u(t')L(\delta)(t - t') = u * L(\delta)$$

As a consequence, a finite response impulse filter (FIR) is a convolution operator: the definition equation

$$y(t) = \sum_{n=0}^{N-1} a_n u(t - n\Delta t)$$

corresponds to $y = h * u$ with

$$h(t) = \begin{cases} a_{t/\Delta t}/\Delta t & \text{if } t \in \{0, \dots, (N-1)\Delta t\}, \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, an autoregressive system whose evolution is given by

$$y(t) = \sum_{n=0}^{N-1} a_n y(t - (n-1)\Delta t) + u(t)$$

is a convolution operator but whose impulse response is not finite.

4.2 Finite Signals

Concrete digital signals are finite because only a finite number of samples may be stored in a finite memory. We usually represent a finite sequence of values x_0, \dots, x_{N-1} and a reference step time Δt , with a finite causal signal $x : \mathbb{Z}\Delta t \mapsto \mathbb{C}$ where the missing values are replaced with 0:

$$x(t) = \begin{cases} x_{t/\Delta t} & \text{if } t \in \{0, \Delta t, \dots, (N-1)\Delta t\}, \\ 0 & \text{otherwise.} \end{cases} \quad (4.17)$$

This signal is said to be **causal** because $x(t) = 0$ whenever $t < 0$ and **finite** because it has only a finite number of non-zero values.

If the two finite causal signals x and y correspond to the finite sequences x_0, \dots, x_{N-1} and y_0, \dots, y_{M-1} their convolution $z = x * y$ is also a finite causal signal and corresponds to the sequence (z_0, \dots, z_{M+N-2}) where

$$z_k = \Delta t \sum_{(i,j) \in S_k} x_i y_j \quad \text{with} \quad S_k = \{(i, j) \in \{0, \dots, N-1\} \times \{0, \dots, M-1\}, i+j = k\} \quad (4.18)$$

The NumPy implementation of the operation is the function `convolve` and it assumes that $\Delta t = 1$. For example

```
>>> x = array([0.5, 0.5])
>>> y = array([0.0, 1.0, 2.0, 3.0, 4.0])
>>> z = convolve(x, y)
>>> z
array([ 0. ,  0.5,  1.5,  2.5,  3.5,  2. ])
```

Now, this approach gives us a practical method to implement filters as long as their impulse response h is finite and causal – that is when filters have a **finite impulse response (FIR)**. If h corresponds to the finite sequence h_0, \dots, h_{M-1} and the filter is to be applied to the finite signal u , then the output y corresponds to

```
>>> y = dt * convolve(h, u)
```

4.2.1 Design of Low-Pass Filters

Let $f_c \in (0, \Delta f/2)$ be the **cutoff frequency** of our lowpass filter. What it means is that we want is a filter that generates from a signal u an output signal y such that

$$y(f) = \begin{cases} x(f) & \text{if } f \in (0, f_c) \\ 0 & \text{if } f \in (f_c, \Delta f/2) \end{cases}$$

As the filter operation $y = h * u$ translates into $y(f) = h(f)u(f)$ in the Fourier domain (see equation (4.11)), the frequency response of the filter shall satisfy

$$h(f) = \begin{cases} 1 & \text{if } f \in (0, f_c) \\ 0 & \text{if } f \in (f_c, \Delta f/2) \end{cases}$$

and because h is a real signal, $h(f) = \overline{h(-f)} = h(-f)$ if $f \in (-\Delta f/2, 0)$. As a consequence, the inverse DTFT formula (4.6) provides

$$h(t) = \int_{-\Delta f/2}^{+\Delta f/2} h(f) \exp(2i\pi ft) df = \int_{-f_c}^{f_c} \exp(2i\pi ft) df, t \in \mathbb{Z}\Delta t$$

and after straightforward computations, with the sine cardinal sinc defined as

$$\text{sinc } x = \frac{\sin \pi x}{\pi x} \text{ if } x \neq 0 \text{ and } \text{sinc } 0 = 1 \quad (4.19)$$

we end up with

$$h(t) = 2f_c \text{sinc } 2f_c t, t \in \mathbb{Z}\Delta t. \quad (4.20)$$

An concrete implementation of such a filter has to overcome several issues. First of all, an implementation as a FIR requires a finite number of non-zero values of $h(t)$ only. We therefore typically replace $h(t)$ with an impulse response that is equal to $h(t)$ for $|t| \leq N$ and 0 for $|t| > N$ and end up with a $2N + 1$ -tap filter. Then, the implementation has to be causal: the $2N + 1$ coefficients are shifted to correspond to the indices 0, 1, \dots , $2N$ which effectively induces a delay of N samples during the filtering (see fig. 4.2, bottom figure). The generation of such low-pass filters may be implemented as

```
def low_pass(fc, dt=1.0, window=ones):
    def h(n):
        t = arange(-0.5 * (n-1), 0.5 * (n-1) + 1) * dt
        return 2 * fc * sinc(2 * fc * t) * window(n)
    return h
```

and used as follows to perform for example a 31-tap low-pass filtering of a 44100 Hz at the cutoff frequency of 8000 Hz:

```
>>> N = 15
>>> h = low_pass(fc=8000.0, dt=1.0/44100.0)(2*N+1)
>>> y = dt * convolve(h, u)
```

Note that $\text{len}(y)$ is equal to $\text{len}(u) + 2*N$. A restriction of the output that compensates for the induced delay and has the same size as the original signal is obtained as $y[N:-N]$.

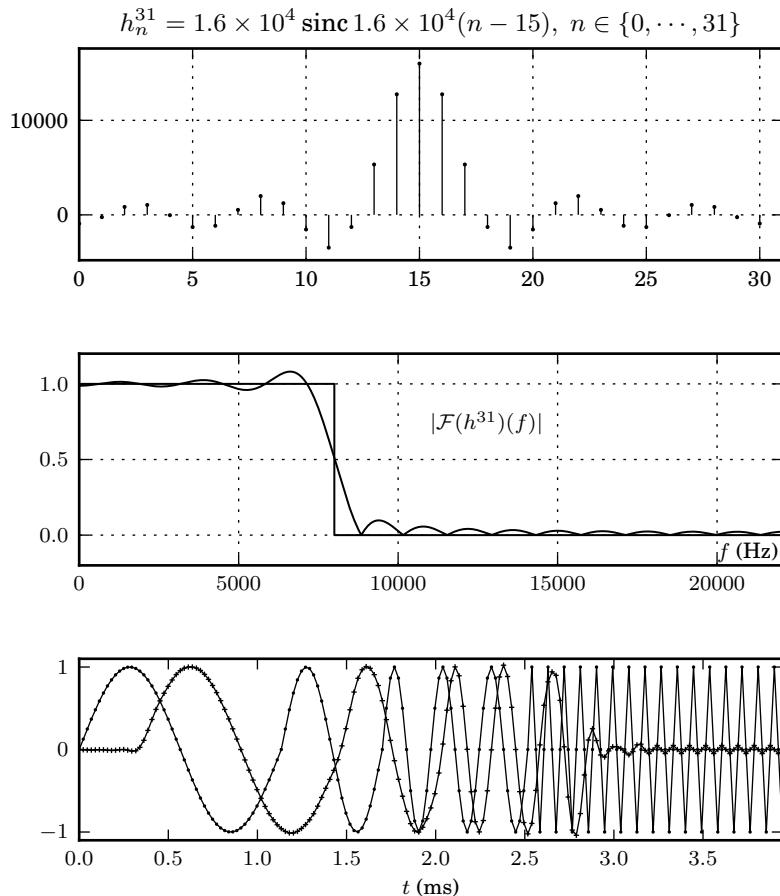


Figure 4.2: top : impulse response of a 31-tap low-pass filter with a cutoff frequency of $f_c = 8000$ Hz. Middle : frequency response (gain) of this filter. Bottom : a signal before (dots) and after (plus signs) low-pass filtering.

The optional `window` argument (that defaults to a rectangular window) is useful to reduce the **Gibbs phenomenon** that we may observe in the frequency response of the filter (see fig. 4.2): an oscillation of the frequency response that may result in overshoots in the filter outputs. Windows such as `hanning`, `bartlett`, `blackman`, etc. are available in NumPy.

4.2.2 Spectrum Computation

Given a finite causal signal x with sample time Δt and possibly non-zero values $x_0 = x(0), x_1 = x(\Delta t), \dots, x_{N-1} = x((N-1)\Delta t)$, the spectrum $\mathcal{F}x$ of x is given by the

formula:

$$\mathcal{F}x(f) = \Delta t \sum_{n=0}^{N-1} x_n \exp(-i2\pi fn\Delta t) \quad (4.21)$$

The signal x being represented as the NumPy array x and the sample time Δt as a float dt , a simple representation Fx of the spectrum $\mathcal{F}x$ – as a function taking arrays of frequencies as arguments – is given by:

```
nx = len(x)
n = reshape(arange(nx), (nx, 1))
def Fx(f):
    f = ravel(f)
    f = reshape(f, (1, len(f)))
    return dt * dot(x, exp(-1j * 2 * pi * dt * n * f))
```

The high-order programming support in Python actually allow us to automate the definition of this function and to represent the Fourier transform itself as a function F , that takes x and dt as arguments and returns the spectrum function Fx .

Fourier transform - straightforward implementation

```
def F(x, dt=1.0):
    nx = len(x)
    n = reshape(arange(nx), (nx, 1))
    def Fx(f):
        f = ravel(f)
        f = reshape(f, (1, len(f)))
        return dt * dot(x, exp(-1j * 2 * pi * dt * n * f))
    return Fx
```

The main issue with this computation of the spectrum is performance: assume that you intend to compute N values of the spectrum, that is, as many values as there are in the signal. Then the number of sums and product needed to compute $F(x)(f)$ is $\mathcal{O}(N^2)$.

An alternate idea is to compute enough spectrum values and then to use interpolation to build an approximation of the spectrum anywhere. If we decide to use N distinct spectrum values, it makes sense to compute regularly sampled values of $\mathcal{F}x(f)$ on the interval $[0, \Delta f]$ – the spectrum being Δf -periodic, there is no point going beyond this interval. We are therefore interested only in the frequencies

$$f_k = \frac{k}{N} \Delta f, \quad k = 0, \dots, N-1 \quad (4.22)$$

and in the values $\hat{x}_k = \mathcal{F}x(f_k)$ given by:

$$\hat{x}_k = \sum_{n=0}^{N-1} x_n \exp\left(-i2\pi \frac{kn}{N}\right), \quad k = 0, \dots, N-1. \quad (4.23)$$

The transformation from the vector (x_0, \dots, x_{N-1}) to the vector $(\hat{x}_0, \dots, \hat{x}_{N-1})$ is called the **discrete Fourier transform (DFT)**:

$$\text{DFT} \left[\begin{array}{ccc} \mathbb{C}^N & \rightarrow & \mathbb{C}^N \\ (x_0, \dots, x_{N-1}) & \mapsto & (\hat{x}, \dots, \hat{x}_{N-1}) \end{array} \right] \quad (4.24)$$

As we noted before, the straightforward implementation of the DFT has a $\mathcal{O}(N^2)$ complexity. Fortunately there is a family of algorithms called **fast Fourier transforms (FFT)** that achieve $\mathcal{O}(N \log N)$ performance instead. In NumPy a fast Fourier

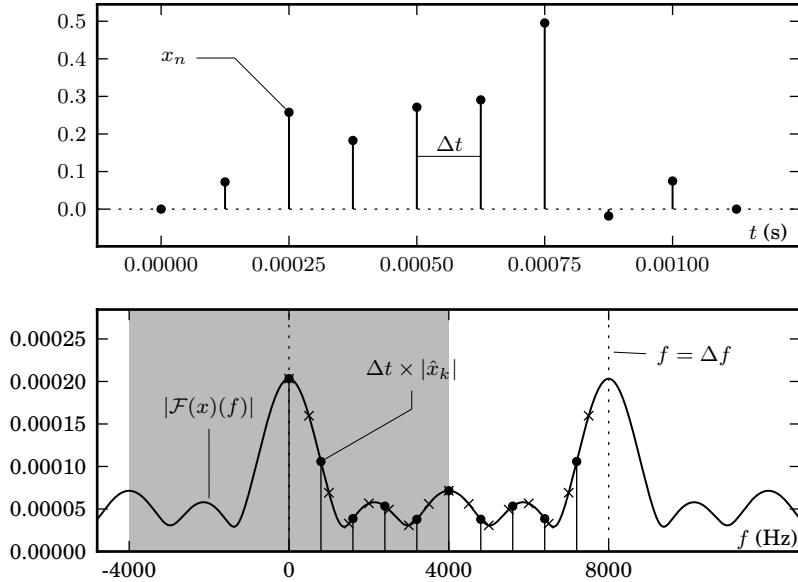


Figure 4.3: temporal and spectral representation of a 10-sample 8kHz signal $x(t)$. The 10-point DFT data are displayed as dots, whereas the 16-point DFT from the zero-padded signal are displayed as crosses.

transform is available as the `fft` function in the module `numpy.fft`⁴. With the help of this function, we may implement an alternative Fourier transform `F`, based on the discrete Fourier transform data and 0-order interpolation.

Fourier transform - FFT-based implementation

```
def F(x, dt=1.0):
    nx = len(x)
    fft_x = fft(x)
    def Fx(f):
        k = (round_((ravel(f) * nx * dt)) % nx).astype(uint64)
        return dt * fft_x[k]
    return Fx
```

We can actually compare the performance of the two approaches by measuring the time needed to compute the values $x(f_k)$, $k = 0, \dots, N - 1$, for a signal x of length N . The results are displayed in figure 4.4 in a log-log scale. The results for the straightforward computation method (dashed curve) are consistent with the $\mathcal{O}(N^2)$ bound as the curve exhibit an asymptotic slope of 2. For the FFT-based computation, the situation is more complex as the computation times varies strongly with respect to the signal length. The lower envelope of the curve is given by data points that correspond to signals whose length is a power of two (dotted data). For those signals, the asymptotic slope is 1, consistent with the $\mathcal{O}(N \log N)$ estimate. However, the performance may be far worse for arbitrary length, the upper envelope being $\mathcal{O}(N^2)$ again and is obtained for signal whose length is a prime number. This is a common artifact of

⁴The NUMPY implementation is a transcription in C of the Fortran-77 `fftpack` (<http://www.netlib.org/fftpack/>)

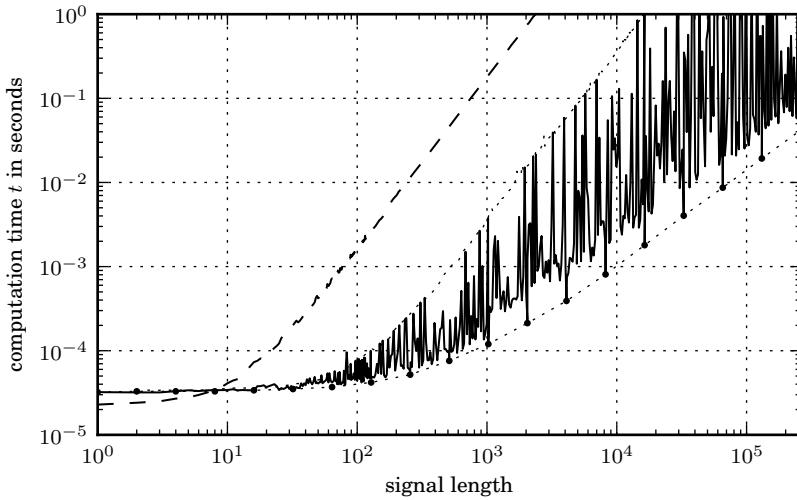


Figure 4.4: **spectrum computation performance:** computation time of $F(x)(f)$ as a function of $nx = \text{len}(x)$ with $f = \text{arange}(0, dt, dt/nx)$ for a straightforward implementation (dashed curve) and a FFT-based one (solid curve) ; dots correspond to power-of-two signal length data. Reference platform: Intel i7 Q820 1.73GHz CPU, 6GiB memory.

many FFT algorithms: they behave well when the signal length has an integer decomposition that consists of many small prime numbers, the best case being a power of two and the worst, a large prime number⁵.

To cope with this fact, we may introduce zero-padding of the original signal: we append as many 0 values as necessary to the original vector so that its length is a power of two. We still compute data on the original signal spectrum as the signals values were 0 anyway. Note that NumPy fft implements 0-padding when it is given as a second argument a desired length for the fft vector larger than the signal length. As a consequence, we can support a power-of-two version of the spectrum computation with the following code.

Fourier transform - FFT-based, power-of-two.

```
def F(x, dt=1.0):
    nx = int(2**ceil(log2(len(x))))
    fft_x = fft(x, nx)
    def Fx(f):
        k = (round_((ravel(f) * nx * dt)) % nx).astype(uint64)
        return dt * fft_x[k]
    return Fx
```

Obviously, zero-padding may also be used to obtain a larger power of 2 in order to get more spectrum data. An additional parameter may be given to define the minimum length of the DFT.

⁵This is explained by the fact that those algorithms use a divide-and-conquer approach to solve the problem. However, there are algorithms that may achieve an asymptotic $\mathcal{O}(N \log N)$ performance, even for signals of length a prime number, see for example [FJ05].

Fourier transform - FFT-based, power-of-two, arbitrary length.

```
def F(x, dt=1.0, n=0):
    nx = len(x)
    nx = max(n, nx)
    nx = int(2**ceil(log2(nx)))
    fft_x = fft(x, nx)
    def Fx(f):
        k = (round_((ravel(f) * nx * dt)) % nx).astype(uint64)
        return dt * fft_x[k]
    return Fx
```

The signal we want to analyze has often more values than the ones contained in x . It may for example be – at least conceptually – be infinite, for example if it is a pure tone. The FFT-based spectral analysis is therefore based on a window of the original signal ; the most common choice is a rectangular window where we select some of the valued of the signal (multiply by 1) and implicitly consider that all other values are 0 (multiply by 0). Using a multiplication by a window function whose behavior is smoother on the window boundary is a classical method to improve the resolution of harmonics in the spectrum. Refer for example to [Har78] for a discussion on this subject and a comparison of the usual windows (such as bartlett, hamming, hanning, etc.). A version of the spectrum that support windows is given by;

Fourier transform - FFT-based, zero-padded, windowed.

```
def F(x, dt=1.0, n=0, window=ones):
    nx = len(x)
    x = window(nx) * x
    nx = max(n, nx)
    nx = int(2**ceil(log2(nx)))
    fft_x = fft(x, nx)
    def Fx(f):
        k = (round_((ravel(f) * nx * dt)) % nx).astype(uint64)
        return dt * fft_x[k]
    return Fx
```

4.3 Multirate Signal Processing

Signal processing systems are **multirate** when they manage signals with different sample times. Filters, introduced in the previous sections, do not alter the sample time of the signals they are applied to, but **decimators** and **expanders** do; they are the new building blocks that allows us to **downsample** – decrease of the data rate – and **upsample** – increase in the data rate, while controlling the impact of these operations on the signal spectral content. A downsampling of a factor 5 for example may be used to get a 44.1 kHz signal down to a 8.82 kHz rate – which is still satisfactory for voice signals – and upsampling can be used to go back to the original data rate.

4.3.1 Decimation and Expansion

Decimation.

The **decimation** of a factor M of a discrete signal x with a sampling time of Δt is the signal with a sampling time of $M\Delta t$ denoted $x \downarrow M$ and defined by:

$$(x \downarrow M)(t) = x(t), t \in \mathbb{Z}M\Delta t. \quad (4.25)$$

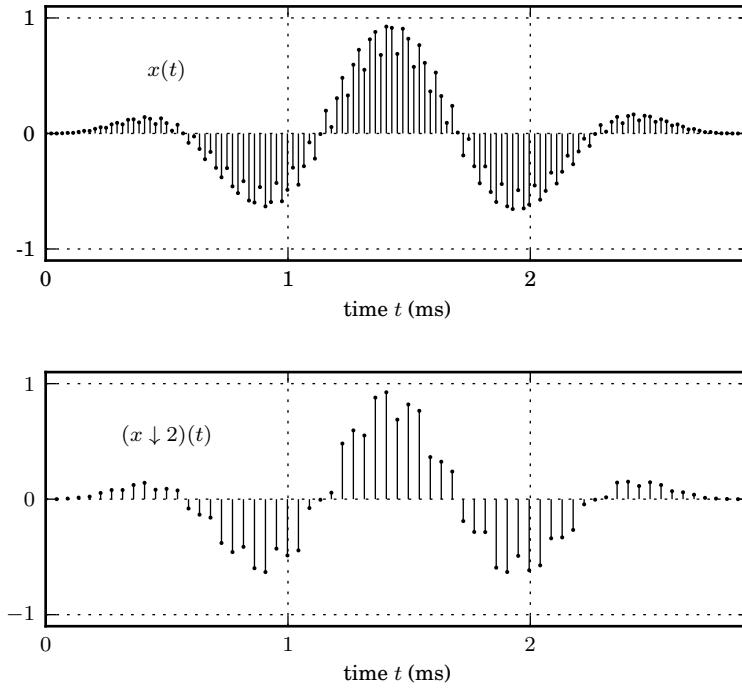


Figure 4.5: **Decimation, Temporal View:** a 44.1 kHz sampled signal $x(t)$ and its decimated version (by a factor of 2).

If x is a finite causal signal represented by the NumPy array x , the implementation of decimation of a factor M is straightforward with the slicing mechanism of arrays:

```
def decimate(x, M=2):
    return x[::M].copy()
```

Note that the length of `decimate(x,M)` is $\text{len}(x)/M$ – the quotient of the integer $\text{len}(x)$ by the integer M – if $\text{len}(x)$ is a multiple of M but $\text{len}(x)/M + 1$ otherwise. The copy may be necessary in some use cases and is therefore included to be safe. Indeed, the slicing operation has a pass-by-reference semantics in NumPy: $x[::M]$ is not a copy of the content of x but merely a view into it, therefore a change in the values of x would also change the sliced data⁶.

Decimation has a arguably strange – but well-defined – effect on the spectrum of a signal. Consider for example the decimation of factor 2 on the signal x with a

⁶see http://www.scipy.org/NumPy_for_Matlab_Users.

sampling time of Δt :

$$\begin{aligned}(x \downarrow 2)(f) &= 2\Delta t \sum_{t \in \mathbb{Z}2\Delta t} (x \downarrow 2)(t) \exp(-2i\pi ft) \\ &= 2\Delta t \sum_{t \in \mathbb{Z}2\Delta t} x(t) \exp(-2i\pi ft) \\ &= \Delta t \sum_{t \in \mathbb{Z}\Delta t} x(t) \exp(-2i\pi ft) + \Delta t \sum_{t \in \mathbb{Z}\Delta t} (-1)^{t/\Delta t} x(t) \exp(-2i\pi ft)\end{aligned}$$

As we have $(-1)^{t/\Delta t} = \exp(-i\pi)^{t/\Delta t} = \exp(-2i\pi t/2\Delta t)$ we end up with $(x \downarrow 2)(f) = \Delta t \sum_{t \in \mathbb{Z}\Delta t} x(t) \exp(-2i\pi ft) + \Delta t \sum_{t \in \mathbb{Z}\Delta t} x(t) \exp(-2i\pi(f + 1/2\Delta t)t)$ and therefore

$$(x \downarrow 2)(f) = x(f) + x(f + \Delta f/2) \quad (4.26)$$

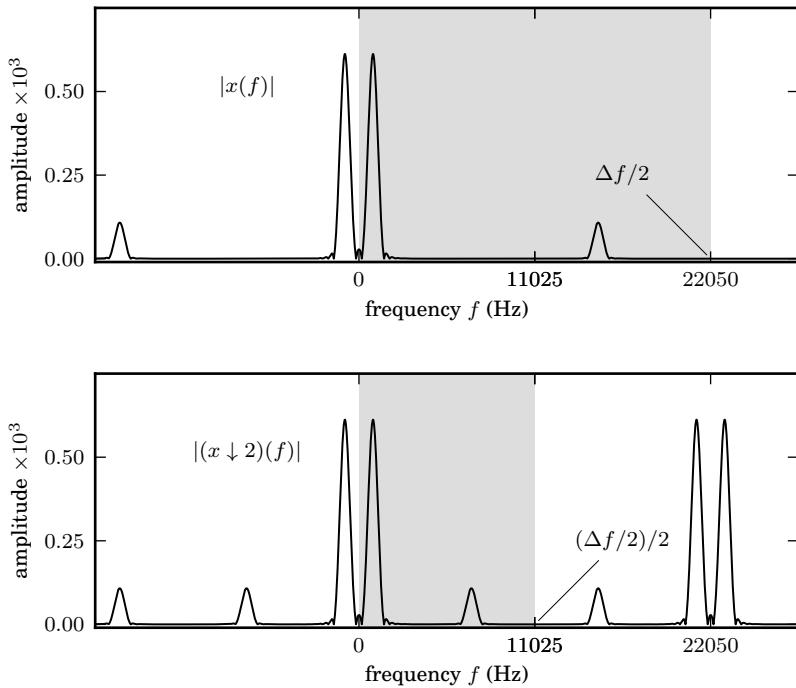


Figure 4.6: **Decimation and Spectrum:** spectra of a 44.1 kHz sampled signal, before and after a decimation of factor 2 (temporal view: fig. 4.3.1). Doubling the sample rate Δt effectively halves Δf and therefore halves the frequency range where the spectral content is significant.

For a decimation of factor M , we obtain by similar computations

$$(x \downarrow M)(f) = \sum_{k=0}^{M-1} x(f + k\Delta f/M)$$

(4.27)

What this formula means is that after decimation of a factor M , the spectral content of the signal at frequency $f \in [0, \Delta f/M)$ is a mix of the spectral content of the original signal at the frequencies

$$f, f + \Delta f/M, f + 2\Delta f/M, \dots, nf + (M - 1)\Delta f/M.$$

This phenomenon is called **(spectral) folding**. As every frequency in the original signals has generated copies of itself at new frequencies, the phenomenon is also called **(spectral) aliasing**.

Expansion.

The **expansion** of factor M applies to a discrete signal x with a time step Δt and creates a signal with a time step $\Delta t/M$ denoted $x \uparrow M$ and defined by:

$$(x \uparrow M)(t) = \begin{cases} x(t) & \text{if } t \in \mathbb{Z}\Delta t \\ 0 & \text{otherwise.} \end{cases} \quad (4.28)$$

Again, the implementation in NumPy for finite causal signals is straightforward:

```
def expand(x, M=2):
    output = zeros(M * len(x))
    output[::M] = x
    return output
```

The length of `expand(x, M)` is $M * \text{len}(x)$. It could be reduced to $(M - 1) * \text{len}(x) + 1$ without any information loss as the last $M - 1$ values of `output` are zeros, but it is often convenient to obtain a signal whose length is a multiple of the expansion factor. This operation does not alter the shape of the spectral content of the signal:

$$\begin{aligned} (x \uparrow M)(f) &= (\Delta t/M) \sum_{t \in \mathbb{Z}\Delta t/M} (x \uparrow M)(t) \exp(-2i\pi tf) \\ &= (\Delta t/M) \sum_{t \in \mathbb{Z}\Delta t} x(t) \exp(-2i\pi tf) \end{aligned}$$

and therefore

$$(x \uparrow M)(f) = \frac{1}{M} x(f) \quad (4.29)$$

4.3.2 Downsampling and Upsampling

Decimation is the basic operation to reduce the data rate of a signal and therefore compress it. However, this operation creates aliases in the spectral content of the signal where high and low-frequency are mixed and cannot be separated one from the other anymore. We can however decide to get rid in a controlled manner of some spectral content of the signal to keep the rest intact.

Note that if the spectral content of a signal before decimation of factor M is entirely into the $(-\Delta f/2M, \Delta f/2M)$ band, aliasing does not happen as we have

$$(x \downarrow M)(f) = x(f) \text{ if } f \in (-\Delta f/2M, \Delta f/2M)$$

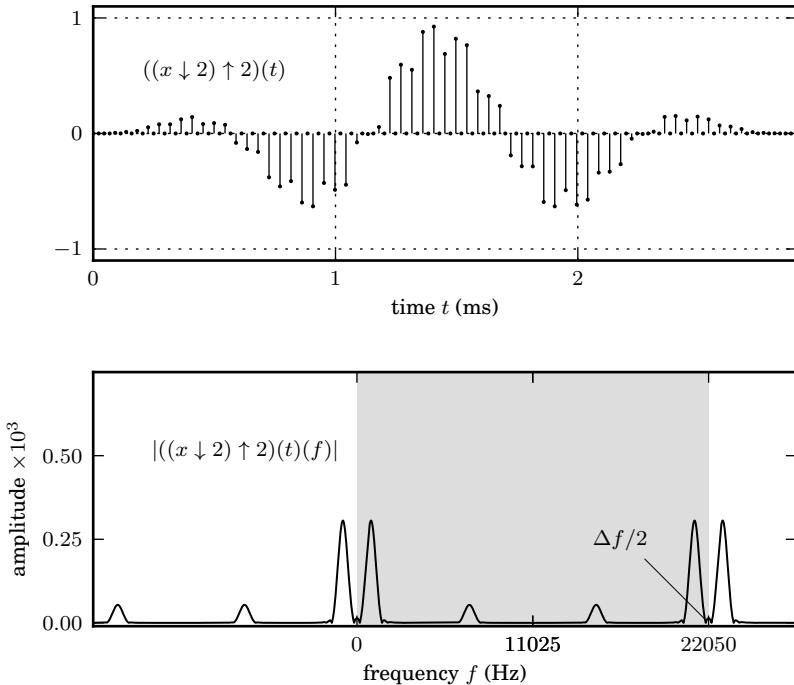


Figure 4.7: **Expansion** : temporal and spectral views of the signal x of figure 4.3.1 after downsampling then upsampling, both of a factor 2.

This can be achieved if we filter the original signal with a perfect low-pass filter of cutoff frequency $f_c = \Delta f/2M$. We then lose signal information in all frequency bands but the one of lowest frequency, but at least, this one is perfectly preserved by decimation. We call this combination of low-pass filtering and decimation **downsampling**.

Reconstruction of the (low-frequency content of) original signal is then just a matter of getting back the the original rate, by expansion and apply a gain of M to the result. That leads to exactly the right spectrum in the band $(-\Delta f/2M, \Delta f/2M)$ but not in the rest of $(-\Delta f/2, \Delta f/2)$ as the spectrum is $\Delta f/2M$ -periodic. To get rid of the high frequency content, we simply apply the perfect low-pass filter with cutoff frequency $f_c = \Delta f/2M$. The combination of (zero-)expansion, gain and filtering is called **upsampling**.

Let's summarize this: a downsampling of order M allows to reduce the data rate by a factor of M and keeps information one M -th of the spectral range – the lowest frequency part. Upsampling may be used to reconstruct a signal at the original rate whose content is the low-frequency content of the original one and has no higher spectral components.

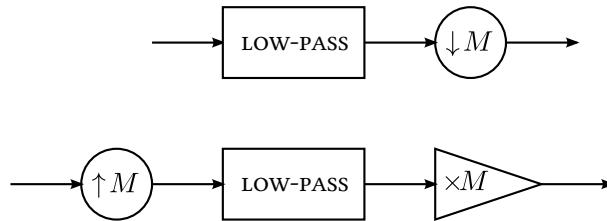


Figure 4.8: downsampling and upsampling diagrams

4.3.3 Ideal Filter Banks and Perfect Reconstruction

In the previous section we have explained how we could divide the signal rate by a factor of M by keeping only one M -th of its spectral content and throwing away everything else. We now consider the steps leading to a more flexible approach: we split the data signal into M frequency bands and we will later design methods to allocate bits to such or such a band depending on the spectral content of the signal.

In order to split the signal into M uniformly spaced spectral bands, we introduce an **analysis filter bank**: a set of M filters a^i , $i = 0, \dots, M - 1$ with $a_n^i = a^i(n\Delta t)$, $n \in \mathbb{Z}$ that we all apply to the original signal. All the filters are band-pass, with low frequency $i\Delta f/M$ and high-frequency limit $(i + 1)\Delta f/M$. We then decimate the signal on all branches, so that the original data rate can be kept. We know what is the spectral content of the signal after decimation on the branch $i = 0$, but what is going on with the other branches? Let x be the original signal and x^i is the signal filtered by the i -th filter. The content of x^i is entirely in the i -th frequency band, that is $(i\Delta f/M, (i + 1)\Delta f/M)$ (and the corresponding negative frequency band), so after decimation, the spectral content is

$$\sum_{k=0}^{M-1} x(f - k\Delta f/M) = x^i(f + i\Delta f/M) \text{ if } f \in (-\Delta f/2M, \Delta f/2M)$$

So again, in each branch, decimation has kept the relevant information. Given those M spectral components, are we able to reconstruct the original signal? In order to get the contribution from the i -th band in the right place, we can first expand the signal and multiply by M : that shifts the subband content to build a $\Delta f/M$ -periodic spectral content. To get this content only in the i -th band, we apply a perfect pass-band that corresponds to the i -th subband. Then we sum all these contributions.

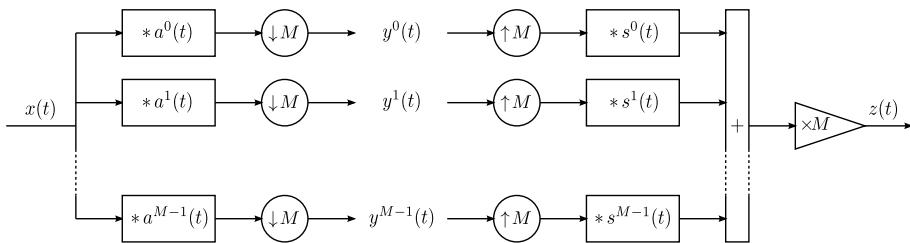


Figure 4.9: analysis and synthesis filter banks diagrams

4.3.4 Filter Banks and Perfect Reconstruction

One issue with the previous scheme is that perfect band-pass filters cannot be implemented. We'd like as to replace them by some finite impulse response filters approximations, study if perfect reconstruction is still possible and if it can't be achieved, measure the error we are introducing.

Consider the diagram in figure (4.9) where the a^i and s^i are the impulse response of the analysis and synthesis filter banks. The formulas (4.27) and (4.29) yield the following expression for the output z of the analysis + synthesis process from the input signal x :

$$z(f) = \sum_{k=0}^{M-1} \left[\sum_{i=0}^{M-1} s_i(f) a_i(f + k\Delta f/M) \right] x(f + k\Delta f/M)$$

which means that the diagram will achieve **perfect reconstruction** if we have

$$\sum_{i=0}^{M-1} s_i(f) a_i(f + k\Delta f/M) = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{if } k = 1, \dots, M-1. \end{cases}, \quad (4.30)$$

or in other words, if all the **distortion functions** $D_k(f)$, $k = 0, \dots, M-1$, defined by

$$D_0(f) = \sum_{i=0}^{M-1} s_i(f) a_i(f) - 1,$$

$$D_k(f) = \sum_{i=0}^{M-1} s_i(f) a_i(f + k\Delta f/M), \quad k = 1, \dots, M-1$$

(4.31)

are identically zero.

4.3.5 Cosine Modulated Filter Banks

We build in this section a family of pass-band filters with impulse responses $a^i(t)$, $i = 0, \dots, M-1$, whose pass-band is $(i\Delta f/M, (i+1)\Delta f/M)$, and based on a single prototype filter. The prototype is selected as a low-pass filter with cutoff frequency $f_c = \Delta f/4M$; the perfect prototype filter impulse response is (see (4.20)):

$$h(n\Delta t) = \frac{\Delta f}{2M} \operatorname{sinc} \frac{\Delta f}{2M} n\Delta t = \frac{\Delta f}{2M} \operatorname{sinc} \frac{n}{2M}$$

To generate the i -th pass-band filter, all we have to do it to shift the spectrum by $(i + 0.5)\Delta f/2M$ to the right, that is, multiply $h(n\Delta t)$ by

$$\exp(i2\pi(i + 0.5)(\Delta f/2M)(n\Delta t)) = \exp(i\pi(i + 0.5)n/M).$$

But then the filter impulse response would no longer be real, so we also perform the opposite frequency shift: we multiply $h(n\Delta t)$ by $\exp(-i\pi(i + 0.5)n/M)$ and add up both contributions; we end up with

$$a^i(n\Delta t) = 2h(n\Delta t) \times \cos(\pi(i + 0.5)n/M)$$

(4.32)

that is, a **cosine modulated filter bank**. The figure (4.10) displays the filters frequency responses where the prototype filter has been approximated by a FIR. If we selecting

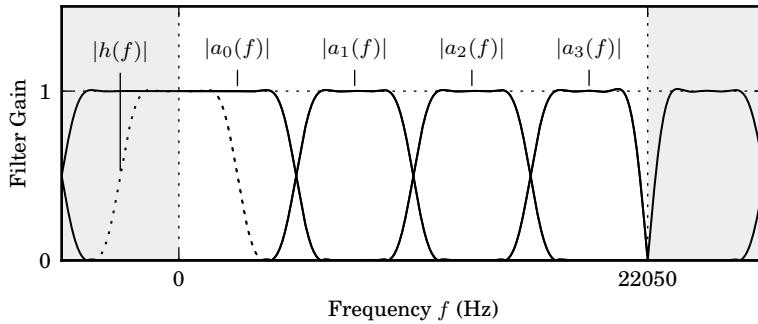


Figure 4.10: **Cosine Modulated Filter Banks:** gain of the components of a cosine modulated filter bank as a function of the frequency f . The sampling time Δt is 44.1 kHz, there are $M = 4$ filters and the prototype filter h – whose spectrum gain is dotted – has been truncated by the application of a Hanning window if length $N = 64$. The phase of such filter is not displayed as it is flat because the filters impulses responses are symmetrical – or linear as a function of f in a causal implementation.

as synthesis filters the same pass-band filters used for the analysis – $s^i(t) = a^i(t)$ – we maye compute the distortions induced by the analysis-synthesis process ; the results, displayed in figure (4.11), clearly points out that the basic approach we have adopted so far does not provide a good approximation to a perfect reconstruction. **Pseudo-QMF** (for **pseudo - quadrature mirror filters**) may be introduced to obtain sufficiently small distortion functions ; they are successfully used in layer I and II of MPEG-Audio for example. Their design relied on two modifications with respect to our approach so far. First, we introduce phase factors ϕ_i in the definition of a^i and s^i

$$\begin{aligned} a^i(n\Delta t) &= 2h(n\Delta t) \times \cos(\pi(i + 0.5)n/M + \phi_i) \\ s^i(n\Delta t) &= 2h(n\Delta t) \times \cos(\pi(i + 0.5)n/M - \phi_i) \end{aligned} \quad (4.33)$$

in order to cancel significant aliasing terms and ensure a relatively flat overall magnitude distortion (see [NE94, MB03]). Among several options, we select

$$\phi_i = \frac{\pi}{2} \left(\frac{N-1}{M} - 1 \right) (i + 0.5) \quad (4.34)$$

where N is the filter length and M the number of sub-bands.

Then the selection of the prototype filter does not rely on the expression of the perfect low-pass filter but is optimized to reduce distortion. The MPEG-Audio standard selection for this filter is displayed in figure 4.12.

4.3.6 Polyphase Representation of Filters Banks

Polyphase representation is an alternate description of filter banks that is suited to a real-time implementation. Unlike convolution-based implementation that require the full input values to be available to produce output values, polyphase representation of analysis and synthesis filter banks are amenable to matrix implementation that work frame by frame: they consume chunks of M samples to produce the same amount of output values.

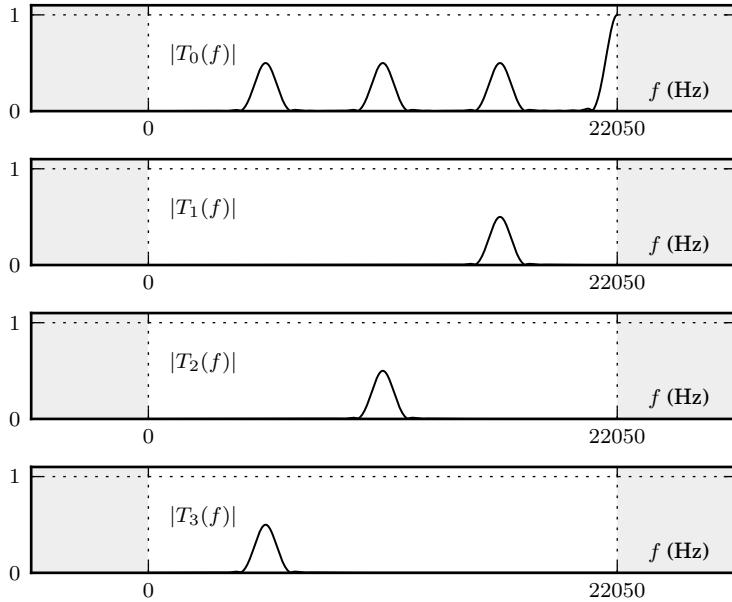


Figure 4.11: cosine modulated filter banks: distortions

Analysis Filter Bank

Consider the analysis filter bank depicted on the left of figure 4.9. Gather the output $y^i(t)$ of the i -th subbands into the vector signal $y(t) \in \mathbb{R}^M$, $t \in \mathbb{Z}M\Delta t$. This output vector is related to the input $x(t) \in \mathbb{R}$, $t \in \mathbb{Z}\Delta t$ by the formula:

$$y^i(t) = \Delta t \sum_{t' \in \mathbb{Z}\Delta t} a^i(t') x(t - t'), \quad t \in \mathbb{Z}M\Delta t \quad (4.35)$$

Let $y_n^i = y^i(nM\Delta t)$, $x_n^i = x^i(n\Delta t)$, $a_n^i = a^i(n\Delta t)$. This relationship takes the form $y_j^i = \Delta t \sum_{n=0}^{N-1} a_n^i x_{Mj-n}$, which can be considered as a simple matrix multiplication:

$$y_n = \mathcal{A} \begin{bmatrix} x_{Mn} \\ x_{Mn-1} \\ \vdots \\ x_{Mn-N+1} \end{bmatrix} \quad \text{with } \mathcal{A} \in \mathbb{R}^{M \times N}, \quad \mathcal{A}_{ij} = \Delta t \cdot a_j^i. \quad (4.36)$$

Alternatively, this form may be turned into an alternate block-diagram displayed in figure 4.13 that is the **polyphase** representation of the analysis filter bank : assume that N is a multiple of M and that we intend to apply the analysis filter bank to a finite causal signal x represented by the NumPy array x . We notice that the vector in right-hand side of the equation (4.36) acts as a buffer: every new value of n shifts the oldest values of x towards the bottom of the vector by M slots – effectively forgetting M of the oldest values – and introduces M new values of x at the top, so the signal x is used in frames of M samples. We also notice that y_0 does not depend of a whole x frame, only of x_0 . To simplify this matter, we assume that $x_0 = 0$ and won't compute y_0 . Effectively, we implement a process that with respect to the theoretical one delays

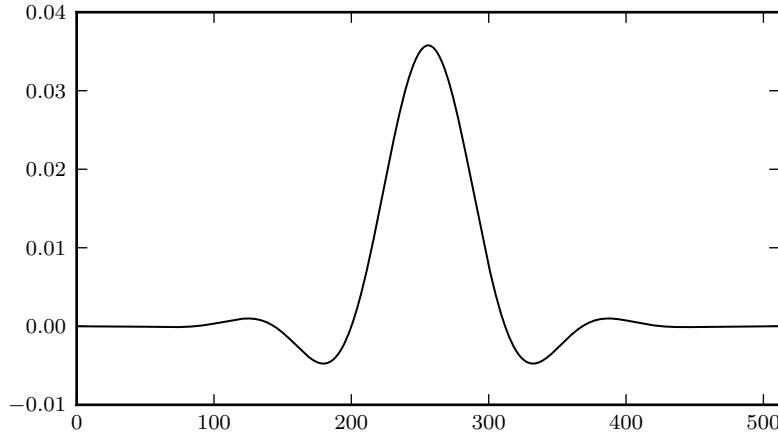


Figure 4.12: MPEG-Audio Layer I and II prototype 513-tap prototype filter

the input by one step – so that x_1 is the first non-zero value, not x_0 – and advances the output by one step – the first output we'll effectively compute is truly y_1 , not y_0 .

These computations may be carried by an instance of the `Analysis` class:

```
class Analysis(object):
    def __init__(self, a, dt=1.0):
        self.M, self.N = shape(a)
        self.A = a * dt
        self.buffer = zeros(self.N)
    def __call__(self, frame):
        frame = array(frame, copy=False)
        assert shape(frame) == (self.M,)
        self.buffer[self.M:] = self.buffer[:-self.M]
        self.buffer[:self.M] = frame[::-1]
        return dot(self.A, self.buffer)
```

The argument `a` in the `Analysis` constructor is meant to be a the 2-dim. array such that `a[i,:]` represent the i -th analysis filter impulse response. In order to use the instance `analysis = Analysis(a, dt)`, the array `x` has to be split in frames of length `M`.

In the implementation of the analysis filter banks for the MPEG PQMF, the pass-band filters are implemented as causal filters, introducing an extra delay of `MPEG.N / 2 = 256` samples. Given the the implementation delays already considered, the total delay induced by the implementation with respect to the original filter banks is `MPEG.N / 2 + 1 - MPEG.M`.

To make sure that the analysis filter banks has produced all its non zero-values, we feed the system extra zero frames. If the input data is available from the start in the array `x`, the corresponding output `y` may therefore be obtained as:

```
from filters import MPEG
from frames import split

x = r_[x, zeros(MPEG.N)]
frames = split(x, MPEG.M, zero_pad=True)
y = []
```

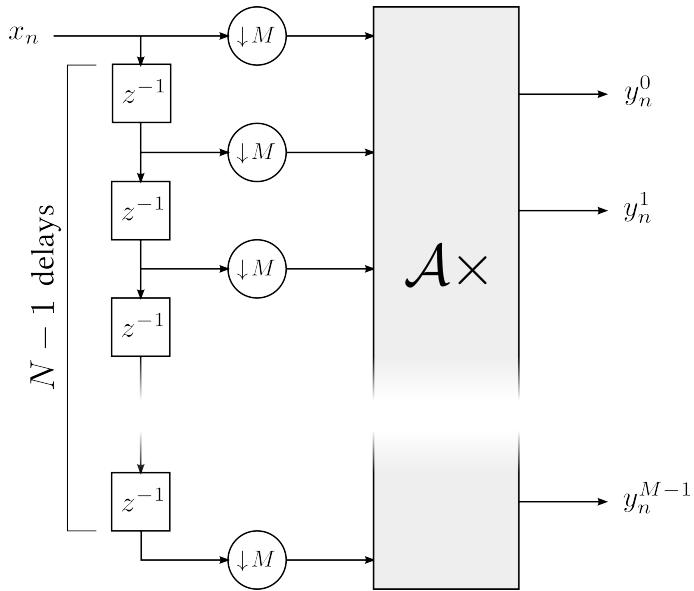


Figure 4.13: analysis filter bank: polyphase representation

```

for frame in frames:
    y.extend(analysis(frame))
y = array(y)

```

Synthesis Filter Bank

Consider the synthesis filter bank depicted on the right of the diagram 4.9. The output vector $z(t) \in \mathbb{R}$, $t \in \mathbb{Z}\Delta t$, is related to the input $y(t) \in \mathbb{R}^M$, $t \in \mathbb{Z}M\Delta t$, by the formula:

$$z(t) = M \sum_{i=0}^{M-1} \Delta t \sum_{t' \in \mathbb{Z}\Delta t} s^i(t')(y^i \uparrow M)(t - t') \quad (4.37)$$

or – using integer indices – by $z_n = M \sum_{i=0}^{M-1} \Delta t \sum_{j=0}^{N-1} s_j^i (y^i \uparrow M)_{n-j}$. With

$$\mathcal{S} = [M\Delta t s_j^i]_{i,j} \quad (4.38)$$

we may turn this equation into

$$z_n = \sum_{j=0}^{N-1} [\mathcal{S}^t (y \uparrow M)_{n-j}]_j. \quad (4.39)$$

Now consider the polyphase synthesis diagram 4.14, dual of the analysis diagram 4.13, where \mathcal{P} is an unknown $N \times M$ matrix. Its output is related to its input by

$$z_n = \sum_{j=0}^{N-1} [\mathcal{P}(y \uparrow M)_{n-j}]_{N-1-j}.$$

So if we set $\mathcal{P} = JS^t$, where J is the $M \times M$ matrix such as $J_{i,j} = 1$ if $i - j = M - 1$ and 0 otherwise, the diagram outputs the same thing as (4.39), only delayed by $N - 1$ samples.

Let $w_n = \mathcal{S}^t y_n$. A careful examination of the polyphase representation of the synthesis filter banks show that the computation may be performed in frames of M values. Indeed, the output z_n is given by $z_0 = w_0^{N-1}, \dots, z_{M-1} = w_0^{N-M}$, then $z_M = w_0^{N-M-1} + w_1^{N-1}, z_{M+1} = w_0^{N-M-2} + w_1^{N-2}$, etc. Here is a possible imple-

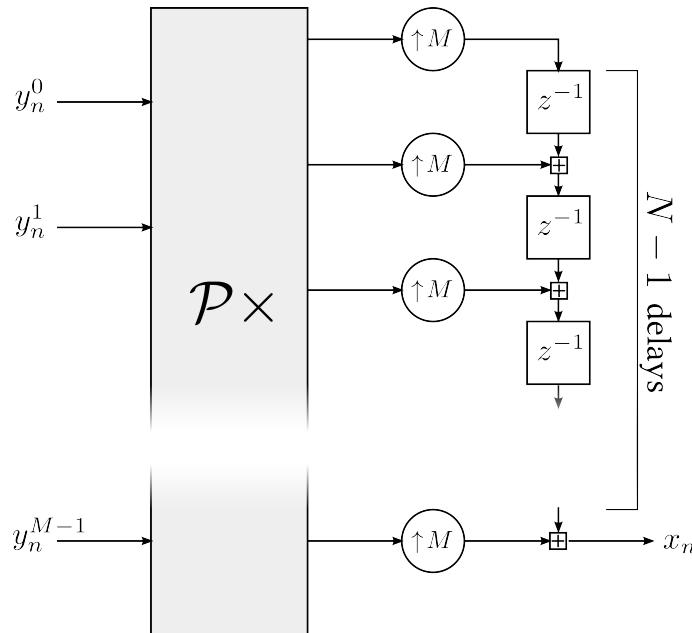


Figure 4.14: synthesis filter bank: polyphase representation

mentation:

```
class Synthesis(object):
    def __init__(self, s, dt=1.0):
        self.M, self.N = shape(s)
        self.P = transpose(self.M * dt * s)[::-1,:]
        self.buffer = zeros(self.N)
    def __call__(self, frame):
        frame = array(frame, copy=False)
        assert shape(frame) == (self.M,)
        self.buffer += dot(self.P, frame)
        output = self.buffer[-self.M:][:-1].copy()
        self.buffer[self.M:] = self.buffer[:-self.M]
        self.buffer[:self.M] = zeros(self.M)
        return output
```

Again, the input vector y can be extended with as many zeros as necessary to get all non-zero output values extracted from the buffer.

4.4 Psychoacoustics - Perceptual Models

4.4.1 Acoustics - Physical Values

Audio signal values represent a variation of the sound pressure with respect to the atmospheric pressure. The standard unit used to measure such pressure is the pascal (Pa), a unit that corresponds to N/m^2 ; while atmospheric pressure is around 100 Pa – the standard atmosphere (atm), an alternative unit, is equivalent to 101.325 Pa – variations of the pressure in the audio context typically range from 10^{-5} Pa (absolute threshold of hearing) to 10^2 Pa (threshold of pain).

In order to measure the **sound pressure level (SPL)** denoted L , we focus on the difference $p(t)$ between the actual pressure and the atmospheric pressure and compute its quadratic mean:

$$P^2 = \langle p^2 \rangle$$

where $\langle \cdot \rangle$ denotes, depending on the context, either a temporal mean or a probabilistic one. We then normalize this value with respect to $P_0 = 20 \mu\text{Pa}$ and measure the ratio in a logarithmic scale

$$L = 20 \log_{10} \frac{P}{P_0} \quad (4.40)$$

The sound pressure level unit is the **decibel (dB)**. When the sound is a plane travelling wave, the normalized **sound intensity** I is related to p by

$$\frac{I}{I_0} = \frac{P^2}{P_0^2} \quad \text{with } I_0 = 10^{-12} \text{ N/m}^2$$

so that

$$L = 10 \log_{10} \frac{I}{I_0} \quad (4.41)$$

Now, sound pressure level may be computed according to the spectral content of the signal: if $p(f)$ denotes the spectrum of the signal $p(t)$, a finite causal signal of length N with $T = N\Delta t$, we have by Parseval's formula (cf. formula (4.8))

$$P = \frac{1}{T} \Delta t \sum_{n=0}^{N-1} p(t)^2 = \frac{1}{T} \int_{-\Delta f/2}^{\Delta f/2} |p(f)|^2 df$$

as we end up with

$$L = 10 \log_{10} \frac{2}{T} \int_0^{\Delta f/2} \frac{|p(f)|^2}{P_0^2} df$$

This result is usually presented in terms of **sound (intensity) density** also commonly called **sound power density**, a value measured in dB that we denote $\ell(f)$ and define as

$$\ell(f) = 10 \log_{10} \frac{2}{T} \frac{|p(f)|^2}{P_0^2} \quad (4.42)$$

so that

$$L = 10 \log_{10} \int_0^{\Delta f/2} 10^{\ell(f)/10} df. \quad (4.43)$$

For example, if a sound has a constant power density ℓ in a frequency range of width ΔF and no power outside this range, its sound pressure level is

$$L = \ell + 10 \log_{10} \Delta F$$

Digital audio signal being scaled to fit the range of their quantizer, we need to normalize somehow the signal before getting into the SPL computation. For 16-bit linearly quantized signals normalization takes place in the following way: first, scale to fit into $[-1.0, 1.0]$, then a quadratic mean of N signal values

$$X^2 = \langle x^2 \rangle = \frac{1}{T} \Delta t \sum_{n=0}^{(N-1)} |x(n\Delta t)|^2$$

is mapped to SPL with the formula

$$L = 10 \log_{10} \langle x^2 \rangle + 96 \text{ dB} \quad (4.44)$$

or equivalently

$$P = 10^{4.8} P_0 \times X \quad (4.45)$$

or even

$$\ell(f) = 10 \log_{10} \left(\frac{2}{T} |x(f)|^2 \right) + 96 \quad (4.46)$$

and

$$L = 10 \log_{10} \left(\frac{2}{T} 10^{9.6} \int_0^{\Delta f / 2} |x(f)|^2 df \right) \quad (4.47)$$

4.4.2 Threshold in Quiet

To understand why the normalization in the previous section actually makes sense, we need to know more about the human hearing range. The **absolute threshold of hearing (ATH)** or **threshold in quiet** is a function of the frequency f such that a pure tone with a frequency f will be noticed if and only if its SPL is above the ATH at this frequency. An approximate analytical model for the threshold of hearing – as a function of the frequency f in kHz – is:

$$T_a(f) = 3.64f^{-0.8} - 6.5 \exp(-0.6(f - 3.3)^2) + 10^{-3}f^4 \quad (4.48)$$

Now consider the values of a uniform 16-bit quantizer on $[-1.0, 1.0]$. Given that the maximum possible value of $\langle x^2 \rangle$ is 1.0, the maximum value of the right-hand side of the equation (4.44) is 96 dB. For a pure tone, before normalization, the maximum value of $\langle x^2 \rangle$ is $1/\sqrt{2}$, and therefore the maximum sound pressure level is ≈ 93 dB.

Now, even if there is no minimum value *per se*, a good reference is the quantization noise energy. For a 16-bit linear quantization on $(-1.0, 1.0)$, the step size Δ is uniform with a value of $2.0/2^{16} = 2^{-15}$. In the context of the high resolution hypothesis, the equation (2.27) provides the estimate $\mathbb{E}[b^2] = \Delta^2/12$ for the noise power and therefore the normalized noise pressure level is

$$10 \log_{10} (2^{-30}/12) + 96 \approx -5.1 \text{ dB}$$

So with the normalization of the previous section, the practical range in terms of sound pressure level of the 16-bit linear quantizer is $[-5.1, 93]$ dB, that is approximately a 100 dB range. Compared to the reference curve for the absolute threshold of hearing (see fig. 4.16), we notice that this region covers essentially all of the frequency range from 20 Hz to 20 kHz and that the low bound closely matches the minimal value of the ATH. So this scaling by 96 dB would correspond to a kind of optimal amplification configuration of the loudspeakers, one that would allow to get into large audible value of the SPL without saturation of the signal and also to allow proper perception of the lowest sounds that the ear can actually detect.

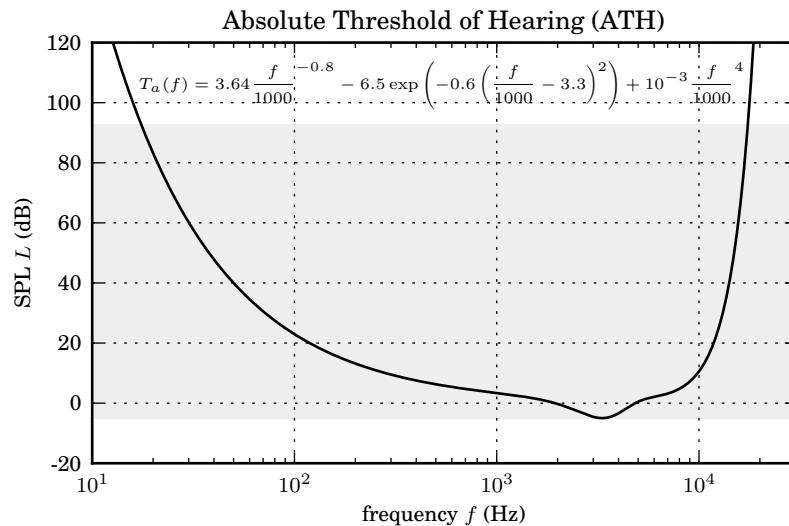


Figure 4.15: Absolute Threshold of Hearing. The grey region denotes the SPL range covered by a 16-bit linear quantization.

4.4.3 Simultaneous Masking

Beyond the model of the absolute threshold of hearing, the main characteristic of the psychoacoustic system that perceptual model used in technologies such as MP3, Ogg/Vorbis, AAC, etc. rely on is **simultaneous masking**. Basically, a loud sound whose energy is located in a given narrow frequency range is going to make every other signal located in the same frequency neighbourhood harder to detect.

A simple first computational model for this type of masking relies on Fletcher's **critical band** concept. Fletcher considers the possible masking of a pure tone with frequency f by a signal whose energy is located in the $[f - \Delta F/2, f + \Delta F/2]$ range and makes the assumption that there exist a **critical bandwidth** ΔF_c – or a **critical band** $[f - \Delta F_c/2, f + \Delta F_c/2]$ – such that:

1. the distribution of the intensity of the masker within the critical band does not influence the outcome of the masking experiment, only the total SPL for the critical band matters,

2. no amount of intensity outside of the critical band may change the outcome of the experiment,
3. masking occurs when the intensity of the masker in the critical band exceeds the intensity of the test tone.

In a few words, the critical band is the largest region around the test tone where the power density of masker signals consistently increases the masking effect. This set of assumption has a number of shortcomings: the distribution of intensity nearby the test tone *does* matter, but not so much as long as the distribution is quite uniform (say a band-limited noise or a combination of 5 pure tones uniformly gathered won't make much of a difference), the influence of the distribution of energy does not have a drop from 100% to 0% at a limit but is smoother and finally, the masker needs from 2 to 4 more intensity than the test tone to properly mask it.

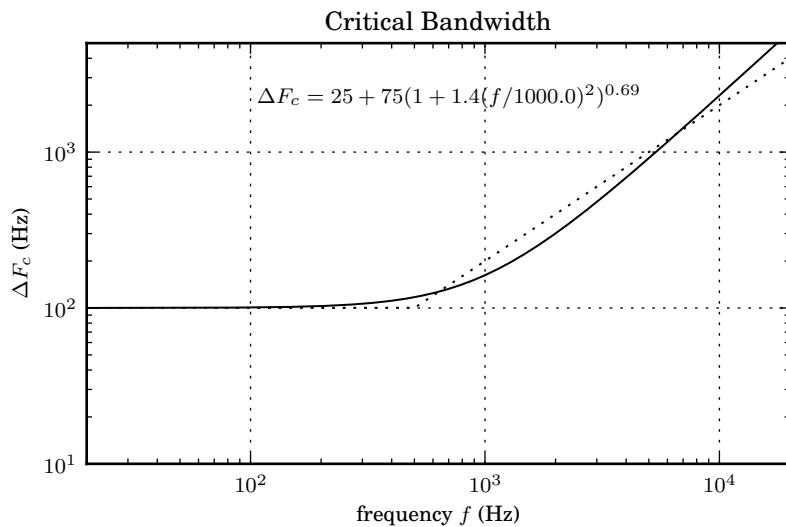


Figure 4.16: Critical Bandwidth. The dashed lines represent the simpler piecewise linear estimate.

Despite all these shortcomings, Fletcher's model of critical bands is important and estimates of the critical bandwidth as a function of the frequency center frequency f may be derived from simple experimental protocols. An approximate analytical formula for the critical bandwidth in Hz – as a function of the center frequency f in Hz is:

$$\boxed{\Delta F_c = 25 + 75(1 + 1.4(f/1000.0)^2)^{0.69}} \quad (4.49)$$

it also accepts the following piecewise linear approximation:

$$\Delta F_c = 100 \text{ Hz if } f \leq 500 \text{ Hz and } \Delta F_c = 0.2 \times f \text{ beyond.} \quad (4.50)$$

The critical band concept is used in many model beyond masking ; for convenience, a unit is introduced to measure frequencies in the critical band rate scale : it is named the **Bark**. By convention, $f = 0$ Bark corresponds to $f = 0$ Hz. Then the

right end of the critical band that starts at 0 Bark corresponds to 1 Bark, the right end of the critical band that starts at 1 Bark corresponds to 2 Bark, and so on and so forth. A convenient analytical approximation of the Hz to Bark conversion is given by:

$$f [\text{Bark}] = 13.0 \arctan(0.76f/1000.0) + 3.5 \arctan(f/1000.0/7.5)^2 \quad (4.51)$$

4.4.4 Spreading Functions

Let's consider for a moment the masks yielded by Fletcher's set of assumption, that is the audibility threshold of pure tones as a function of their frequency f . We may consider as maskers either pure tones or band-limited white noises. The graphs in figure 4.4.4 is an example of the masks levels that can be derived from Fletcher's model of masking. If we assume that elementary maskers combine into a global one by

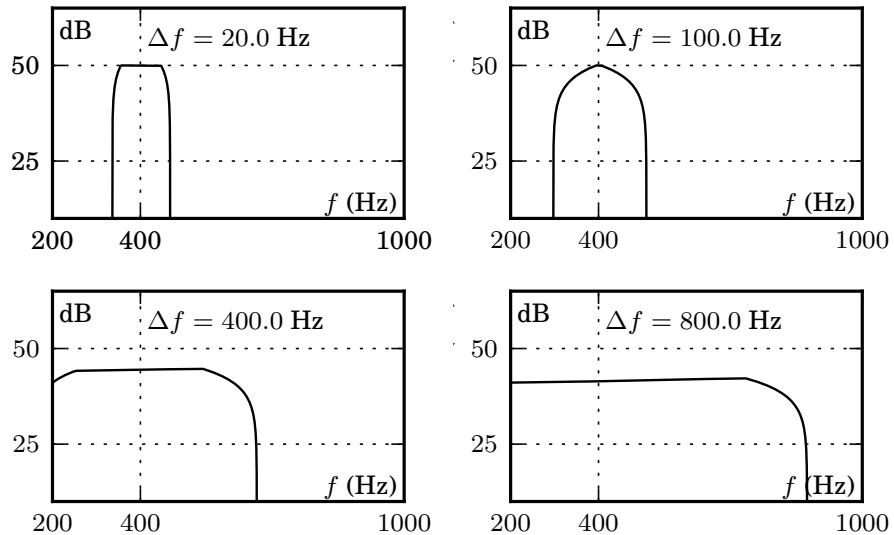


Figure 4.17: Fletcher's Model – Band-Limited Noise Masking Curves. The four maskers share a common SPL of $L = 50$ dB and center frequency of $f_c = 400$ Hz while their bandwidth increases from $\Delta f = 20.0$ Hz to 800 Hz.

addition of intensity – and take into account the absolute threshold of hearing as yet another mask, we end up with the kind of masking curves displayed in figure 4.4.4.

4.4.5 Implementation - Bit Allocation Strategies

Consider a random signal X split into M subband signals X_k . Assume that in every subband an estimate $P_m(k)$ of the masking level intensity is available. Given a selection of quantizers $[\cdot]_k$, if we have

$$\forall k \in \{0, \dots, M-1\}, \mathbb{E}[(X_k - [X_k]_k)^2] \leq P_m(k) \quad (4.52)$$

then in every channel, the quantization noise is masked by the signal itself. These conditions (4.52) may be satisfied by a variable bitrate algorithm, but it is more likely

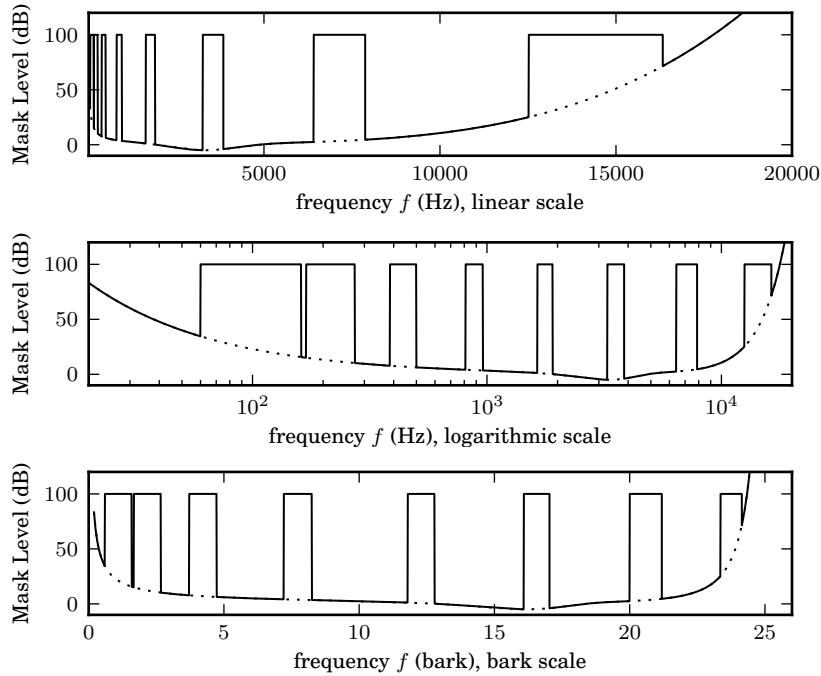


Figure 4.18: **Fletcher's Model – Pure Tones Masking Curves.** The masker is a combination of 8 pure tones of level 100 dB whose frequencies start with 110 Hz and double for each new tone. Frequencies are displayed in linear, logarithmic and bark scales.

that we have a total budget of bits to allocate per frame and that we are merely trying to spread the quantization noise above the masking levels among all subbands. We may achieve this by solving

$$\min \sum_{k=0}^{M-1} \frac{\mathbb{E}[(X_k - [X_k]_k)^2]}{P_m(k)} \quad (4.53)$$

or, under the high resolution assumption, if Δ_{f_k} denotes the quantizer step size of $[\cdot]_k$, as we have $\mathbb{E}[(X_k - [X_k]_k)^2] = (1/12)\mathbb{E}[\Delta_{f_k}(X_k)^2]$, by solving

$$\min \sum_{k=0}^{M-1} \frac{\mathbb{E}[\Delta_{f_k}(X_k)^2]}{P_m(k)} \quad (4.54)$$

Uniform Quantizers

Assume that every quantizer $[\cdot]_k$ is a uniform quantizer on $(-1, 1)$ with step size Δ_k . In every subband, the number of bits b_k is related to the step size by $\Delta_k = 2/2^{b_k}$. The availability of a constant number of bits per frame therefore leads to

$$\sum_{k=0}^{M-1} \log \Delta_k = \text{const.} \quad (4.55)$$

The constrained optimization problem (4.54) + (4.55) may be solved by lagrangian methods: we introduce $\Delta = (\Delta_0, \dots, \Delta_{M-1})$,

$$L(\lambda, \Delta) = \sum_{k=0}^{M-1} \frac{\Delta_k^2}{P_m(k)} + \lambda \sum_{k=0}^{M-1} \log \Delta_k$$

and solve the equation $\nabla_{\Delta} L(\lambda, \Delta) = 0$. As for any k , $\frac{\partial L}{\partial \Delta_k} = 2 \frac{\Delta_k}{P_m(k)} + \frac{\lambda}{\Delta_k}$, the optimal set of step size satisfies

$$\boxed{\Delta_k^2 \propto P_m(k)} \quad (4.56)$$

the proportionaly constant being adjusted to match the bit budget.

Optimal Quantizers.

Instead of using uniform quantizers in every subbands, we may attempt to minimize every quantizer signal-to-noise ratio for a yet unknow number of bits, then consider the optimal allocation of bts. We assume that the characteristic function f_k used to implement $[\cdot]_k$ maps the real numbers into $[-1.0, +1.0]$ and hence that

$$\int_{-\infty}^{+\infty} f'_k(x) dx = 2 \quad (4.57)$$

Note that if a uniform quantization applied on $[-1, 1]$ and has a budget of b_k bits, the corresponding constant step size is $\Delta_k = 2/2^{b_k}$. As we have $\Delta_{f_k}(x) = \Delta_k/f'_k(x)$, the equation (4.57) yields

$$\int_{-\infty}^{+\infty} \frac{1}{\Delta_{f_k}(x)} dx = \frac{2}{\Delta_k} \quad (4.58)$$

If the signal X_k has a density p_k , the step size Δ_{f_k} that is optimal with respect to the quantization signal-to-noise ratio satisfies $\Delta_{f_k}(x) \propto p_k^{-1/3}(x)$. Combined with (4.58), this equation yields

$$\Delta_{f_k}(x) = \frac{\Delta_k}{2} \left[\int_{-\infty}^{+\infty} p_k^{1/3}(y) dy \right] p_k^{-1/3}(x)$$

and therefore

$$\mathbb{E}[\Delta_{f_k}(X_k)^2] = \frac{\Delta_k^2}{4} \left[\int_{-\infty}^{+\infty} p_k^{1/3}(x) dx \right]^3$$

Now, for common probability distributions p_k such as normal distributions or Laplace distributions, we have

$$\int_{-\infty}^{+\infty} p_k^{1/3}(x) dx \propto \mathbb{E}[X_k^2]^{1/3}$$

and hence

$$\mathbb{E}[\Delta_{f_k}(X_k)^2] \propto \Delta_k^2 \times \mathbb{E}[X_k^2] \quad (4.59)$$

The new minimization problem is therefore

$$\min \sum_{k=0}^{M-1} \Delta_k^2 \frac{\mathbb{E}[X_k^2]}{P_m(k)} \quad (4.60)$$

under

$$\sum_{k=0}^{M-1} \log \Delta_k = \text{const.} \quad (4.61)$$

If we introduce the **signal-to-mask (SMR) ratio** in the band k , defined by

$$\boxed{\text{SMR}_k^2 = \frac{\mathbb{E}[X_k^2]}{P_m(k)}} \quad (4.62)$$

the solution to this optimization problem is given by

$$\boxed{\Delta_k \propto \text{SMR}_k^{-1}} \quad (4.63)$$

Bibliography

- [FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, pages 216–231, 2005.
- [GvV75] R. Gallager and D. van Voorhis. Optimal source codes for geometrically distributed integer alphabets (corresp.). *Information Theory, IEEE Transactions on*, 21(2):228 – 230, mar 1975.
- [Har78] F. J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. 1978.
- [HSW01] L. Hanzo, F.C.A. Somerville, and J.P. Woodard. *Voice Compression and Communications: Principles and Applications for Fixes and Wireless Channels*. IEEE Press-John Wiley & Sons, February 2001.
- [Kie04] Aaron B. Kiely. Selecting the golomb parameter in rice coding. *IPN Progress Report*, Vol. 42-159, nov 2004.
- [Mar72] J. Markel. The sift algorithm for fundamental frequency estimation. *Audio and Electroacoustics, IEEE Transactions on*, 20(5):367 – 377, dec 1972.
- [MB03] Richard E. Goldberg Marina Bosi. *Introduction to Digital Coding Audio Standards*. 2003.
- [MSW00] N. Merhav, G. Seroussi, and M.J. Weinberger. Optimal prefix codes for sources with two-sided geometric distributions. *Information Theory, IEEE Transactions on*, 46(1):121 –135, jan 2000.
- [NE94] Truong Q. Nguyen and Member Eee. Near-perfect-reconstruction pseudo-qmf. *IEEE Trans. Signal Processing*, 42:65–76, 1994.