# DIGITAL FILTERING AND COMPRESSION IN IMAGE PROCESSING AND VOLUME RENDERING

By

Jindřich Nový

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
BRNO UNIVERSITY OF TECHNOLOGY
TECHNICKÁ 2, 616 69 BRNO
MAY 2005

BRNO UNIVERSITY OF TECHNOLOGY

Date: **May 2005**

Author: **Jindřich Nový**

Title: **Digital Filtering and Compression in Image Processing and Volume Rendering**

Department: **Computer Graphics and Geometry**

Degree: **Ph.D.** Convocation: **June** Year: **2005**

Permission is herewith granted to Brno University of Technology to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

*To my parents Jindřich and Alica*

# Contents

# List of Tables

# List of Figures

# Contribution of the dissertation

The main contribution of the dissertation should be a complete fulfillment of the following tasks:

1. Development of new volume rendering software able to interactively analyze one-channel and multi-channel general large scale volume datasets.

2. Development of new lossless compression algorithm and software for compression of digital images and volume data with compression ratios comparable or better than recent image formats used for lossless compression.

3. Development of new lossy compression algorithm and software for compression of digital images and volume data with compression ratios comparable or better than recent lossy image formats with comparable or higher signal to noise ratio.

4. Application of volume rendering software to analysis of the Visible Human Project™ true color volume dataset.

# Introduction

The purpose of the dissertation is to introduce a new approach of general volume data analysis. Many recent measurements in engineering practise lead to a three dimensional volume representation of results which are quite hard to be analyzed in an image representation since its true nature is three dimensional. As a particular example can be noted data acquired by a confocal microscope or nuclear magnetic resonance scanners which are of great importance in medicine and biology. Because of the fact the volume data are stored in a general three dimensional matrix it can be used for analysis of any volume representation of a physical phenomenon such as results of numerical solution of differential equations etc. Therefore it is not limited only to measured datasets.

The visualization method is designed especially to simulate a human perception of observing a volume object, particularly linear perspective projection, visibility based on raycasting and color combining techniques are used. Thus one can deduce features of a volume object naturally without a proper knowledge of these methods. An important property of this visualization method is that a volume object can be observed interactively so a factual emphasis in the dissertation is aimed to render a visualization in the fastest possible time, but without a hardware acceleration, because recent accelerators do not provide sufficient support for high quality accelerated volume rendering.

The next part of the dissertation is dedicated to principles and application of digital filters to image and volumetric data, where also methods of adaptive filtering such as adaptive kernel convolution are presented with particular examples of their

use in practise. Volume analysis unfortunately brings lots of problems to solve. The most apparent problem is a vast amount of storage space required to contain all the information contained in a volume object in contrast with a less expensive 2D image representation. It is essential to overcome this problem since rather small volume data can reach beyond a capacity of todays computers. To reduce a total amount of information there are proposed two types of compression algorithms. The first, lossless algorithm is based on Burrows-Wheeler block sorting and adaptive arithmetic encoding with finite context prediction. The second is aimed to lossy compression, which is based on progressive DCT encoding.

# Chapter 1

# Digital Color Representation

*"Colour.*
*Fades to grey.*
*I dream so exciting.*
*But I, I feel so bold."*

— Seal - Colour

## 1.1 Introduction

The most popular part of image processing is processing of color images. There exist various ways to store and interpret color information present in such images. A color is often represented by three coordinates which can be imagined as a spatial vector. Thus a color can be represented by a vector in so called **color space**. Further will be discussed properties of the most frequent color spaces in order to learn how a color information can be stored and which color space is suitable for a patricular processing. More in-depth information about color spaces can be found in [12].

## 1.2 CIE Chromacity Diagram

The name CIE comes from french **Comission Internationale de l'Éclairage** and means International Commission on Illumination. It is the international authority on light, illumination, color and color spaces. The standard reference for color definition is CIE chromacity diagram (fig.1.1).

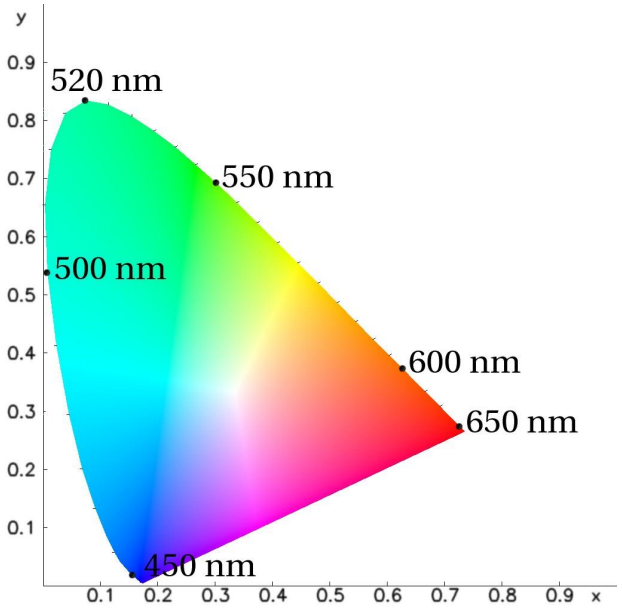



Figure 1.1: CIE color representation.

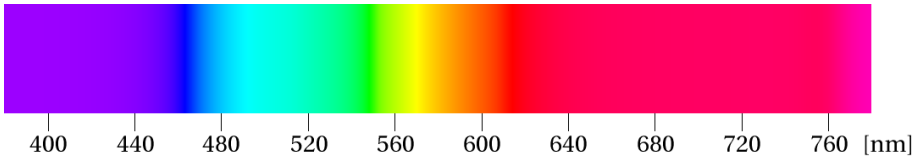



Figure 1.2: Normalized spectral colors derived from the CIE diagram.

This diagram was developed in 1931 and in full plot it is three dimensional with **tristimulus** $X, Y, Z$ coordinates. The coordinates are linear measures of light power. The $Y$ coordinate is called "luminance" which can be regarded as a measure of percepted brightness. For better operating it was transformed to 2D $x, y$ diagram where

$y$ axis shows color intensity and $x$ axis chrominance of CIE-1931 standard observer as illustrated by the transformation equations:

$$x = \frac{X}{X+Y+Z}, \qquad y = \frac{Y}{X+Y+Z}. \tag{1.1}$$

The $x, y$ coordinates are known as **chromacity** coordinates. The CIE diagram is based on visual response of this observer and is determined by physiological measurements of human color vision. Upper rounded boundary of the diagram is composed of pure spectral colors and their purety (saturation) decreases down to a white point in the center. Lower straight purple part is not a spectral color and is used as arbitrarily selected wavelength cut-off. Most frequently the color properties are described by red, green and blue $x, y$ coordinates, which are called **primary illuminants** and so called **white point**. The CIE diagram in chromacity coordinates is presented in fig.1.1 and the recommended [2] CIE $D_{65}$ 'daylight' whitepoint is:

$$x_w = 0.3127, \qquad y_w = 0.3290, \tag{1.2}$$

and the respective primary illuminants are:

$$\begin{pmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \end{pmatrix} = \begin{pmatrix} 0.64 & 0.30 & 0.15 \\ 0.33 & 0.60 & 0.06 \end{pmatrix}. \tag{1.3}$$

The $D_{65}$ primary illuminants are intended to represent 'daylight' temperatures at various correlated color temperatures, which are colors that in a well defined sence corresponds to black body color at certain temperature. The correlated color temperature for $D_{65}$ corresponds to roughly $6500K$[1].

## 1.3 RGB color space

The RGB color space is an additive color space which represents a color by a combination of three basic colors, which are red ($\lambda \approx 650$nm), green ($\lambda \approx 540$nm) and blue

[1] About 6504K since the later more proper evaluation of $hc/k$ fundamental constats ratio in the Planck formula.

($\lambda \approx 450$nm). This color space is called additive because if two different monochromatic light sources are observed, with different wavelenghts, the result we see is an addition of both colors which is percepted as a new color. The same property has the RGB color space[2]. The colors of red, green and blue were chosen because human color perception is created from three types of detectors on retina where each of them is roughly most sensitive to one of these colors. In fig.1.3 one can see that two close

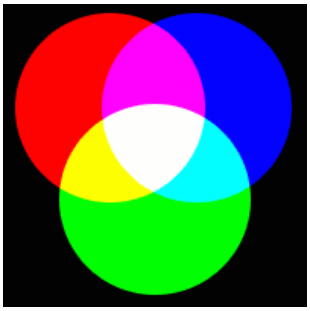


Figure 1.3: Additive R,G,B color mixing.

light sources shining red and green with the same intensity seems to be yellow. Standard representation of RGB color values is called **true color**, where each channel is encoded as 8-bit number, so it is 24 bits per color sample in total, i.e. 16 777 216 of possible colors in total.

The RGB color space is most frequently used for computer image processing because it is composed from pure spectral colors and is the most similar to mechanism of human color perception. For this reason RGB is used as the basic color space for further calculations and a conversion of any other mentioned color space[3] back to RGB is described here because of this fact.

## 1.4 CMY color space

The CMY color space is a subtractive color space where a color is represented by cyan, magenta and yellow components. In contrast to RGB color space the CMY is

[2]When no overflow in addition occurs.
[3]Except the artistic ones.

not based on a light emission but on light absorbtion. It means that the final color is composed from CMY colors which filters out only red, green or blue color component from white background. This is shown in the fig.1.4. Therefore CMY colors have to

Figure 1.4: Subtractive C,M,Y color mixing.

be colors composed from two of basic RGB colors. This is shown in table 1.1, what could be interpreted in the way that if cyan and magenta are filtered out from white background we get blue, etc.

| | red | green | blue |
|---|---|---|---|
| cyan | 0 | 1 | 1 |
| magenta | 1 | 0 | 1 |
| yellow | 1 | 1 | 0 |

Table 1.1: CMY colors expressed in RGB color space and vice versa.

The CMY color model is used in most cases in notebook displays and printing. In case of printing the CMYK color space modification where K=black is used. It is because when the CMY printer pigments are equally mixed, the black color is not the result as we could expect. For that reason it is one extra black cartridge present in recent color printers.

## 1.5 YUV color space

The YUV[4] color space separates brightness and color information of the light in the way that Y contains information about intensity of light (luminance) and U, V components represent the color (chrominance) information. In fact the YUV is a linear transformation of RGB color space. The Y component is calculated from weighted sum of R,G,B values of input signal which is based on eye sensitivity to these colors. Chrominance components are calculated relatively to Y. More precisely the U channel is obtained when B component is subtracted from Y. Similarly the V component we get when we subtract Y from R. This property is easy to see from (1.5) So the RGB $\rightarrow$ YUV conversion is:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.437 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{1.4}$$

and the backward YUV $\rightarrow$ RGB is:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.140 \\ 1 & -0.394 & -0.581 \\ 1 & 2.028 & 0 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix}. \tag{1.5}$$

The YUV color space is a part of PAL standard and is used for television broadcasting. The reason why YUV color space is used is because it can be simply viewed by either color or monochromatic televisions where U and V chrominance channels are simply ignored. In digital image processing the YUV color space is used in JPEG still image compression and MPEG digital video compression codecs.

## 1.6 YIQ color space

The YIQ color space is very similar to the YUV. It is also a linear transformation of RGB color space and is also derived from human visual system to reduce signal

[4]Also known as Y'$C_bC_r$ or $YP_bP_r$.

bandwidth. The RGB → YIQ transformation is:

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.212 & -0.523 & -0.311 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{1.6}$$

and backward YIQ → RGB is

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.105 & 1.702 \end{pmatrix} \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}. \tag{1.7}$$

YIQ is the primary color space for the NTSC television broadcasting standard.

## 1.7 HSV and HLS color space

The HSV is an artistic color space created by Alvy Ray Smith in 1978. It is nonlinear transformation of RGB space and describes color by **hue** (color type), **saturation** (purety) and **value** (intensity). So it is easier to specify a particular color because the final color does not depend on a ballance between channels[5] but on color properties percepted roughly separately by human.

- Hue component has units of degrees and represents periodic spectral color wheel where for $H = 0°$ is red color hue.
- Saturation component specifies purety of color specified by hue. Saturation of a color is defined in terms of how much white color is present in it. A color with maximal saturation is pure thus no white is present in it and any color with zero saturation is a pure level of white[6].
- Value component specifies brightness of the color.

[5] Like in above mentioned color spaces.
[6] Hence for zero saturation no hue is defined (or defined to 0).

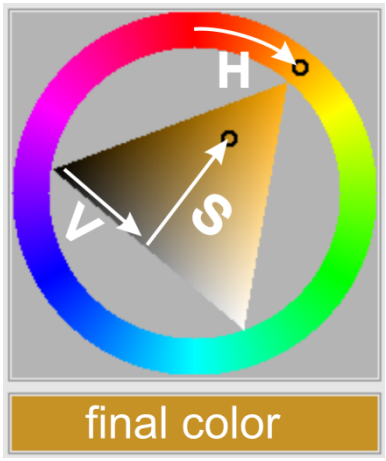



Figure 1.5: The HSV color space illustration.

A particular calculation of the $H, S, V$ color components from a $R, G, B$ color triplet is illustrated as follows, where $\Delta$ is used as a helper variable:

$$\Delta = \max(R, G, B) - \min(R, G, B) \tag{1.8}$$

$$H = \begin{cases} \left(\frac{G-B}{\Delta}\right)60 & \text{if } \max(R, G, B) = R; \\ \left(2 + \frac{B-R}{\Delta}\right)60 & \text{if } \max(R, G, B) = G; \\ \left(4 + \frac{R-G}{\Delta}\right)60 & \text{if } \max(R, G, B) = B; \end{cases} \tag{1.9}$$

$$S = \begin{cases} \frac{\Delta}{\max(R,G,B)} & \text{if } \max(R, G, B) > 0; \\ 0 & \text{if } \max(R, G, B) = 0; \end{cases} \tag{1.10}$$

$$V = \max(R, G, B). \tag{1.11}$$

The hue color wheel is composed from six segments where hues are linearly interpolated between two different components of RGB space what roughly approximates spectral color sequence with increasing hue.

There exist yet another very similar color space called **HLS**. In this color space H component is identical as in HSV, L stands for **'lightness'**[7]. The saturation and lightness are defined differently from HSV as:

$$S = \Delta, \qquad L = \Big[\max(R, G, B) + \min(R, G, B)\Big]/2. \tag{1.12}$$

[7] Also luminance, luminosity.

## 1.8 Black Body and Sun Color

The effective sun temperature is about $5780K$. It is hard to determine its color since the sun is not a perfect black body and the final percepted color is also affected by atmospheric effects. However not a big mistake will be made by an assumption that irradiation of the sun is a perfect black body. We can calculate a color of it rather easily then using the **Planck irradiation law**:

$$P(\lambda, T) = \frac{4\pi^2 \hbar c^2}{\left(e^{\frac{hc}{k\lambda T}} - 1\right)\lambda^5}, \tag{1.13}$$

what tells what is the probability of emittance of photons at a particular wavelength $\lambda$ for a black body with known temperature $T$. As a color percepted by human for a particular wavelength of light is known from the CIE diagram (fig. 1.2), it is possible to calculate black body color for a particular temperature as:

$$\vec{c}(T) = \frac{1}{P_T} \int_{\lambda_1}^{\lambda_2} P(\lambda, T)\vec{s}(\lambda) d\lambda, \tag{1.14}$$

where $\vec{s}(\lambda)$ is R,G,B color vector and $\vec{c}(T)$ is the final color of black body at temperature $T$,

$$P_T = \int_{\lambda_1}^{\lambda_2} P(\lambda, T) d\lambda \tag{1.15}$$

and the range of visible color wavelengths is $\Delta\lambda = \langle \lambda_1, \lambda_2 \rangle = \langle 450, 800 \rangle$ nm. The result for this calculation can be seen at fig.1.6, where the color assigned to the sun effective temperature is roughly white and turns to red for lower and to cyan for higher temperatures.

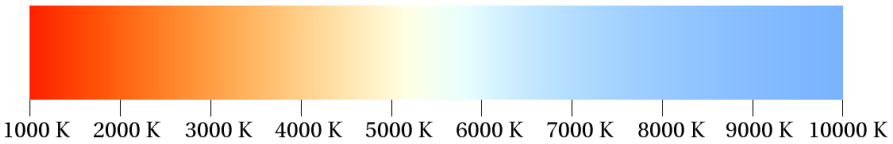



Figure 1.6: Approximate black body color at various temperatures.

# Chapter 2

# Digital filters in image processing

## 2.1 What a digital filter is?

First it is required to properly define a concept of a filter which is essential to understand methods described further. It is important to point out at the very beginning of this chapter that we are going to be involved exclusively in analysis of **digital signals**. Thus it is assumed that an input signal to be processed is sampled at a given sampling rate and quantized in a way that every sample has a finite bit depth which is the same for all the samples within the signal. Later, if we talk about a signal we mean a digital signal when not otherwise stated. By **digital filtering**[1] is meant processing samples of an input signal in the way that some arithmetics is performed to its samples to yield an output signal. Filtering methods are described in the following chapters.

## 2.2 Basic sample arithmetics

Processing of a signal means to perform numerical operations to its samples to obtain a processed signal. These operations differ from the arithmetics of real numbers what are used to employ in mathematics. It is caused by the fact that each signal,

[1] Further abbreviated only to "filtering".

therefore each its sample, must contain a finite amount of information what puts several limitations on the arithmetics. The least piece of information is called one **bit** what corresponds to a single digit number of base 2 and is called a binary number. Let us denote the two states of the number by symbols 0 and 1. Now let us define binary operations OR and XOR between two binary numbers $A$, $B$ by the following tables.

| $A$ | 1 | 0 | 1 | 0 |
|---|---|---|---|---|
| $B$ | 1 | 1 | 0 | 0 |
| $A$ OR $B$ | 1 | 1 | 1 | 0 |

| $A$ | 1 | 0 | 1 | 0 |
|---|---|---|---|---|
| $B$ | 1 | 1 | 0 | 0 |
| $A$ XOR $B$ | 0 | 1 | 1 | 0 |

Table 2.1: Logical disjunction (OR) and Exclusive logical disjunction (XOR).

Note that the operations XOR and OR are defined completely by these two tables since they are defined for any possible permutation of binary numbers $A$, $B$. Let us derive another binary operation AND from the previous two:

$$A \text{ AND } B = (A \text{ OR } B) \text{ XOR } (A \text{ XOR } B), \tag{2.1}$$

and unary NOT which calculates a binary complement of $A$ as:

$$\text{NOT } A = A \text{ XOR } 1. \tag{2.2}$$

We can also calculate all the possible results of the operations as shown above.

| $A$ | 1 | 0 | 1 | 0 |
|---|---|---|---|---|
| $B$ | 1 | 1 | 0 | 0 |
| $A$ AND $B$ | 1 | 0 | 0 | 0 |

| $A$ | 1 | 0 |
|---|---|---|
| NOT $A$ | 0 | 1 |

Table 2.2: Logical conjunction (AND) and Logical negation (NOT).

These are the basic operations in processing samples of a one-bit per sample signal. Naturally in most cases we are not interested in processing of a signal with such small samples so we need to modify these operations to let us process signals with higher bit depth.

The basic difference is that it is needed to consider a b–tuples of bits as a multi-digit binary number representing one sample. Thus each sample can represent $2^b$ possible states of a quantized phenomenon. The operation of addition is demonstrated here in order to investigate its different properties from an usual integer mathematics. It is possible to derive operation of addition from operation XOR as:

$$N(n) = N_1(n) + N_2(n) = \left[N_1(n) \text{ XOR } N_2(n)\right] \text{ XOR } C(n). \tag{2.3}$$

To make the calculation clear a notation is followed that $X(n)$ specifies n–th bit of a multi-digit binary number $X$ where $X(0)$ denotes the least significant bit. Number $C(n)$ in the previous equation has a special meaning. It is called **carry flag** and holds a temporary result of a previously performed operation. The value of $C(n+1)$ is set to 1 either if $N_1(n) = 1$ and $N_2(n) = 1$ or $N_1(n)$ XOR $N_2(n) = 1$ and $C(n) = 1$, therefore

$$C(n+1) = N_1(n) \text{ AND } N_2(n) \text{ OR } N_1(n) \text{ XOR } N_2(n) \text{ AND } C(n). \tag{2.4}$$

Let us consider an usual case of 4-bit sample addition[2] $11 + 2 = 13$, demonstrated in table 5, where is set $C(0) = 0$ because no addition has been performed before.

| $n$ | 3 | 2 | 1 | 0 | decimal |
|---|---|---|---|---|---|
| $N_1$ | 1 | 0 | 1 | 1 | 11 |
| $N_2$ | 0 | 0 | 1 | 0 | 2 |
| $N = N_1$ XOR $N_2$ | 1 | 0 | 0 | 1 | |
| $C$ | 0 | 1 | 0 | 0 | |
| $N$ XOR $C$ | 1 | 1 | 0 | 1 | 13 |

| $n$ | 3 | 2 | 1 | 0 | decimal |
|---|---|---|---|---|---|
| $N_1$ | 1 | 1 | 0 | 1 | 13 |
| $N_2$ | 0 | 1 | 0 | 0 | 4 |
| $N = N_1$ XOR $N_2$ | 1 | 0 | 0 | 1 | |
| $C$ | 1 | 0 | 0 | 0 | |
| $N$ XOR $C$ | 0 | 0 | 0 | 1 | 1 |

Table 2.3: Addition without overflow (left) and with overflow (right).

One can see the correct result. A different case of addition is presented in table 2.3 (right). The addition is $13 + 4 = 17$ but a result of 1 is obtained. This is caused by **overflow** which occurs when a result of an operation has too many bits to fit

[2]Realized by ADD or ADC instructions on x86 processors for 8,16,32 bit additions.

a number representing result. The result of overflown operation is not completely lost. Actually we see only lower significant bits of the result, i.e. 17 AND $(2^4 - 1) = 1$. The highest significant bit is in carry flag $C(4)$, which is used to detect overflow in assembly language. For instance, in table 2.3 (left) $C(4) = 0$, but in table 2.3 (right) $C(4) = 1$, what indicates overflow, so the correct result is $2^4 + 1 = 17$.

Unfortunately there is limited possibility to check the carry flag in higher level languages, so we have to overcome this problem either with usage of sufficiently bit-wide numbers whose limits never be exceeded during calculation or, a worse way, with using of **saturated** operations such as addition with saturation[3] where the result is set to maximal possible value a number can contain[4] in case of overflow.

But not only addition introduces overflow issues. Much worse overflows can be caused by multiplication. For example, in the worst case of adding two $n$ - bit numbers we need $n + 1$ bits to store the result. In the worst case of multiplication $2n$ bits is needed! When subtracting we have to face **underflow** when a result reaches under zero, etc.

Of course when we do not want to bother with overflows, one can convert integer samples into floating point numbers, do a calculation, and convert them back to their integer representation. This way of processing is followed in the next chapters to remain the text comprehensive.

## 2.3 Classification of Filters

Digital filters are often [14] divided in two cathegories dependent on the fact whether the filter can or cannot be considered as a linear system. A system is called linear if it has properties of **homogenity** and **additivity**. If one of these assumptions are not true, the system is nonlinear. To demonstrate these and other filter properties let us have a filter $\mathcal{F}$ to process an input signal $a(t)$ to output signal $b(t)$, then:

[3] Realized by a PADDSx instructions on x86 processors with MMX or SSE2 vector extensions.
[4] What leads to information loss.

- Filter $\mathcal{F}$ is called **homogenous** if amplitude change in $a$ causes the same amplitude change in $b$:

$$\mathcal{F}\big[a(t)\big] = b(t) \implies \mathcal{F}\big[ka(t)\big] = kb(t) \qquad \forall t, \tag{2.5}$$

  where $k$ is a constant.

- Filter $\mathcal{F}$ is called **additive** if processing of addition of two input signals $a_1$ and $a_2$ result in addition of two output signals $b_1$ and $b_2$:

$$\mathcal{F}\big[a_1(t)\big] = b_1(t),\ \mathcal{F}\big[a_2(t)\big] = b_2(t) \implies \mathcal{F}\big[a_1(t) + a_2(t)\big] = b_1(t) + b_2(t) \quad \forall t. \tag{2.6}$$

- Filter $\mathcal{F}$ is called **shift invariant** if shift in $a$ causes identical shift in $b$:

$$\mathcal{F}\big[a(t)\big] = b(t) \implies \mathcal{F}\big[a(t+dt)\big] = b(t+dt) \qquad \forall t. \tag{2.7}$$

The last property of shift invariance is not required to consider a system linear, but most of linear and even nonlinear filters have this property.

## 2.4 Discrete Convolution Filters

A nice example of linear filter is a convolution filter. A discrete convolution filter can be defined as:

$$b(t) = \mathcal{F}_{h(k)}\big[a(t)\big] = a(t) * h(k) = \sum_{k=-K/2}^{K/2-1} a(t-k)h(k) \quad \forall t \in \{0, 1, ..., N-1\}, \tag{2.8}$$

where $h(k)$ is **convolution kernel** of $K$ samples and $a(t)$ is a signal to be convolved. The $h(k)$ is also called **impulse response function** or **point spread function** in image processing. In this text is considered only the **finite impulse response (FIR)** or **truncated infinite impulse response (TIIR)** function $h(k)$ exclusively so that $K$ is set to some sufficiently large finite value. The name of $h(k)$ is a finite impulse response because if an impulse is passed, i.e. $a(t = t_0) = 1$, zero otherwise,

through the convolution filter, function $h(k)$ shifted of $t_0$ is obtained as the output. The sum of all samples within the kernel $h(k)$ must be a non-zero constant in case of low-pass filter what is fulfilled if it is normalized, so the condition 2.9 must hold.

$$\sum_{k=0}^{K-1} h(k) = 1. \tag{2.9}$$

An analogous condition 2.10 must hold for a high-pass filter:

$$\sum_{k=0}^{K-1} g(k) = 0. \tag{2.10}$$

Since a digital signal is processed, it is also possible to imagine discrete convolution as matrix-vector product. For instance, let us define normalized $K = 5$ convolution kernel:

$$h(k) = (0.1, 0.2, 0.4, 0.2, 0.1) \tag{2.11}$$

and a $N = 5$ sample input signal:

$$a(t) = (0.1, 0.7, 0.2, 0.3, 0.8)^T. \tag{2.12}$$

If we look at (2.8) the argument of $a$ can reach below zero and above the total samples $N$ in the signal. We can not neglect this boundary problem because signal of a finite lenght is processed. By a style how the boundary problem is solved can be distinguished various types of discrete convolution.

### 2.4.1 Cyclic Convolution

The first possibility how to solve the situation close to the beginning and end of the signal is the **cyclic convolution** which can be expressed as:

$$b(t) = \begin{pmatrix} 0.4 & 0.2 & 0.1 & 0.1 & 0.2 \\ 0.2 & 0.4 & 0.2 & 0.1 & 0.1 \\ 0.1 & 0.2 & 0.4 & 0.2 & 0.1 \\ 0.1 & 0.1 & 0.2 & 0.4 & 0.2 \\ 0.2 & 0.1 & 0.1 & 0.2 & 0.4 \end{pmatrix} \begin{pmatrix} 0.1 \\ 0.7 \\ 0.2 \\ 0.3 \\ 0.8 \end{pmatrix} = \begin{pmatrix} 0.39 \\ 0.45 \\ 0.37 \\ 0.40 \\ 0.49 \end{pmatrix}, \tag{2.13}$$

where indices reaching out of range from $a(t)$ are mapped back to a valid range by modulo $N$:

$$b(t) = \sum_{k=-K/2}^{K/2-1} a\big[(t-k) \bmod N\big] h(k) \quad \forall t \in \{0, 1, .., N-1\}. \tag{2.14}$$

This type of discrete convolution is suitable for processing of periodic signals and thanks to the convolution theorem[5] can be also calculated by FFT algorithms. Unfortunately it is rather rare case we need to process periodic images so a different kind of convolution is likely to be used in common practise.

### 2.4.2 Acyclic Convolution with Kernel Renormalization

This type of convolution solves the boundary problem in the way that any part of convolution kernel $h(k)$ which is not in valid range of $a(t)$ during calculation is set to zero. This solution comes with a slight disadvantage because the truncated kernel must be renormalized during convolution:

$$b(t) = \begin{pmatrix} 0.571 & 0.286 & 0.143 & 0 & 0 \\ 0.222 & 0.444 & 0.222 & 0.111 & 0 \\ 0.1 & 0.2 & 0.4 & 0.2 & 0.1 \\ 0 & 0.111 & 0.222 & 0.444 & 0.222 \\ 0 & 0 & 0.143 & 0.286 & 0.571 \end{pmatrix} \begin{pmatrix} 0.1 \\ 0.7 \\ 0.2 \\ 0.3 \\ 0.8 \end{pmatrix} = \begin{pmatrix} 0.286 \\ 0.411 \\ 0.370 \\ 0.433 \\ 0.571 \end{pmatrix}, \tag{2.15}$$

but because of this fact it does not require the input signal to be periodic. This type of convolution is very important in image processing as no image in general can be considered periodic and does not consume considerable amount of memory.

### 2.4.3 Acyclic Convolution with Sample Extensions

The second approach to acyclic convolution without need of kernel renormalization are sample extensions. The idea is to extend boundaries of the convolved signal $a(t)$

[5] Described in section 3.2.1.

either on left and right hand side by $K-1$ copies of the leftmost and rightmost sample so that the convolution kernel $h(k)$ will fit to the extended signal without a need to be renormalized. This approach is not very useful as the extended signal is mostly a poor estimation of how the signal would actually behave beyond the boundaries. This leads to artefactual effect such as increasing or decreasing of intensity in the convolved signal near the signal boundaries because the final convolution is too much affected by the guessed extended samples close to ends of $a(t)$.

### 2.4.4 Acyclic Convolution with Symmetric Signal Extensions

More advanced extensions of the signal can be made if the original signal is extended symmetrically on both sides. There are two possible ways how to implement the symmetric extension. The first one duplicates the first or the last sample and mirrors the rest of the signal, the second approach uses no boundary sample duplication. Both these approaches are illustrated in fig.2.1. In some cases discussed further it is useful to combine both ways. This way of signal extension is better that in the cyclic

a) With boundary sample duplication. b) Without boundary sample duplication.

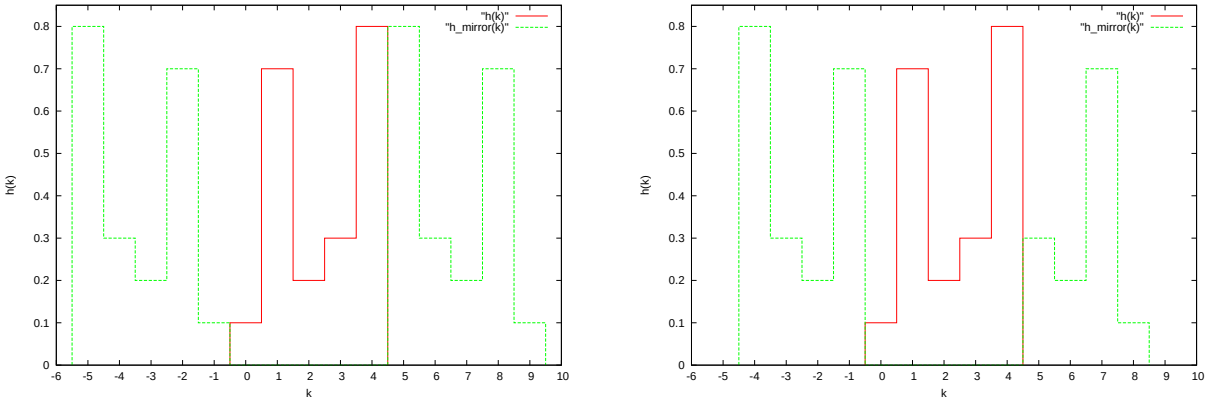



Figure 2.1: Examples of symmetric signal extensions.

convolution (sec. 2.4.1), because there exist a large disconuity between the end and the beginning of a real-life signal if we use the cyclic convolution.

### 2.4.5 Adaptive Kernel Convolution

In section 2.4.2 is demonstrated that for some signals it is appropriate to change a shape of kernel during convolution. If the idea is improved a little bit, we can say that we need not only change $h(k)$ at the end and beginning of $a(t)$ but also let $h(k)$ be somehow dependent on contents of $a(t)$ itself. We call it **adaptive kernel convolution** when this approach is used.

One application of this type of convolution is demonstrated here which is very suitable for error elimination from images. The divide and conquer strategy is followed here to fulfill this task, so the method itself consists of not one but a few simple steps. The calculation consists of three steps, where adaptive kernel convolution is used in the end:

1. Low-pass filtering of the original image $I_1(x, y)$ to obtain processed image $I_2(x, y)$.
2. Detection of defect pixels based on difference checking between $I_1$ and $I_2$.
3. Replacing of defect pixels and their interpolation using acyclic adaptive kernel convolution.

For a given image $I_1(x, y)$ 2D discrete acyclic convolution with symmetric TIIR gaussian kernel of $K$ samples is calculated in each dimension to obtain $I_2$:

$$I_2(x,y) = \mathcal{F}_{h(k_x,k_y)}\big[I_1(x,y)\big] = I_1(x,y) * h_1(k_x,k_y) = I_1(x,y) * \left[e^{-\frac{k_x^2+k_y^2}{\sigma_1^2}}\right]_K \tag{2.16}$$

where $K$ is large enough to represent the point spread function $h_1(k_x, k_y)$. The next step is to mark defective pixels. An information about defective pixels is held in error matrix $E(x, y)$, which is calculated by using the following bad pixel criterion:

$$E(x,y) = \begin{cases} 0 & \text{if } |I_1(x,y) - I_2(x,y)| \leq I_t; \\ 1 & \text{otherwise.} \end{cases} \tag{2.17}$$

where $I_t$ is appropriatelly selected threshold. When $E(x, y)$ is calculated, we perform adaptive kernel convolution (AKC) of $I_1(x, y)$ for every $E(x, y) = 1$ in order to interpolate bad pixel in $I_1(x, y)$. Even if it is a 2D problem (and might be 3D problem

for volume data) it can be reduced to one dimensional AKC, what can be done by creating a vector function $\vec{p}(n)$ which contains coordinates of neighbouring pixels in not decreasing distance order with increasing $n$. It is shown in the fig.2.4. The filtered

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 151 | 129 | 101 | 89 | 81 | 69 | 82 | 90 | 102 | 130 | 152 |
| 131 | 97 | 70 | 61 | 49 | 45 | 50 | 62 | 71 | 98 | 132 |
| 103 | 72 | 57 | 37 | 29 | 25 | 30 | 38 | 58 | 73 | 104 |
| 91 | 63 | 39 | 21 | 13 | 9 | 14 | 22 | 40 | 64 | 92 |
| 83 | 51 | 31 | 15 | 5 | 1 | 6 | 16 | 32 | 52 | 84 |
| 74 | 46 | 26 | 10 | 2 | 0 | 3 | 11 | 27 | 47 | 75 |
| 85 | 53 | 33 | 17 | 7 | 4 | 8 | 18 | 34 | 54 | 86 |
| 93 | 65 | 41 | 23 | 19 | 12 | 20 | 24 | 42 | 66 | 94 |
| 105 | 76 | 59 | 43 | 35 | 28 | 36 | 44 | 60 | 77 | 106 |
| 133 | 99 | 78 | 67 | 55 | 48 | 56 | 68 | 79 | 100 | 134 |
| 157 | 135 | 107 | 95 | 87 | 80 | 88 | 96 | 108 | 136 | 158 |

Table 2.4: Pixel ordering $n$ in non-decreasing distance from the center (Euclid metric).

image is then calculated by AKC as:

$$I_1(x,y) = \frac{1}{||h_2||} \sum_{n=1}^{N-1} I_1\big[(x,y) - \vec{p}(n)\big] h_2(n,x,y) \qquad \forall E(x,y) = 1, \tag{2.18}$$

where

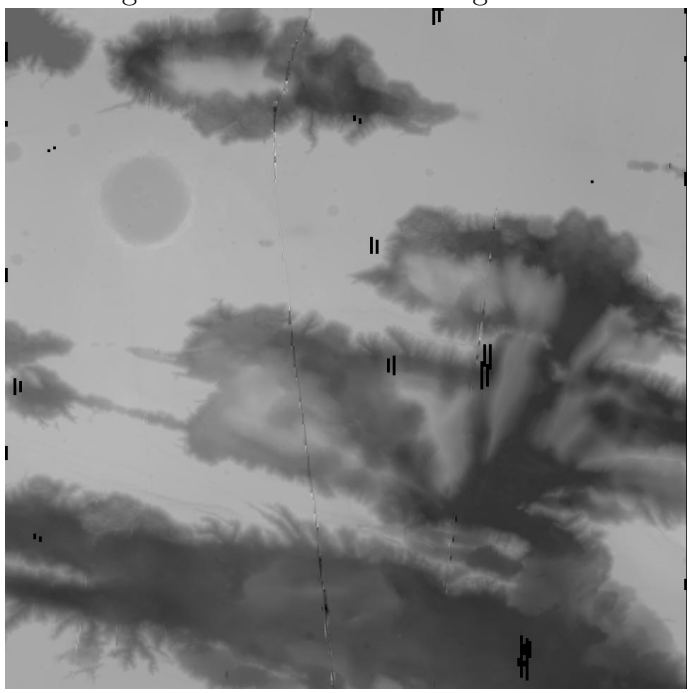
$$h_2(n,x,y) = \begin{cases} e^{-\frac{|\vec{p}(n)|^2}{\sigma_2^2}} & \text{if } E\big[(x,y) + \vec{p}(n)\big] = 0; \\ 0 & \text{otherwise.} \end{cases} \tag{2.19}$$

is adaptive kernel constructed from TIIR gaussian kernel which expresses decreasing weight of convolved pixel with increasing distance $|\vec{p}(n)|$. The norm $||h_2||$ is calculated during AKC as:

$$||h_2|| = \sum_{n=1}^{N-1} h_2(n,x,y). \tag{2.20}$$

Note that the norm is also dependent on position $x, y$ by equation 2.19.

The original MOLA scanned image with errors.



Filtered by adaptive kernel convolution.

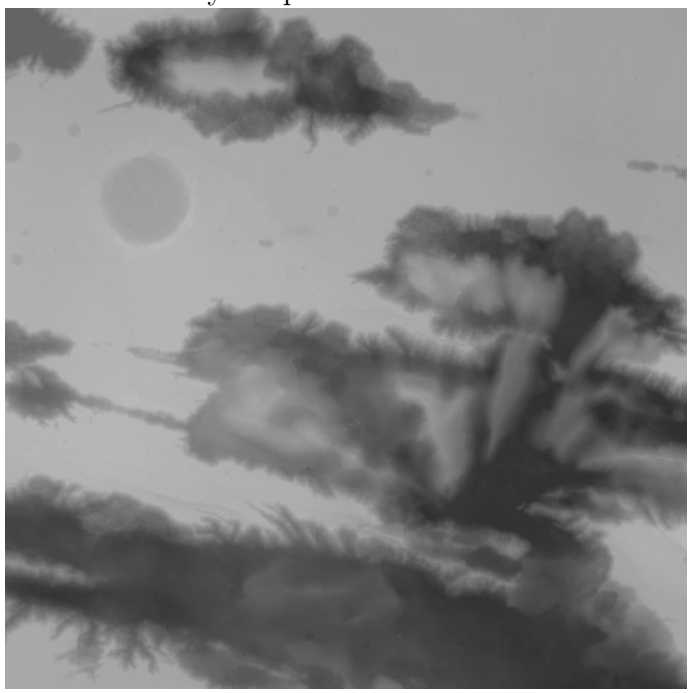


Figure 2.2: Usage of the adaptive kernel convolution.

# Chapter 3

# Fourier Transform

*"I never understood the frequency, uh-huh.*
*'What's the frequency Kenneth?' is your Benzedrine, uh-huh."*

— R.E.M. - What's the frequency Kenneth?

## 3.1 Fourier Transform Algorithms

Fourier transform techniques are frequently used in many physical and image processing operations in frequecy domain analysis and is very helpful part of signal processing. There exist several methods how to calculate the Fourier transform of a discrete signal. The most simple one is the discretized variant of the Fourier transform called **discrete Fourier transform (DFT)**. One dimensional DFT is defined as

$$F(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n) e^{-i2\pi kn/N}, \tag{3.1}$$

and backward (inverse) DFT as

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} F(k) e^{i2\pi kn/N}, \tag{3.2}$$

where $f(n)$ is the signal, $F(k)$ its spectrum and $N$ the total number of samples. The calculation of DFT is unfortunately rather expensive. In case of forward DFT

(eq.3.1) it is needed to calculate $N$ complex multiplications for one spectral coefficient $k$. As the total number of $k$'s is $N$, $N^2$ complex multiplications are needed to calculate forward DFT[1]. In computer jargon can be said that an algorithmic complexity of DFT is $O(N^2)$. Fortunately there exist a couple of algorithms intended to make a DFT calculation more effective. These algorithms are known as **fast Fourier transform (FFT)** algorithms and have algorithmic complexity of $O(N \log N)$. To demonstrate this fact let us consider $N = 4096$ point input signal. To calculate FFT is few hundred times[2] faster than DFT! This is the reason why in practise FFT algorithms are used and why we are going to be involved in principles and implementations of basic FFT algorithms in detail. Further information can be found in excellent article [5].

### 3.1.1 Radix 2 Decimation in Time FFT Algorithm

Basic idea behind the Decimation in Time (DIT) FFT can be described by **Danielson-Lanczos Lemma** which shows that a DFT of length $N$ can be calculated as a sum of two DFT's of length $N/2$ where each $N/2$ DFT is composed either from even or odd samples from the original signal. If an unnormalized DFT is considered, the Danielson-Lanczos Lemma can be written as

$$\begin{aligned} F(k) &= \sum_{n=0}^{N-1} f(n) e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} f(2n) e^{-i2\pi k2n/N} + \sum_{n=0}^{N/2-1} f(2n+1) e^{-i2\pi k(2n+1)/N} \\ &= \sum_{n=0}^{N/2-1} f(2n) e^{-i2\pi kn/(N/2)} + e^{-i2\pi k/N} \sum_{n=0}^{N/2-1} f(2n+1) e^{-i2\pi kn/(N/2)}. \end{aligned} \tag{3.3}$$

Result of the Danielson-Lanczos Lemma can be rewritten in the abbreviated form as

$$F(k) = F_{even}(k) + T(N,k) F_{odd}(k), \tag{3.4}$$

[1] Apparently the same number of operations is needed to calculate its inverse (3.2).
[2] For details see section 3.1.7.

which is called (DIT) **"butterfly"**. In (3.4) $F_{even}(k)$ stands for DFT from even samples and $F_{odd}(k)$ for odd samples of the original signal.

$$T(N,k) = e^{-i2\pi k/N} \tag{3.5}$$

is called (DIT) **"twiddle" factor**. A practical impact of this lemma is that by application of it a significant number of operations can be spared. It is apparent the $N$ point DFT requires $N^2$ complex multiplications. After application of this lemma it is reduced to $N^2/2$. We cannot say that the computational load is halved in respect to the $N$ point DFT since some overhead for twiddle factor multiplication and summation is needed. However the total calculation time should be close to $1/2$ of the $N$ point DFT as these operations are not quite expensive.

The idea of FFT is that the Danielson-Lanczos Lemma can be applied recursively. So in fact we need not to perform $N/2$ point DFT but two $N/4$ point DFT's for each $N/2$ transform etc. This is the reason why a calculation of FFT is extremely simple when total number of input samples obeys the condition $N = 2^m$, where $m$ is a positive integer. In that case we can decimate the input signal to even and odd halves up to one sample DFT is reached. A result of single sample DFT is the sample itself. So what is needed now is to perform a backward hierarchy of butterfies, i.e. two point DFT's[3], and summation to obtain the result.

Fortunately it is not needed to perform even and odd samples decimation during the FFT calculation itself. Much faster way is to convert the input signal to the bit reversed order (see 3.1.3). My in-place implementation of DIT FFT algorithm of $N = 2^m$ samples in bit reversed order is

```
void dit_fft( int p, int N, __complex__ *f ) {
    if ( N>1 ) {                    /* if the signal size is greater than one */
        int N2=N>>1, p2=p+N2, k;
        __complex__ t, b;

        dit_fft(p,  N2, f);         /* do FFT on even samples */
        dit_fft(p2, N2, f);         /* do FFT on odd samples */
```

[3]This is why this sort of algorithms are called Radix 2.

```
        for ( k=0; k<N2; k++ ) {
            t = f[p +k];              /* calculate DIT butterflies */
            b = f[p2+k] * ( cos(2.*M_PI*k/N) - I*sin(2.*M_PI*k/N) );
            f[p +k] = t + b;
            f[p2+k] = t - b;
        }
    }
}
```

What is especially nice is that the implementation directly follows our mathematical derivation. To calculate DIT FFT of our signal we should call it as `dit_fft(0,N,f)`.

### 3.1.2 Radix 2 Decimation in Frequency FFT Algorithm

A different but similar approach can be used to implement the FFT algorithm. In previous chapter is discussed FFT implementation which is based on decimation in time. The second type of implementation of FFT algorithm is based on decimation in frequency (DIF) and can be more suitable for some cases discussed further.

Derivation of DIF FFT algorithm can be based on slightly modified Danielson-Lanczos Lemma. The difference is that we will not decimate the original signal to even and odd samples but we will use the fact that $N$ point DFT can be expressed as a summation of two $N/2$ DFT's of each half of the original signal. Therefore

$$\begin{aligned} F(k) &= \sum_{n=0}^{N/2-1} f(n)e^{-i2\pi kn/N} + \sum_{n=N/2}^{N-1} f(n)e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} \Big[ f(n) + f(n+N/2)e^{-i\pi k} \Big] e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} \Big[ f(n) + f(n+N/2)(-1)^k \Big] e^{-i2\pi kn/N}. \end{aligned} \tag{3.6}$$

From the last expression in (3.6) can be seen that it is worth to separate the one DFT

to two $N/2$ point DFT's for even and odd $k$. So one can write

$$\begin{aligned} F(2k) &= \sum_{n=0}^{N/2-1} \Big[ f(n) + f(n+N/2) \Big] e^{-i2\pi 2kn/(N/2)} \\ F(2k+1) &= \sum_{n=0}^{N/2-1} \Big[ f(n) - f(n+N/2) \Big] e^{-i2\pi n/N} e^{-i2\pi 2kn/(N/2)}, \end{aligned} \tag{3.7}$$

which shows the DIF butterfly needed to be calculated in our implementation. this algorithm is called Decimation in Frequency FFT just because the frequency components are divided to even and odd parts. My in-place implementation follows.

```
void dif_fft( int L, int N, __complex__ *f ) {
    if ( N>1 ) {                    /* if the signal size is greater than one */
        int N2=N>>1, R=L+N2, n;
        __complex__ l, r;

        for ( n=0; n<N2; n++ ) {
            l = f[L+n];
            r = f[R+n];
            f[L+n] =  l + r;        /* calculate DIF butterflies */
            f[R+n] = (l - r) * ( cos(2.*M_PI*n/N) - I*sin(2.*M_PI*n/N) );
        }

        dif_fft( L, N2, f );        /* do FFT on left half */
        dif_fft( R, N2, f );        /* do FFT on right half */
    }
}
```

### 3.1.3 Radix 2 Bit Reversal Sample Reordering

In both DIT or DIF FFT algorithms is needed to perform sample reordering. The principal difference is that in case of DIT algorithm we have to do reordering before FFT (in time domain) and in case of DIF algorithm after FFT (in frequency domain). The reordering can be done during the calculation what is rather slow since a reordering of all the samples is needed. Fortunately sample reordering can be excluded completely from the calculation process.

The task of this section is to simplify a process of recursive decimation of input signal to its even and odd samples. If we consider we have $N = 8$ samples in our

input signal (so that $N = 2^b$, where $b = 3$) the even/odd decimation of the signal is shown in the middle of table 3.1.

| level 1 in binary |
|---|
| 000 |
| 001 |
| 010 |
| 011 |
| 100 |
| 101 |
| 110 |
| 111 |

| level 1 | level 2 | level 3 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 2 | 4 |
| 2 | 4 | 2 |
| 3 | 6 | 6 |
| 4 | 1 | 1 |
| 5 | 3 | 5 |
| 6 | 5 | 3 |
| 7 | 7 | 7 |

| level 3 in binary |
|---|
| 000 |
| 100 |
| 010 |
| 110 |
| 001 |
| 101 |
| 011 |
| 111 |

Table 3.1: Recursive even/odd sample decimation.

It is shown here that we need $b = \log_2 N$ levels of recursion to decimate the signal to set of two point signals. In the middle table, for the smallest level of recursion, can be seen the original sample ordering and for highest level the final ordering is shown. To help us realize the reordering is pretty simple[4] there are two tables added which shows the input and output sample indices as binary numbers. The only thing to do is to reverse bits in the index numbers! Furthermore, if we consider in-place reordering the most of samples will hold their original position (they have the same index), otherwise the two samples should be swapped. For instance, our bit reversal reordering filter is that we swap samples $1 \leftrightarrow 4$ and $3 \leftrightarrow 6$. In analogous way can reordered any $N = 2^b$ point signal. My implementation of bit reversal reordering of complex signal `s` which has `b` $= \log_2 N$ bit indices follows.

```
void reorder( __complex__ *s, int b ) {
    int i, n, l, h;
    __complex__ t;

    for ( i=1; i<(1<<b)-1; i++ ) {          /* for all[5] the orders do */
        for ( l=1, h=1<<(b-1), n=0; h; ) { /* calculate bit reversed order */
            n   = i&l?n^h:n;
            l <<= 1;
```

[4]What is not apparent when we use decimal numbers.

[5]Except the first and last where swapping can never happen.

```
            h >>= 1;
        }
        if ( n>i ) {     /* swap only if bit reversed order>original order */
            t = s[n];
            s[n] = d[i];
            s[i] = t;
        }
    }
}
```

### 3.1.4 Normalized FFT

One more important thing to mention is the FFT algorithms calculate unnormalized DFT of the input signal. Therefore the amount of energy present in the input signal is not conserved in frequency domain after FFT. The discrete variant of Parseval's Theorem can be used to fix it. So the following equation

$$\sum_{k=0}^{N-1} F(k)F^*(k) = \frac{1}{N}\sum_{n=0}^{N-1} f(n)f^*(n) \tag{3.8}$$

must hold to let the signal and its spectrum contain the some amount of energy. In other words it means we have to divide each sample of the entire signal by $N$ either before or after FFT.

### 3.1.5 Inverse FFT

Up to now we have discussed only forward FFT algorithms. There are a couple of ways to calculate backward (inverse) FFT. The simplest one is to realize the forward and backward DFT differs only in the sign in the Fourier basis[6] (3.1), (3.2). Therefore if we want to make IFFT from FFT, the only thing to change is the sign in the twiddle factor (3.5). Alternatively we can calculate IFFT directly by an unchanged FFT algorithm because

$$f^*(n) = \frac{1}{N}\sum_{k=0}^{N-1} F^*(k)e^{-i2\pi kn/N}. \tag{3.9}$$

[6]Because the forward and backward Fourier bases are complex conjugated.

Hence after conjugation of $F(k)$, performing FFT, normalization and conjugation of the result $f(n)$ we have the inverse FFT.

### 3.1.6 Multidimensional FFT

It is not needed to change the FFT algorithm completely in order to perform multidimensional FFT. The basic idea is to compose the multidimensional transformation from one dimensional FFT with a sample stride. We can demonstrate this fact on the following 2D FFT example.

We will theat a 2D $X \times Y$ point complex signal $f(x, y)$ as one dimensional $N = XY$ point $f(x)$. Then one dimensional $N$ point FFT of $f(x)$ with stride $S$ is denoted $FFT_S[f(x), N]$. To calculate 2D FFT we need to perform 1D FFT's of all rows and all columns present in $f(x)$:

$$\begin{aligned} f_1(x) = F(k_x, y) &= FFT_1[f(yX), X] & \forall y \in \{0, 1, ..., Y-1\} \\ F(k_x, k_y) &= FFT_X[f_1(x), Y] & \forall x \in \{0, 1, ..., X-1\} \end{aligned} \tag{3.10}$$

This idea can be easily generalized to more dimensions. Thus the only thing to modify in presented FFT algorithms is a sample stride during the FFT calculation.

### 3.1.7 Algorithmic Complexity of FFT Algorithms

We know the FFT algorithms have the complexity of $O(N \log N)$. It tells an approximate number of operations needed to calculate FFT of a signal consisting of $N$ samples. Because of the fact this value is only an estimation, we present here exact number of butterfly operations calculated by FFT in order to do slightly more precise prediction of calculation time. If we look back on table 3.1, it can be seen that $m = \log_2 N$ levels of recursion are needed, where $N = 2^m$ is the total number of samples, $m > 0$, $m$ is an integer, to decimate the input signal to $N/2$ two-point DFT signals. So we have to calculate $N/2$ butterflies for the last level of recursion. If the recursion is backtraced, it can be seen that it is needed to calculate $N/2$ butterflies

for each level of recursion, so the total number of butterflies is:

$$B = \frac{N}{2}m = \frac{N}{2}\log_2 N. \tag{3.11}$$

Thus, for instance, FFT of a $N = 4096$ point signal requires $B = 24576$ butterflies to be calculated. DFT of the same signal requires 16777216 Fourier base multiplications[7]. If the scaling constant and logarithm base is neglected we obtain the FFT algrorithm complexity of $O(N \log N)$.

## 3.2 Useful FFT Filters

### 3.2.1 FFT Convolution

It is possible to calculate cyclic convolution[8] using FFT. That is because of the **convolution theorem**. It says that a cyclic convolution in space domain is equal to multiplication in frequency domain:

$$a * h = F^{-1}\Big[F(a)F(h)\Big], \tag{3.12}$$

where $F$ is normalized forward FFT and $F^{-1}$ is inverse FFT.

### 3.2.2 Phase Correlation

Phase correlation was used to fit photographed slices of the Visible Human Project™ cadaver together in order to do volume rendering. The phase correlation is defined as

$$P = F^{-1}\left[\frac{F^*(a)F(b)}{|F^*(a)F(b)|}\right], \tag{3.13}$$

where $a$ and $b$ are the shifted images. After calculation of phase correlation (3.13), ten biggest peaks in $P$ is found and for all the peaks is checked the difference between the original image $a$ and shifted image $b$ by means of error metric (6.13). The shift, where the metric gives the smallest error is chosen to be correct. The results of correcting 248 slices of resolution $2069 \times 1554$ is shown in fig.5.4.

[7] Which is almost as expensive as butterfly calculation.

[8] see section 2.4.

# Chapter 4

# Wavelets

## 4.1 Why wavelets?

Up to now we have discussed only transforms calculating decomposition of the entire signal to a linear combination of non-localized basis vectors. Thus in general, a change of one coefficient in transform domain affects all samples in reconstructed signal. The right opposite is the case of wavelets. The most apparent comparison can be made if we consider FFT of a time domain signal. By FFT of a signal we obtain an information about its frequency components precisely, but we can say absolutely nothing about its frequency content in a particular time interval shorter than a total duration of the signal. In case of wavelets one can distinguish local features of a signal so an information about signal in limited time and frequency band can be obtained. This is the reason why are wavelets frequently used for feature detection and compression.

## 4.2 Uncertainty Principle and Wavelets

In quantum physics the **Heisenberg uncertainty principle** says that there exist a limitation in accurancy of nearly simultaneous measurements of observables such

as a particle position and it is momentum so that:

$$\Delta x \Delta p \geq \frac{\hbar}{2}. \tag{4.1}$$

It says that product of dispersions in measurements of a particle position and momentum never goes below a certain limit. In other words it could be interpreted that the more accurately is a position of a particle measured the worse results in measurements of momentum can be seen and vice versa. This rule does not hold for position and momentum observables exclusively. There is many of such observables for which a similar condition holds.

This is what the uncertainty principle has in common with wavelets. Two of such observables are frequency and time. Wavelets are localized in both time and frequency domains and we cannot exactly know what frequency exists at what time instantly, but we can only know what frequency bands exist at what time intervals, both with limited finite width related by similar relation like 4.1, what is an analogy with the Werner Heisenberg's uncertainty principle found in 1927.

## 4.3 Properties of a General Wavelet Transform

A general wavelet transform can be described as a conversion of an input signal $f(t)$ to a weighted superposition of base functions $\psi_i$:

$$f(t) = \sum_{i=0}^{N-1} c_i \psi_i(t), \tag{4.2}$$

where the functions $\psi_i(t)$ are localized in both time[1] and frequency domains and should match features of the input signal so that less coefficients $c_i$ are needed to describe the input signal. The localization of $\psi_i$ is a very useful feature as almost all real–life signals are limited in space and frequency domain what leads to better decorrelation. The most frequently used characteristics of wavelets is a number of

[1] Or space domain in case of images.

vanishing moments, what is actually a measure of how a function is smooth:

$$\int_a^b f(x)x^i dx = 0, \tag{4.3}$$

for $i = 0, 1, ...n-1$, where $n$ is a number of vanishing moments. The higher is the number of vanishing moments in a wavelet basis, the better smooth signals are approximated by it.

Another useful property of the wavelet decomposition is that a signal can be analyzed at various resolution levels. It is mostly apparent in the dyadic wavelet decomposition discussed further. There exist fast discrete algorithms performing wavelet decomposition with complexity $O(n)$ [15]. One of these algorithms is described here.

## 4.4 Wavelets and Multi–Resolutional Analysis

The definition of wavelets can be based on multi–resolutional analysis approach as noted in [15]. Let us consider a vector space of square integrable functions in $\mathcal{R}$:

$$L^2 = \Big\{ f : \int_{-\infty}^{+\infty} f^2(x)dx < \infty \Big\}. \tag{4.4}$$

In multi–resolutional analysis we decompose $L^2$ to nested subspaces $V_j$ such that their union is $L^2$:

$$\bigcup_{j=-\infty}^{+\infty} V_j = L^2, \tag{4.5}$$

and their intersection contains only the zero function:

$$\bigcap_{j=-\infty}^{+\infty} V_j = \big\{ \emptyset \big\}. \tag{4.6}$$

In the dyadic case, i.e. when each subspace $V_j$ is twice as large as $V_{j-1}$, a function $f(x)$ that belongs to one of these subspaces $V_j$ has the following properties:

$$\begin{aligned} f(x) \in V_j &\Longleftrightarrow \text{dilatation } f(2x) \in V_{j+1}, \\ f(x) \in V_0 &\Longleftrightarrow \text{translation } f(x+1) \in V_0. \end{aligned} \tag{4.7}$$

If we can find a function $\phi(x) \in V_0$ such that the set of functions consisting of $\phi(x)$ and its integer translates:

$$\phi(x-k), \quad k \in \mathcal{Z} \tag{4.8}$$

form a basis for the space $V_0$, we call it a **scaling function** or **father function**. For the other subspaces $V_j$ (with $j \neq 0$) we define:

$$\phi_{j,k}(x) \equiv 2^{j/2}\phi(2^j x - k). \tag{4.9}$$

Because the subspaces $V_j$ are nested:

$$V_j \subset V_{j+1}, \tag{4.10}$$

we can decompose $V_{j+1}$ to $V_j$ and $W_j$ what is the orthogonal complement of $V_j$ in $V_{j+1}$:

$$\begin{aligned} V_j \oplus W_j &= V_{j+1}, \\ W_j &\perp V_j. \end{aligned} \tag{4.11}$$

The direct sum of the subspaces $W_j$ is equal to $L^2$:

$$\bigcup_{j=-\infty}^{+\infty} V_j = \bigoplus_{j=-\infty}^{+\infty} W_j = L^2. \tag{4.12}$$

This means that $V_j$ is a "coarse resolution" representation of $V_{j+1}$, while $W_j$ holds the "high–resolution" difference information between $V_{j+1}$ and $V_j$. If we can find a function $\psi(x) \in W_0$ that obeys the translation property:

$$\psi(x) \in W_0 \Longleftrightarrow \text{dilatation } \psi(x+1) \in W_0, \tag{4.13}$$

and such that the set of functions consisting of $\psi(x)$ and its integer translates

$$\psi(x-k), \quad k \in \mathcal{Z} \tag{4.14}$$

form a basis for the space $W_0$, we call it a **wavelet function** or **mother function**. For the other subspaces $W_j$ (with $j \neq 0$) we define:

$$\psi_{j,k}(x) \equiv 2^{j/2}\psi(2^j x - k). \tag{4.15}$$

## 4.5 The Fast Wavelet Transform

A fast method how to calculate discrete wavelet transform is called **fast wavelet transform**, which is realized by the **filter bank algorithm**. Because both $V_0$ and $W_0$ are subspaces of $V_1$:

$$V_0 \in V_1 \text{ and } W_0 \in V_1,$$

we can express $\phi(x)$ in terms of the basis functions of $V_1$:

$$\begin{aligned} \phi(x) &= 2\sum_k h_k \phi(2x-k), \\ \psi(x) &= 2\sum_k g_k \phi(2x-k). \end{aligned} \tag{4.16}$$

Because of the mutli–resolutional analysis, these relations are also valid between $V_{j+1}$, $V_j$ and $W_j$ for arbitrary $j$. The filter coefficients $h_k$ and $g_k$ uniquely define the scaling function $\phi(x)$ and wavelet $\psi(x)$. The filter $h_k$ acts like a low pass filter and $g_k$ is a high pass filter. Since $V_{j+1} = V_j \oplus W_j$, we can express a function $f(x)$ that is written in terms of the basis functions $V_{j+1}$ in terms of the basis functions of $V_j$ and $W_j$ also:

$$f(x) = \sum_k \lambda_{j+1,k}\phi_{j+1,k}(x) = \sum_l \lambda_{j,l}\phi_{j,l}(x) + \sum_l \gamma_{j,l}\psi_{j,l}(x), \tag{4.17}$$

with the transform coefficients $\lambda_{j,l}$ and $\gamma_{j,l}$ defined by:

$$\begin{aligned} \lambda_{j,l} &= \sqrt{2}\sum_k h_{k-2l}\lambda_{j+1,k}, \\ \gamma_{j,l} &= \sqrt{2}\sum_k g_{k-2l}\lambda_{j+1,k}. \end{aligned} \tag{4.18}$$

This is called **Fast Wavelet Transform (FWT)** and has computational complexity of $O(n)$. The calculation itself is done in a way that $h_k$ and $g_k$ filters are applied to $f(x)$, then the results are decimated by half so the both filtered signals have the same size as the original in total. The low–pass filtered part can be again successively processed in the similar way.

## 4.6 Orthogonal Wavelets

If the $\phi_{j,k}(x)$ and $\psi_{j,k}(x)$ are orthonormal:

$$\begin{aligned} V_j &\perp W_j, \\ \langle \phi_{j,l}, \phi_{j,l'} \rangle &= \delta_{l-l'}, \\ \langle \psi_{j,l}, \phi_{j',l'} \rangle &= \delta_{l-l'}\delta_{j-j'}, \end{aligned} \tag{4.19}$$

then we can calculate the coefficients of the decomposition:

$$f(x) = \sum_l \lambda_{j,l}\phi_{j,l}(x) + \sum_l \gamma_{l,j}\psi_{j,l}(x) \tag{4.20}$$

by taking inner products with the scaling and wavelet functions:

$$\begin{aligned} \lambda_{j,l} &= \langle f, \phi_{j,l} \rangle, \\ \gamma_{j,l} &= \langle f, \psi_{j,l} \rangle. \end{aligned} \tag{4.21}$$

Decomposition to the wavelet basis is stable, so if the function $f(x)$ is only slightly changed, coefficients $\lambda_{j,l}$, $\gamma_{j,l}$ are changed slightly as well. The Parseval's identity holds for orthonormal wavelets:

$$||f||^2 = \sum_k \lambda^2_{j+1,k} = \sum_l \lambda^2_{j,l} + \sum_l \gamma^2_{j,l}. \tag{4.22}$$

## 4.7 Daubechies Orthogonal Wavelets

Daubechies [3] constructed scaling function $\phi(x)$ and wavelet $\psi(x)$ with 2 vanishing moments. The simplest case of Daubechies scaling function and wavelet is generated by four–coefficient low–pass $h_k$ and high–pass $g_k$ filters. This type is called "DAUB4" or 4–tap Daubechies wavelet filters. The 4–tap Daubechies filters have these coefficients:

$$\begin{aligned} h_k = \{&\texttt{0.4829629131445341, 0.8365163037378079,} \\ &\texttt{0.2241438680420134, -0.1294095225512604}\} \end{aligned} \tag{4.23}$$

for low–pass filter and

$$g_k = \{\texttt{-0.1294095225512604, -0.2241438680420134,} \\ \texttt{0.8365163037378079, -0.4829629131445341}\} \tag{4.24}$$

for high–pass filter. After a more careful look at the both filters we see they are not completely uncorrelated. In fact, if we denote $h_k$ coefficients as $c_0, c_1, c_2, c_3$ we see that $g_k = \{+c_3, -c_2, +c_1, -c_0\}$. So that we are able to create a high–pass filter $g_k$ only from knowledge of $h_k$ in the way that we mirror the coefficients and swap sign of odd coefficients. These filters are called **quadrature mirror filters (QMF)** in signal processing, because the coefficients are mirrored and:

$$\sum_k h_k^2 = \sum_k g_k^2, \tag{4.25}$$

i.e. sum of their quadrates equals for both the filters. The forward DAUB4 wavelet decomposition of a discrete signal $s$ of $N$ samples can be then described as a cyclic convolution (see 2.4.1). To demonstrate this we write it as a matrix–vector product:

$$t = \mathbf{M}s =$$

$$\begin{pmatrix} l_0 \\ h_1 \\ l_2 \\ h_3 \\ \vdots \\ l_{N-4} \\ h_{N-3} \\ l_{N-2} \\ h_{N-1} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 & & & \cdots & & & & \\ c_3 & -c_2 & c_1 & -c_0 & & & \cdots & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & & \\ & & c_3 & -c_2 & c_1 & -c_0 & & & & & \\ \vdots & \vdots & & & & & \ddots & & & & \\ & & & & & & & c_0 & c_1 & c_2 & c_3 \\ & & & & & & & c_3 & -c_2 & c_1 & -c_0 \\ c_2 & c_3 & & & & & & & & c_0 & c_1 \\ c_1 & -c_0 & & & & & & & & c_3 & -c_2 \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_{N-4} \\ s_{N-3} \\ s_{N-2} \\ s_{N-1} \end{pmatrix}. \tag{4.26}$$

This demonstrates the first pass of the wavelet decomposition, i.e. transformation of the first resolution level of a signal $s$ to low–pass coefficients $l_n$ and high–pass $h_n$. This is an analogy of 4.11 where we try to express $V_{j+1}$ as $V_j \oplus W_j$. For that we need

$V_j$ to be orthogonal to $W_j$ what is achieved by the fact that matrix $M$ is orthogonal, so its inverse must be its transpose:

$$\mathbf{M^T} = \mathbf{M^{-1}} = \begin{pmatrix} c_0 & c_3 & & & \dots & & & & & c_2 & c_1 \\ c_1 & -c_2 & & & \dots & & & & & c_3 & -c_0 \\ c_2 & c_1 & c_0 & c_3 & & & & & & & \\ c_3 & c_0 & c_1 & -c_2 & & & & & & & \\ \vdots & \vdots & & & \ddots & & & & & & \\ & & & & & c_2 & c_1 & c_0 & c_3 & & \\ & & & & & c_3 & -c_0 & c_1 & -c_2 & & \\ & & & & & & & c_2 & c_1 & c_0 & c_3 \\ & & & & & & & c_3 & -c_0 & c_1 & -c_2 \end{pmatrix}. \quad (4.27)$$

From the comparison of 4.26 and 4.27 one can see immediatelly that the following conditions must be hold to fulfill it:

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2 c_0 + c_3 c_1 &= 0 \end{aligned} \quad (4.28)$$

These equations are not sufficient as we have four unknowns, so we will require the coefficients to have 2 vanishing moments (4.3) what yields these additional equations:

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \quad (4.29)$$

These equations were first found and solved by Daubechies [3] and have the following solution in analytical form:

$$\begin{aligned} c_0 &= (1+\sqrt{3})/4\sqrt{2} & c_1 &= (3+\sqrt{3})/4\sqrt{2} \\ c_2 &= (3-\sqrt{3})/4\sqrt{2} & c_3 &= (1-\sqrt{3})/4\sqrt{2} \end{aligned} \quad (4.30)$$

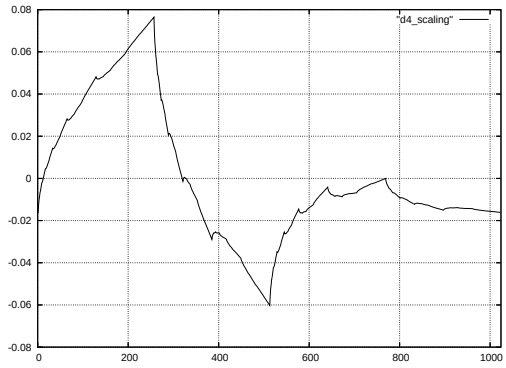
Note that the coefficients noted in 4.23 and 4.24 are only approximations to this analytical solution. There exit also higher tap daubechies filters with smoother wavelet and scaling function shape, but not for all of those it is possible to find a solution in closed form as in this case.

## 4.8 Discrete Wavelet Transform

The application of 4.26 is not a complete forward wavelet transform as it is only the first pass where highest resolution part of the signal is transformed. After calculation of $h_x$ and $l_x$ in 4.26 we keep only low–pass $l_x$ values to which successive wavelet transforms are applied hierarchycally. The $h_x$ values are the final wavelet coefficients and they will not be further transformed. As the signal size is actually halved by wavelet transformation in one resolution level, we stop the processing when the total number of $l_x$ coefficients is less or equal to filter size. This transform is called the **discrete wavelet transform (DWT)**.

The inverse transform is calculated by the reversed algorithm, so that the signal is successively expanded from the lowest to the highest resolution from wavelet coefficients.

a) Scaling function $\phi(x)$.




b) Wavelet $\psi(x)$.

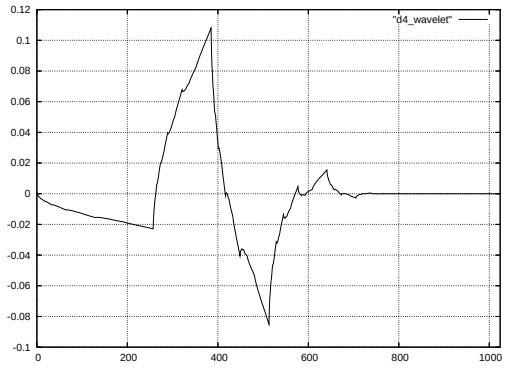



Figure 4.1: Scaling function and wavelet for Daubechies 4–tap decomposition.

## 4.9 Multidimensional Discrete Wavelet Transform

The multidimesional discrete wavelet transform can be done in a similar way as FFT, described in 3.1.6, i.e. to perform one dimensional transform to rows and columns

in case of 2D signal in two separate passes. When this approach is used, the final wavelet spectrum is shown if 4.2b. This kind of calculation is quite expensive and is

a) Dyadic decomposition. b) Non–dyadic decomposition.

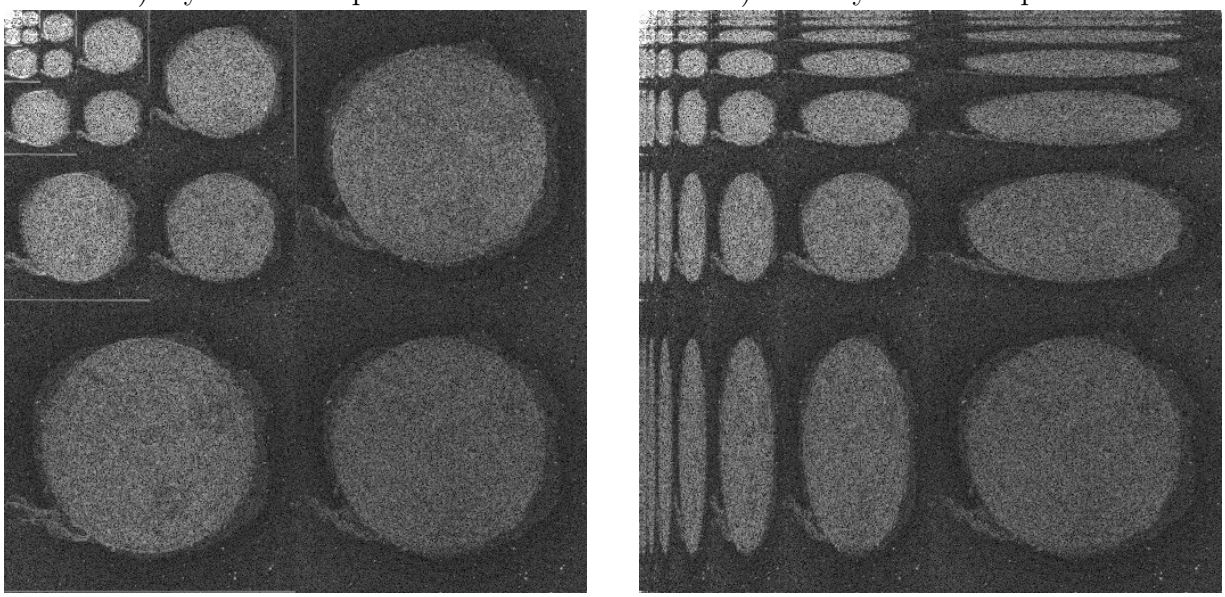


Figure 4.2: Spectrum of sun image for dyadic and non–dyadic wavelet decomposition.

not very suitable for analysis. For this reason the **dyadic** DWT is used frequently, what is calculated in the way that the first pass of DWT is calculated for both the rows and columns and further hierarchies of the transform is only applied to the low–pass filtered subimage. This situation is shown at 4.2a. This approach is faster and more suitable for compression.

## 4.10 Searching an Optimal Orthonormal Wavelet Basis

The filters $h_k$ and $g_k$ must obey certain criteria in order to generate scaling function and wavelet. The most important property is orthogonality of the transformation matrix performing the wavelet decomposition:

$$\mathbf{M}\mathbf{M}^{\mathbf{T}} = \mathbf{1}, \tag{4.31}$$

what is for the 4–tap orthogonal wavelet decomposition fulfilled by 4.28. To find the coefficients of the filters we have to specify yet another conditions that can be for example the requirement that the coefficients have to have two vanishing moments (4.29). But we have freedom specifying these additional conditions so my concern was to figure out how does the other possible wavelet bases look like and how are they suitable for encoding images.

This was done in the way that values of the coefficients were quantized and for all the possible combinations of coefficients in this precision is checked whether an orthogonal matrix can be formed from a particular combination of the coefficients. If so, a testing image is then encoded by the dyadic orthogonal wavelet transform generated from these coefficients. The spectrum in the wavelet domain is then thresholded in the way, that:

$$W(x,y) = \begin{cases} 0 & \text{if } 5000 \leq W(x,y)^2; \\ W(x,y) & \text{otherwise} \end{cases}$$

and then reconstructed. The original and reconstructed image is compared in terms

| $n$ | $h_k$ unnormalized | $h_k$ normalized | zeroed | MSE |
|---|---|---|---|---|
| 1 | $+0+0-7-7$ | $+0.000000+0.000000-0.707107-0.707107$ | 98.35% | 108.37 |
| 2 | $-7+0+0-7$ | $-0.707107+0.000000+0.000000-0.707107$ | 96.48% | 245.55 |
| 3 | $-2+1-3-6$ | $-0.282843+0.141421-0.424264-0.848528$ | 97.28% | 188.25 |
| 4 | $-3+1-2-6$ | $-0.424264+0.141421-0.282843-0.848528$ | 96.88% | 220.09 |
| 5 | $+1-2-6-3$ | $+0.141421-0.282843-0.848528-0.424264$ | 98.57% | 87.63 |
| 6 | $-6-2+1-3$ | $-0.848528-0.282843+0.141421-0.424264$ | 96.89% | 219.41 |
| 7 | $+1-3-6-2$ | $+0.141421-0.424264-0.848528-0.282843$ | 98.49% | 95.79 |
| 8 | $-6-3+1-2$ | $-0.848528-0.424264+0.141421-0.282843$ | 97.33% | 187.85 |
| 9 | $+3+6+2-1$ | $+0.424264+0.848528+0.282843-0.141421$ | 98.57% | 86.44 |
| 10 | $+2+6+3-1$ | $+0.282843+0.848528+0.424264-0.141421$ | 98.49% | 97.17 |
| 11 | $+0-7-7+0$ | $+0.000000-0.707107-0.707107+0.000000$ | 98.39% | 109.62 |
| 12 | $-7-7+0+0$ | $-0.707107-0.707107+0.000000+0.000000$ | 98.40% | 108.75 |

Table 4.1: 4–tap coefficients with 4–bit quantization forming orthogonal wavelet base.

of mean square error (6.13) and as the best transform is considered the one with the

minimal MSE. The table 4.1 shows all low–pass $h_k$ coefficients from which an orthogonal base can be constructed with 4–bit quantization[2] where the "Lena" (fig.6.1) is used as sample image. The "zeroed" column denotes how many spectrum components were set to zero before reconstruction. One can see that the least difference between the original and reconstructed image is made by filter no.9, what is also the closest to the one suggested by Daubechies (4.23). The visual comparison between selected filters can be made in fig.4.3.

a) filter no.9 in table 4.1

b) filter no.6 in table 4.1

Figure 4.3: 4–tap filter image reconstruction comparison after thresholding.

## 4.11 Biorthogonal Wavelet Bases

The orthogonal wavelets (4.6) introduce several drawbacks. The most apparent one is that the coefficients near boundaries in the hierarchycal decomposition are of high magnitudes (see fig.4.2). This is because the fact that the orthogonal wavelet transform is actually calculated by a cyclic convolution. To overcome this and additional issues we introduce **biorthogonal wavelets** by relaxing the orthogonality conditions but not on behalf of lossess reconstructability of the signal.

[2]There were found 12 bases from the total 65536 candidates.

Not two but four filters are introduced by the biorthogonal transform, so we have primal scaling function $\psi$, wavelet $\phi$ and their duals $\widetilde{\psi}$, $\widetilde{\phi}$. Biorthogonality conditions are:

$$\begin{aligned} \widetilde{V}_j &\perp W_j, \\ V_j &\perp \widetilde{W}_j, \\ \langle \widetilde{\phi}_{j,l}, \phi_{j,l'} \rangle &= \delta_{l-l'} \\ \langle \widetilde{\psi}_{j,l}, \psi_{j',l'} \rangle &= \delta_{j-j'} \delta_{l-l'}. \end{aligned} \tag{4.32}$$

The biorthogonal wavelets form Riesz basis:

$$C_1 ||f^2|| \leq \sum_l \lambda_{j,l}^2 + \sum_l \gamma_{j,l}^2 \leq C_2 ||f||^2, \tag{4.33}$$

where $C_1 \leq C_2$ are positive constants. This condition grants that the biorthogonal wavelet basis is still stable. The coefficients $\lambda$ and $\gamma$ are obtained as inner product with dual basis functions:

$$\begin{aligned} \lambda_{j,l} &= \langle f, \widetilde{\phi}_{j,l} \rangle, \\ \gamma_{j,l} &= \langle f, \widetilde{\psi}_{j,l} \rangle. \end{aligned} \tag{4.34}$$

This means that the dual filters $\widetilde{h}_k, \widetilde{g}_k$ are used in the forward (analysis) part of the wavelet decomposition and primal filters $h_k, g_k$ are used for reconstruction (synthesis). Furthermore signal extensions of both the sides of the signal have to be made since the biorthogonal wavelet transform is not based on the cyclic convolution.

## 4.12 Antonini–Daubechies Biorthogonal Wavelets

The Antonini–Daubechies class of biorthogonal wavelets are frequently used in image compression as it decorrelates images better than the ordinary orthogonal wavelets because of a smarter solution of non–periodicity of a general image. This class of discrete wavelet transform is also used in the **JPEG2000** lossy image compression codec.

| $\widetilde{h}_k$ | $\widetilde{g}_k$ | $h_k$ | $g_k$ |
|---|---|---|---|
| +0.037828798579918 | | | +0.037828798579918 |
| −0.023849297515860 | +0.064539050132459 | −0.064539050132459 | +0.023849297515860 |
| −0.110624027489511 | −0.040689752616599 | −0.040689752616599 | −0.110624027489511 |
| +0.377402688109134 | −0.418092440725732 | +0.418092440725732 | −0.377402688109134 |
| +0.852698653219295 | +0.788484872206183 | +0.788484872206183 | +0.852698653219295 |
| +0.377402688109134 | −0.418092440725732 | +0.418092440725732 | −0.377402688109134 |
| −0.110624027489511 | −0.040689752616599 | −0.040689752616599 | −0.110624027489511 |
| −0.023849297515860 | +0.064539050132459 | −0.064539050132459 | +0.023849297515860 |
| +0.037828798579918 | | | +0.037828798579918 |

Table 4.2: Antonini-Daubechies 9/7 tap dual and primal filter coefficients.

In the first phase the dual (analysis) $\widetilde{h}_k, \widetilde{g}_k$ filters are applied and the primal (synthesis) $h_k, g_k$ filters are used for reconstruction. The shape of scaling functions $\phi_k$ for the Antonini–Daubechies biorthogonal analysis is illustrated in fig.4.5. In fig.4.6 is shown how the synthesis scaling function behaves near the signal boundary. This can be done because the signal is extended in the way shown in fig.2.1a on the left side and fig.2.1b on the right side when the analysis filter is applied. In fig.4.4a is

a) Coefficients. b) basis vectors for $16 \times 16$ tiling.

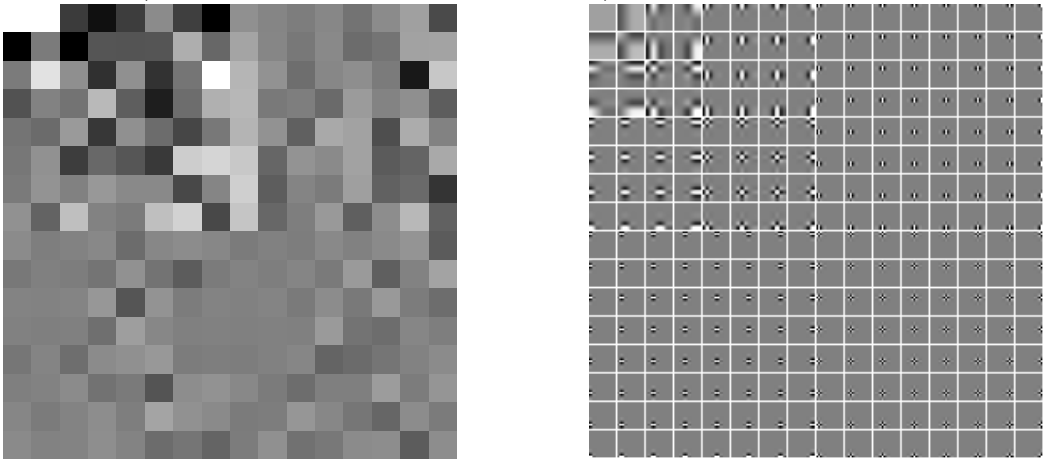


Figure 4.4: Antonini-Daubechies 7/9 tap biorthogonal synthesis filter demonstration.

shown downsampled "Lena" image to $16 \times 16$ pixel resolution in the wavelet domain. Basis vectors of the synthesis transformation are shown in fig.4.4b.

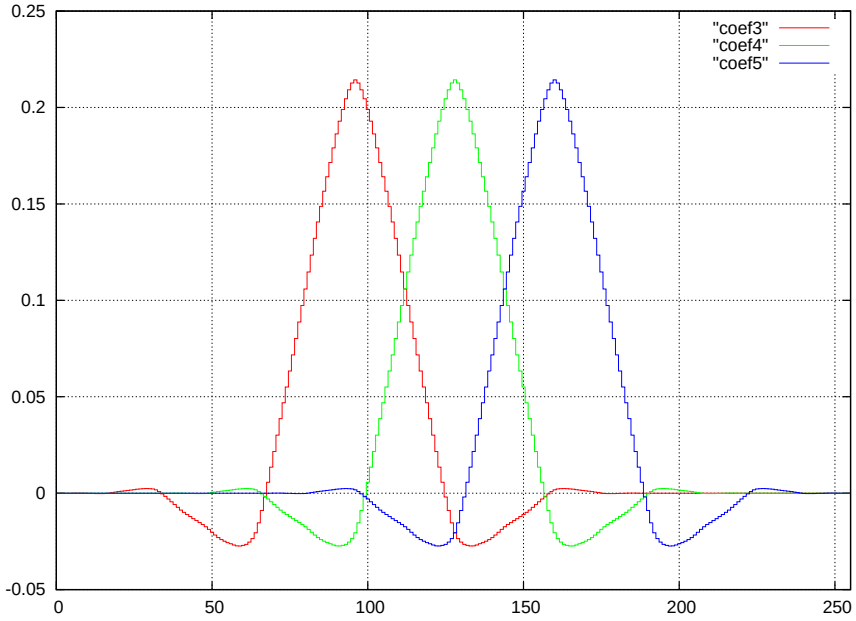



Figure 4.5: Impulse responses for the 3rd, 4th, 5th spectral coefficient.

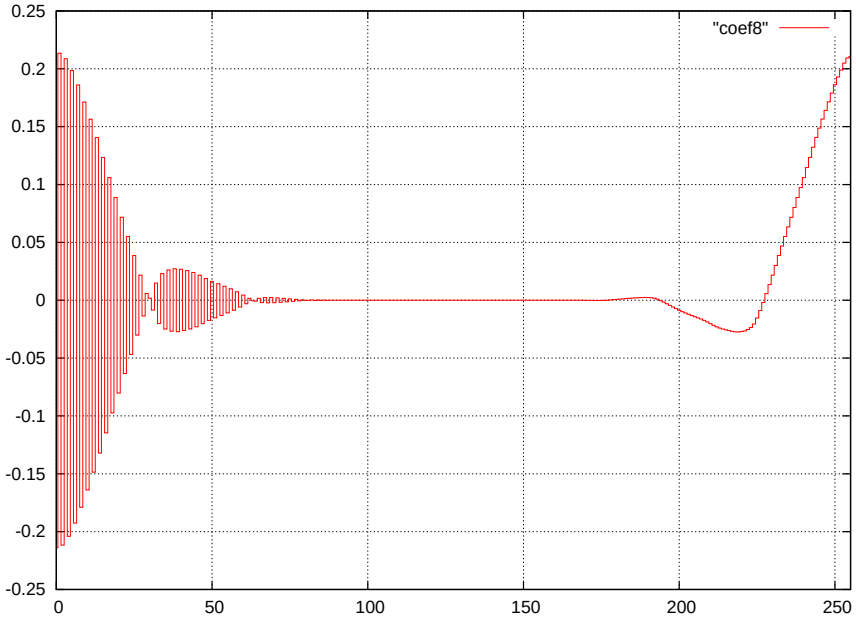



Figure 4.6: Impulse response for the 8th spectral coefficient.

# Chapter 5

# Volume rendering

Analysis of volume data is quite complicated task to solve. There exist several approaches to do such analysis. The basic one is to decompose a 3D object to slices and to analyze them as usual images. This method allows us to look inside the volume of 3D object but seems to be unusable when our demand is to observe the object in a different direction. Then a 3D visualization technique is likely to be used. The basic methods of such 3D visualization are based on extraction of polygonal meshes describing isosurfaces[1]. One can enjoy a 3D impression when observing such isosurface but a disadvantage of it is that the isosurface is in fact two dimensional so one has to extract more such isosurfaces to predict how a property of 3D object changes in volume. Furthermore, algorithms of isosurface extraction are rather complicated and often erroneous for a complex 3D objects. Our approach is to employ a volume rendering technique, which is intended to consider volume information as native 3D. Unfortunately this technique is very computationaly expensive so optimizations and speedup is of our particular importance. The purpose of this chapter is to develop a volume rendering method, apply it to large scale VHP volume dataset and conclude with discussion.

[1] A surface on which a feature of the 3D object is (almost) constant.

## 5.1 Principles of Volumetric Raycasting

In order to develop a suitable technique of volume rendering we must define how the 3D object will actualy be analyzed. First of all we want to analyze a 3D object from various points of view and directions. Therefore a position of an observer and his viewing orintation has to be specified. Then, because of the fact we will display an image of the 3D object on a 2D screen, we have to use some method of projection. Since a human perception of observing a 3D scene is similar to linear perspective projection we will use such projection to let the result look natural to a human observer. Finaly, because we have a volumetric object, it is good idea to have a possibility to "see in depth", i.e. to model translucency of the data. All the noted techniques are discussed in the following sections.

### 5.1.1 Principles of Linear Perspective Projection

The linear perspective projection is defined by a position of observer, orientation of view and his field of view. The principle of creating such projection we can imagine as projections of spatial points into the perspective plane, which is perpendicular to field of view axis. The axis passes through center of the perspective plane, which

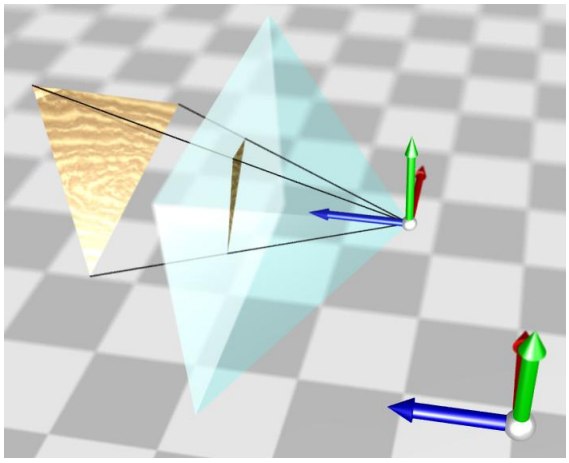


Figure 5.1: Illustration of the linear perspective projection.

is limited by a field of view of the observer. Because of specific shape of connection lines between the observer position and edges of perspective plane it is often said that a linear perspective projection is defined by position and orientation of **perspective pyramid**. In fig. 5.1 is shown the perspective pyramid (blue) and a triangle whose vertices are projected to perspective plane (cyan).

There exist a lot of methods for creating perspective projections on a computer. The most of them can be classified to two groups. The first is based of vertex projections, where all vertices of a 3D object is projected and then connected by polygons. The second are methods based either on raytracing or raycasting techniques, where intersections between ray cast from observer (or from light source) and a 3D object in scenery is calculated. In both the methods we have to employ orthogonal coordinate system transforms in order to convert vertex coordinates to the observer's coordinate system or to define paths of rays to be cast into the scenery. These coordinate transforms are described [6] by the following matrices of basic motions:

$$M_1(x) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad M_2(y) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad M_3(z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5.1}$$

$$M_4(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad M_5(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$M_6(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{5.2}$$

The action of the first three matrices (5.1) is a translation and for (5.2) a rotation around origin. The conversion of a vertex $(x', y', z')^T$ in global coordinate system to

a vertex $(X, Y, Z)^T$ in the observer's coordinate system which is shifted of $(x, y, z)$ and rotated about axes of $(\alpha, \beta, \gamma)$ can be expressed as:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \prod_{i=1}^{6} M_i \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}, \tag{5.3}$$

what is quite computationaly expensive. Since calculation of the transform is very frequent, because moderate complex 3D scenes are composed from $\approx 10^3 - 10^5$ vertices which have to be transformed, a reduction of calculation overhead is welcome. A basic simplification can be made if we separate action of translational and rotational matrices, then the matrix order can be reduced to 3. We can calculate single rotation matrix from first 3 rows and columns extracted from (5.2), denoted $m$:

$$R(\alpha, \beta, \gamma) = \prod_{i=4}^{6} m_i =$$

$$\begin{pmatrix} \cos\beta\cos\gamma & -\cos\beta\sin\gamma & \sin\beta \\ \sin\alpha\sin\beta\cos\gamma + \cos\alpha\sin\gamma & -\sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & -\sin\alpha\cos\beta \\ -\cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & \cos\alpha\sin\beta\sin\gamma + \sin\alpha\cos\gamma & \cos\alpha\cos\beta \end{pmatrix}. \tag{5.4}$$

So the coordinate transformation which is nearly optimal for general rotation angles and translations is

$$\begin{pmatrix} X + x \\ Y + y \\ Z + z \end{pmatrix} = R(\alpha, \beta, \gamma) \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}. \tag{5.5}$$

### 5.1.2 What Raycasting is

The basic idea of the raycasting is to cast a ray through a volume object for every single pixel on the screen and trace properties of the volume object along the ray in order to return a single pixel value per ray to describe volume properties of the

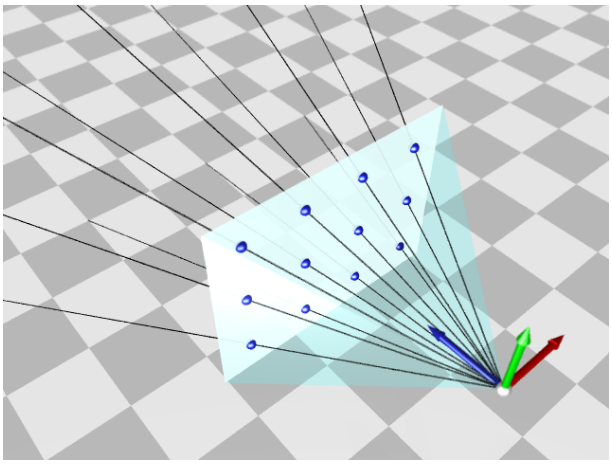


Figure 5.2: Illustration of rays cast from the position of observer.

object. The situation is shown on fig.5.2, where position and orientation of the observer is defined by the perspective pyramid and rays are colored black with the tracing direction away from the observer. Note that in the fig.5.2 is demonstrated creating of raycasted projection to a screen of $4 \times 3$ pixel resolution which is mapped to the perspective plane and rays passes through the center of their corresponding pixels, what is denoted by small blue spheres in the image.

## 5.2 Voxel interpolation

Since a voxel, as a cubical element of 3D scene, is not isotropic, some unnatural artifacts are often introduced in rendered images. To suppress creation of such artifacts, techniques of upsampling are used. These techniques allows "walk" along a ray by sub-voxel steps, what often result in better image.

### 5.2.1 Trilinear Interpolation

For sub-voxel interpolation can be used a well-known linear interpolation in three dimensions, which can be calculated by successive application of usual 1D linear

interpolation:

$$L(y_1, y_2, f) = y_1 + (y_2 - y_1)f, \tag{5.6}$$

where we assume that for $y_1 = f(x_1)$ and $y_2 = f(x_2)$, $x_2 > x_1$ and $x_2 - x_1 = 1$. The constant $f \in \langle 0, 1\rangle$ specifies a position between $x_1$ and $x_2$. Then the trilinear interpolation between eight neighbouring voxels intensities $y_n$ is:

$$\begin{gathered} L_3(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, f_x, f_y, f_z) = \\ L\Big\{ L\Big[L(y_1, y_2, f_x), L(y_3, y_4, f_x), f_y\Big], L\Big[L(y_5, y_6, f_x), L(y_7, y_8, f_x), f_y\Big], f_z \Big\}. \end{gathered} \tag{5.7}$$

### 5.2.2 Tricosine Interpolation

Trilinear intepolation can be enhanced a little in order to smooth discontinuities on voxel boundaries. A way how ensure smooth interpolation[2] is cosine interpolation, which if

$$k(f) = 1 - \cos(f\pi)/2, \tag{5.8}$$

is defined as:

$$C(y_1, y_2, f) = y_1\big[1 - k(f)\big] + y_2 k(f). \tag{5.9}$$

Then the tricosine interpolation is:

$$\begin{gathered} C_3(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, f_x, f_y, f_z) = \\ C\Big\{ C\Big[C(y_1, y_2, f_x), C(y_3, y_4, f_x), f_y\Big], C\Big[C(y_5, y_6, f_x), C(y_7, y_8, f_x), f_y\Big], f_z \Big\}. \end{gathered} \tag{5.10}$$

[2]The same derivatives, equals to zero on a voxel boundary.

### 5.2.3 Tricubic Interpolation

In order to perform even smoother interpolation, the tricubic interpolation can be used. If

$$\begin{aligned} p &= (y_4 - y_3) - (y_1 - y_2), \\ q &= (y_1 - y_2) - p, \\ r &= y_3 - y_1, \\ s &= y_2, \end{aligned} \tag{5.11}$$

the cubic interpolation is:

$$S(y_1, y_2, y_3, y_4, f) = pf^3 + qf^2 + rf + s. \tag{5.12}$$

The tricubic interpolation $S_3$ is defined in the same way as (5.10), but it requires 16 voxel intensities on input.

## 5.3 Volume Renderer Design

The developed volume rendering software saves all the volume data in compressed form in the way that the volume data is decomposed into cubes[3] so that only this part of volume is needed to be stored in memory when a ray passes through it. Frequently used cubes are stored in memory in order to prevent a need to decompress them when a ray penetrates to another cube that was traced recently. The software allows to use only a memory specified by an user. To keep only last recently used cubes the LRU [10] algorithm is used. The software uses progressive rendering in the way that at first only a fraction of total rays is cast into a 3D scenery. After calculation of resultant colors, these are interpolated for pixels for which no rays have been cast and shown on the screen. This allows an user to see a temporary result of rendering even if rendering goes on. In fig.5.3a is demonstrated a projection from the volume renderer made in the full resolution and 5.3b shows a projection which calculation

[3] of $8 \times 8 \times 8$ up to $32 \times 32 \times 32$ voxels each.

was 16–times faster as only 1/16 of total rays was needed to be cast into the scenery. The pixel values are calculated in a way that pixel intensities are gathered along a ray in a given[4] precision so that the result is a vector of intesities $\vec{V_c}$ of arbitrary size for each channel $c$ in the volume dataset. The $\vec{V_c}$ can be processed in various ways to obtain a single pixel value to be displayed on the screen. At first a convolution with thin gaussian TIIR kernel is calculated to suppress sharp intensity jumps and similar artifacts caused by discontinuities between voxels:

$$\sum_{k=-K/2}^{K/2-1} \vec{V_c}(i-k)h(k), \qquad \forall i, c \tag{5.13}$$

then opacity tracing algorithm is used. Description of such techniques can be found in [8], [4]. The opacity tracing algorithm works in the way that a certain small amount of the first intensities along the ray are ignored in order to skip low–intensity layer around a volume object. Then the final pixel intensity of $c$–th channel of the volume data is calculated as

$$I_c = \frac{1}{N_2 - N_1} \sum_{i=N_1}^{N_2} T^{i-N_1} \vec{V_c}(i), \qquad \forall c \tag{5.14}$$

where $T$ is translucency coefficient[5], $N_1$ is the index in $\vec{V_c}$ for the first not ignored intensity value, $N_2$ is an index in $\vec{V_c}$, where the contribution to the final pixel intesity is neglectable due to small value of $T^{i-N_1}$ in 5.14. Then the final color $\vec{P}$ in RGB space of the pixel displayed on screen is calculated like

$$\vec{P} = \left\| \sum_{c=0}^{C} \vec{P_c} I_c \right\|, \tag{5.15}$$

where $C$ is the cotal number of channels in volume data, $P_c$ is normalized base color of the channel in RGB color space. Normalization in 5.15 denotes normalization and/or saturation of the RGB components in $\vec{P}$.

[4] mostly sub–voxel

[5] $T \in (0, 1\rangle$, where for $T = 1$ is the object completely translucent

a) full resolution

b) calculated from 1/16 of all rays

Figure 5.3: Progressive rendering demonstration.

a) uncorrected slices

b) aligned by phase correlation

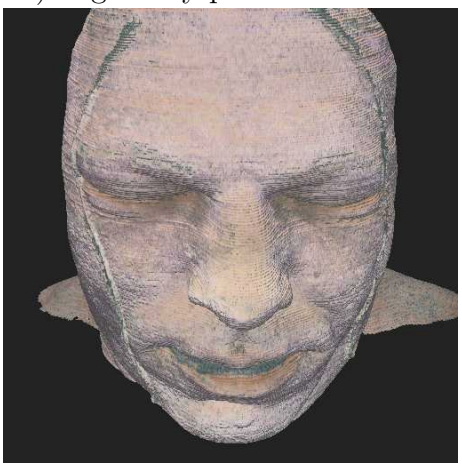


Figure 5.4: Demonstation of phase correlation shift detection in volume rendering.

# Chapter 6

# Lossy Image Compression

## 6.1 Karhunen-Loève Transform

The Karhunen-Loève transform, also known as principal orthogonal decomposion or Hotelling transform, creates optimal basis to convert correlated input random vectors to its decorrelated form. Its application is very straightforward in image compression as pixel values in a natural picture such as a photography are highly correlated.

### 6.1.1 Principles

The basic idea is to decompose an input image to statistically significant number $F$ of smaller not overlapping fragments and consider them random vectors of $V \in \{v_0, v_1, ..., v_{F-1}\}$ of $N$ samples each. Then elements of $N \times N$ autocovariance matrix $\mathbf{C_V}$ is calculated as:

$$\begin{aligned} \mathbf{C_V}(x,y) &= \frac{1}{F}\sum_{f=0}^{F-1}\Big[v_f(x) - \overline{v}(x)\Big]\Big[v_f(y) - \overline{v}(y)\Big] = \\ &= \frac{1}{F}\sum_{f=0}^{F-1} v_f(x)v_f(y) - \overline{v}(x)\overline{v}(y), \end{aligned} \tag{6.1}$$

where $\overline{v}(n),\ n \in \{0, 1, ..., N-1\}$ is mean value of $v_f(n),\ \forall f \in \{0, 1, ..., F-1\}$. The calculated $\mathbf{C_V}$ is always symmetric. Thus an orthogonal matrix $\mathbf{K}$ can be always found to diagonalize $\mathbf{C_V}$. So for $\mathbf{K}$ the orthogonality condition:

$$\mathbf{K}\mathbf{K}^\mathbf{T} = \mathbf{K}^\mathbf{T}\mathbf{K} = \mathbf{1}, \tag{6.2}$$

where $\mathbf{1}$ is the identity matrix, must hold. If we denote eigenvalues and normalized column eigenvectors of $\mathbf{C_V}$ as $\lambda_i$ and $e_i$, $i \in \{0, 1, ..., N-1\}$, then we can arrange all the eigenvectors $e_i$ to form the matrix $\mathbf{K}$. If we sort the eigenvectors in $\mathbf{K}$ in the way that the eigenvector corresponding to the largest eigenvalue is in the first column and the other eigenvectors are sorted with decreasing eigenvalues, then

$$k_f = \mathbf{K}^\mathbf{T} v_f \tag{6.3}$$

is called Karhunen-Loéve transform (KLT), which converts $v_f$ to decorrelated vector $k_f$. The decorelation property of KLT can be shown as diagonalization of $\mathbf{C_V}$:

$$\mathbf{K}^\mathbf{T}\mathbf{C_V}\mathbf{K} = \begin{pmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_F \end{pmatrix}, \tag{6.4}$$

where we see the output matrix is diagonal and contains only eigenvalues (or degrees of freedom) of selected image fragment collection. The inverse transformation can be done simply exploiting the orthogonal property (6.2) of $\mathbf{K}$ as:

$$v_f = \mathbf{K} k_f. \tag{6.5}$$

### 6.1.2 Eigenvectors and Eigenvalues Calculation

The most difficult problem in calculation of KLT is calculation of eigenvectors of autocovariance matrix $\mathbf{C_V}$. There exist various iterative methods like Jacobi transformations of symmetric matrix or two-pass method which uses Givens or Householders

a) original image

b) optimal $8 \times 8$ KLT basis vectors

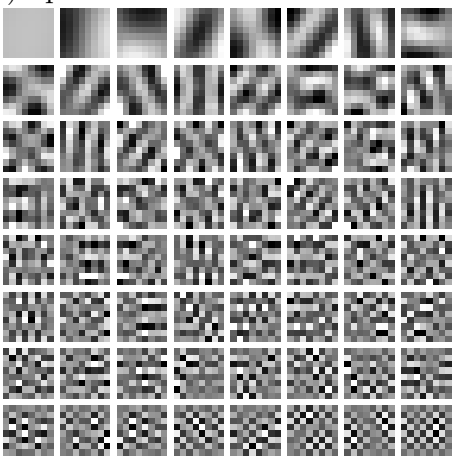


Figure 6.1: Karhunen-Loéve $8 \times 8$ transform demonstration.

reduction to tridiagonal form and successive application of either QR or QL algorithm. Unfortunately these methods are beyond the scope of this dissertation and their proper description can be found in [11].

## 6.2 Discrete Cosine Transform

Another method of image decorrelation is the Discrete Cosine Transform (DCT). In numerous cases DCT is used frequently in practice since there exist lots of fast algorithms which calculates DCT decorrelation much faster with results close to the KLT. Many image compression algorithms are based on DCT, for instance JPEG [16]. There exist various types of DCT which are designed to fast image decorrelation in separate image fragments or by partially overlapped fragments[1]. The most frequent DCT type is DCT-II, which is in 1D defined as

$$F(k) = C(k) \sum_{x=0}^{X-1} f(x) \cos \frac{k(2x+1)\pi}{2X}, \tag{6.6}$$

[1]So called Lapped Orthogonal Transform (LOT).

where $X$ is total number of samples in $f(x)$, and

$$C(k) = \begin{cases} \sqrt{\frac{2}{N}} & \text{if } k \neq 0; \\ \sqrt{\frac{1}{N}} & \text{otherwise.} \end{cases} \tag{6.7}$$

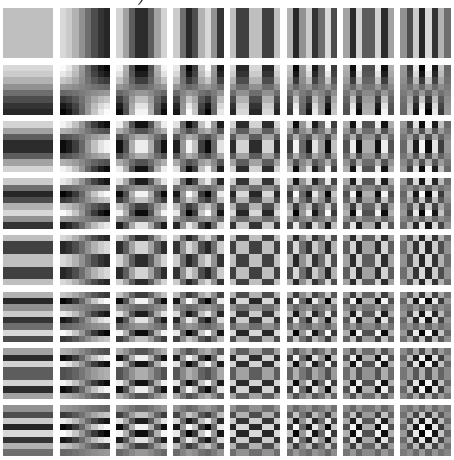
In our case we will use 2D DCT-II, where we can also compose it from row and column 1D transforms in the same fashion as in 3.1.6. Then we can write the 2D DCT-II as

$$F(k_x, k_y) = C(k_x)C(k_y) \sum_{y=0}^{Y-1} \left[ \sum_{x=0}^{X-1} f(x, y) \cos \frac{k_x(2x+1)\pi}{2X} \right] \cos \frac{k_y(2y+1)\pi}{2Y}. \tag{6.8}$$

As we have mentioned, the DCT transform is used because its calculation speed. It is so because close similarity of DCT basis vectors with their optimal ones for a natural picture such as a photography shown at fig.6.1a. To see the similarity, compare the optimal basis vectors calculated by KLT in the fig.6.1b with fig.6.2b. These discrete bases are calculated as a spatial response to impulse in KLT[2] or DCT[3] domain. We can interpret the shape of basis vectors shown at fig.6.2a as an information of what type of detail from an input image contains a single DCT-II basis coefficient. In particular, we can say that the most of horizontal information contain the leftmost coefficients, the highest details the coefficient in bottom right, etc.

[2]Fig.6.1b.
[3]Fig.6.2.

a) natural order



b) variance descending (zig-zag) order

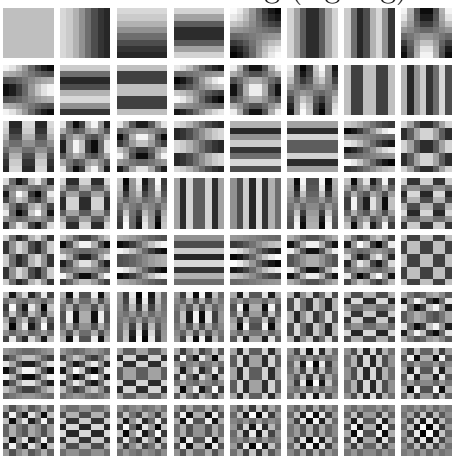


Figure 6.2: DCT basis vectors for $8 \times 8$ transform.

## 6.3 Discrete Walsh-Hadamard Transform

Another method of KLT bases approximation is the discrete Walsh-Hadamard transform (DWHT). It is one of the simplest transformations used in image decorrelation. It uses even simplier shape of basis vectors:

$$F(k) = \sqrt{\frac{1}{X}} \sum_{x=0}^{X-1} f(x) \text{sign}\Big[\cos \frac{k(2x+1)\pi}{2X}\Big], \tag{6.9}$$

which makes this transform extremely simple to calculate because the sign function is defined as

$$\text{sign}(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \end{cases} \tag{6.10}$$

Hence the DWHT transformation matrix consist only from $-1$ or $+1$ elements multiplied by a scaling coefficient. For instance, the 4 point 1D DWHT matrix is:

$$\mathbf{K} = \sqrt{\frac{1}{4}} \begin{pmatrix} +1 & +1 & +1 & +1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \\ +1 & -1 & +1 & -1 \end{pmatrix} \tag{6.11}$$

a) natural order

b) variance descending (zig-zag) order

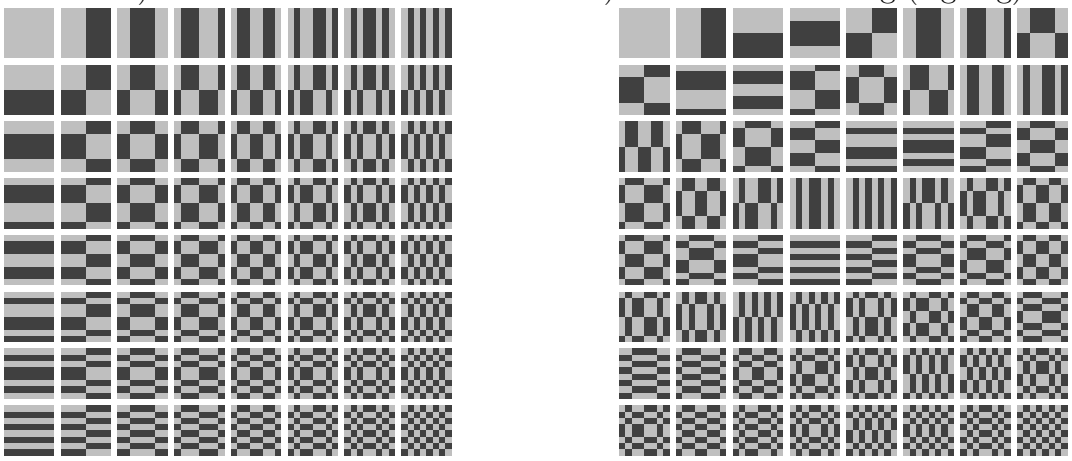


Figure 6.3: DWHT basis vectors for $8 \times 8$ transform.

## 6.4 DCT–based Lossy Compression Algorithm Proposal

The purpose of all compression algorithms in image compression is to reduce the information capacity up to its minimum in the fastest possible time at minimal memory requirements. Furthermore, the quality loss of decompressed image should be minimal. Unfortunately, compromises have to be made because these optimum criteria are dependent on each other. In particular, the most of recent lossy image compression algorithms are based on the DCT (section 6.2) even if it is not optimal, because the calculation time is much shorter in comparison with KLT (section 6.1).

The preferred scheme for the compression algorithm is progressive. It means that an original image $O$ to be compressed is stored at various levels of detail and only differences are encoded. The progressive scheme was chosen because of error resistence and possibility of displaying preview at various scales of image even from its small loaded context. The encoding itself is based on 2D DCT but is designed to be easily modified to KLT. The original image $O$ is decimated to a size to fit one single 2D DCT fragment and saved to output file[4]. Then the fragment is upsampled to the original

[4]It means quantized, zig-zag ordered, RLE and entropy encoded.

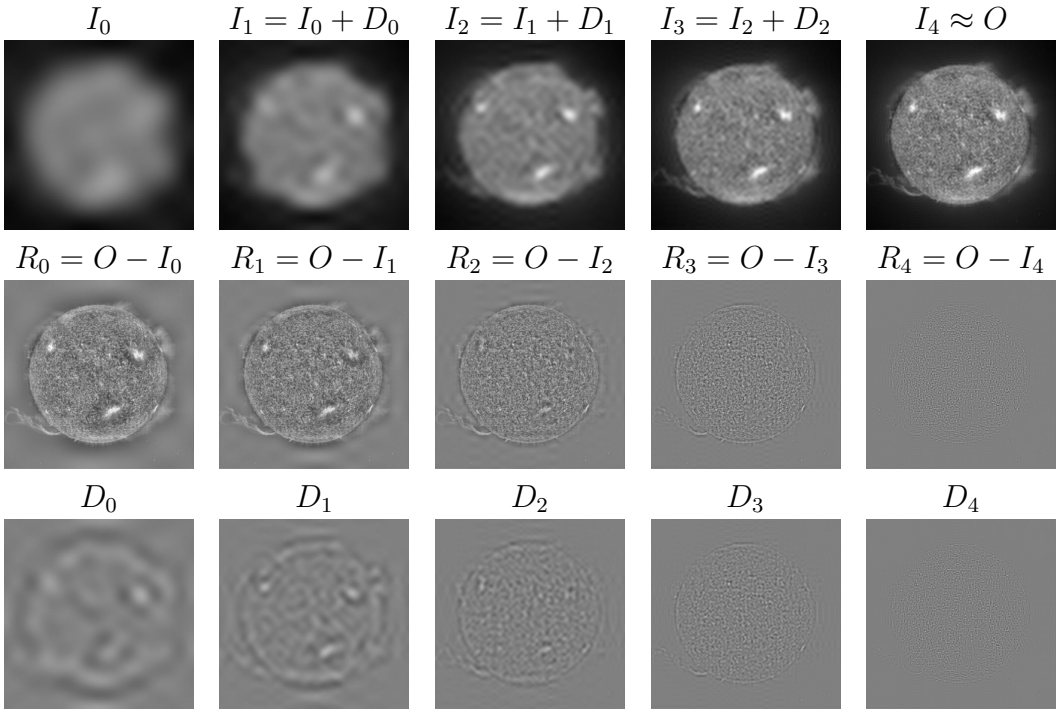



Figure 6.4: Progressive $8 \times 8$ encoding with zero-padded 2D IDCT prediction.

size of $O$ with suitable technique of interpolation. After it the upsampled image $I_0$ (see fig.6.4 where the fragment size is $8 \times 8$ pixels) is subtracted from $O$ which yields $R_0$. That was the first step and the following steps are calculated iteratively in the way that the number of DCT fragments is increased by factor of $4^5$ per iteration what ensures convergence to the original image:

$$\begin{aligned} D_i &= \uparrow \mathrm{DCT}\Big(\downarrow R_i\Big), \\ I_{i+1} &= I_i + D_i, \\ R_{i+1} &= O - I_{i+1}, \\ i &= i + 1, \end{aligned} \tag{6.12}$$

where $\downarrow$ denotes decimation and $\uparrow$ denotes interpolation. After each iteration $D_i$ is encoded and saved to output file. The total number of iterations is equal to $b = \log_2 S$,

[5] 2 per dimension

where $S$ is the largest dimension of image, where $S \leq 2^b$. This method gives better or comparable compression in comparison with recent lossy image compression formats with comparable signal–to–noise ratio.

### 6.4.1 Image Interpolation

The most important part to achieve the best possible compression in the progressive encoding scheme is an interpolation algorithm which is used to predict pixels beyond resolution of decimated original image. As the best interpolation algorithm which gives the best prediction we consider an algorithm whose predictions makes the smallest difference between the original image and interpolated image during the encoding iteration. To measure such differences there are used various metrices. We will use mean square error (MSE) metric, defined as

$$\mathrm{MSE}(I_i) = \frac{1}{XY} \sum_{y=0}^{Y-1} \sum_{x=0}^{X-1} \Big[ O(x,y) - I_i(x,y) \Big]^2, \tag{6.13}$$

where $O$ is the original image and $I_i$ is the interpolated image in the $i$-th iteration. As a sample image is used photography of HeLa cancer cells[6], fig.6.5a.

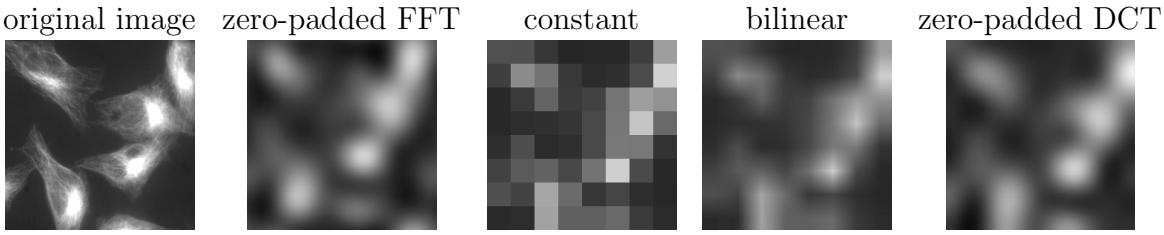



Figure 6.5: Image interpolation from decimated image $8 \times 8$ to $256 \times 256$ pixels.

From the fig.6.6 can be seen that from the usual interpolation methods the DCT based interpolation gives the best prediction. This is the reason why this method is used for interpolation in the compression algorithm.

[6]Measured by Mgr.Pavlína Bečvářová from Dept. of Biophysics, Medical Faculty, Masaryk University in Brno.

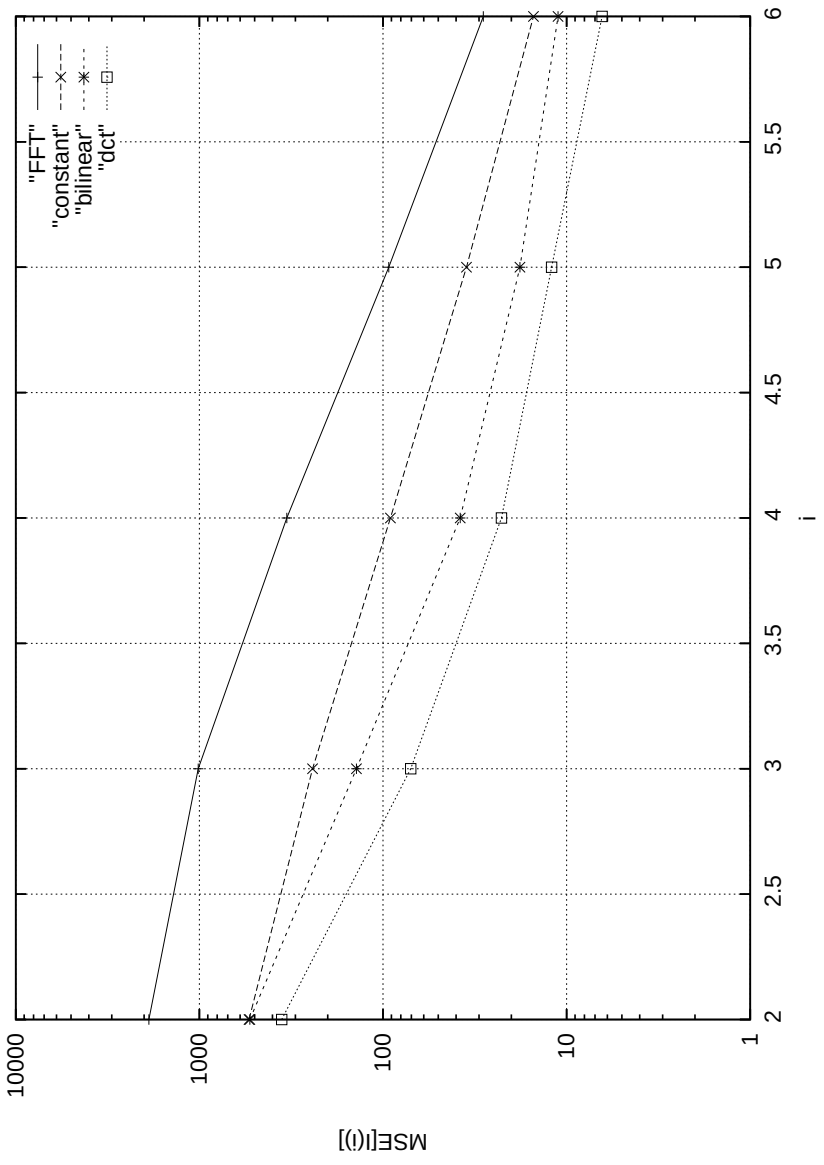



Figure 6.6: Pixel predictions from decimated images with increasing iteration.

# Chapter 7

# Lossless Image Compression

A different treatment to image compression has to be made in order to remove redundant information from input data with a request of lossless reconstruction of the original image. There exist a few approaches to lossless compression algorithms. The simplest approaches are RLE algorithms that replaces large sequences of identical characters with length-code pairs. The other classical methods used frequently are alphabet substitution approaches such as Shannon-Fano coding or Huffman [7] coding. These methods construct binary trees in order to assign the smallest bit sized code to the most probably used characters. To the other group belong dictionary or sequentional based methods from a wide Lempel-Ziv family such as LZ77 [17] or LZ78 [18]. These methods uses dynamically updated databases of used sequences, where a compression is achieved because a single code is sent out for every frequently used sequence. In our approach we use multiple level compression, where arithmetic coding scheme is used at the end, which achieves the best compression in general from all the mentioned methods.

The arithmetic coding is a bit similar to Huffman coding in the way that it assigns less bit-sized codes to the most probable characters. The principal difference is that in case of arithmetic coding the codes need not be of integer bit length. In fact the output of arithmetic coder is one huge binary number which represents all characters present in message of any finite alphabet.

## 7.1 Run–Length Encoding Techniques

The **Run–length encoding** (RLE) is frequently used as the first pass compression to reduce length of highly redundant message before sequential or entropy coders are applied in the following passes. It is highly recommended to apply further compression to RLE encoded data as the RLE itself comes with rather poor compression in general. The purpose of RLE is that long runs of same characters are badly encoded by entropy coders as the long runs significantly affect probability distribution of characters in the message what degrades entropy coder performance. The compression performance of sequential coders is affected as well. To avoid these problems we replace an information about long runs by a code that represents length of a run. So that RLE encoded message should no longer contain long runs.

### 7.1.1 Basic RLE Scheme

The most straightforward implementation of the RLE codec is likely to replace each character of an input message by a pair of code telling us how many times is the following character repeated and the character itself. So the RLE encoding of a sample message looks like this:

$$M = \text{“}\texttt{abaaaaaacdaaaaca}\text{”} \quad \Longrightarrow \quad M_{\text{RLE}} = \text{“}\texttt{1a1b6a1c1d4a1c1a}\text{”}.$$

We can see this approach is rather wasteful as the encoded message could be twice as large as original in the worst case. Even if our sample message $M$ has two runs, the output message has exactly the same number of characters as the original so the idea needs to be slighly improved.

### 7.1.2 Improved RLE Scheme

To improve a compression of RLE we have to think either from the coder and decoder point of view. The biggest waste in the previous approach was caused by the requirement that each character is transformed to a pair. We can output the pair

only when needed. The problem is to realize how decoder will learn that it should expect a character or a code specifying the run length. A simple assumption can be made to resolve this. It is likely that we should expect a run in decoder if incoming character is the same as the former one. So that we can replace all at least 2–character sequences by the two characters and a number representing a run length. So this is how it looks for a given message:

$$M = \text{“}\texttt{abaaaaaacdaaaaca}\text{”} \quad \Longrightarrow \quad M_{\mathrm{RLE}} = \text{“}\texttt{abaa4cdaa2ca}\text{”} \tag{7.1}$$

We see the message was shortened by this improvement but this design of RLE also introduces an apparent drawback. In case of a message composed exclusively from equal character pairs the message will be larger of about 1/3 compared to its original length.

### 7.1.3 Switched RLE Encoding

The idea from the previous section can be modified by a consideration that code specifying a run length could have two meanings depending on whether the order of the code is even or odd. In the even case we assume that the length code specifies length of non-run sequence and run length in the even case. This approach is rather effective because character runs are bounded by non-run data at both the ends in most cases. So the RLE of a given message is:

$$M = \text{“}\texttt{abaaaaaacdaaaaca}\text{”} \quad \Longrightarrow \quad M_{\mathrm{RLE}} = \text{“}\texttt{2ab6a2cd4a2ca}\text{”} \tag{7.2}$$

Coder can output zero even length code followed by non–zero odd length code when a run is followed by another run of characters instantly. Because one character within a message has finite size, say one byte, the length of either a run or non-run data is limited to 255. When a very long sequence of non-run data is found, zero odd code and non–zero even code can be sent to output. So in the worst case an output message from this coder will be larger of about 1/128 when the original message contain no runs.

### 7.1.4 Hierarchycal Bitmap–based RLE Encoding

The most effective approach for the RLE encoding is the hierarchycal bitmap encoding. The basic idea is to create a bitmap where is one bit reserved for each character in a message. We set a corresponding bit to `1` if a character belongs to a run, `0` otherwise. Note that the first character of a run has the first bit set to `0` to let the decoder know what character is repeated. This is illustrated on bottom of table 7.1. We can reconstruct the original message from a knowledge of $B$ and all characters

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| hierarchy $B_1$ (1) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| hierarchy $B_2$ (0) | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| hierarchy $B_3$ (1) | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bitmap $B$ | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| message $M$ | a | b | a | a | a | a | a | a | c | d | a | a | a | a | c | a |

Table 7.1: Illustration of hierarchycal RLE encoding.

where bits are set to `0`. The size of $B$ can be significantly decreased by hierarchycal encoding as there are many of bit runs present in it. In upper part of table 7.1 is shown the hierarchycal decomposition, which is done in the way that the original $B$ is downsampled by a factor of two to $B_3$, so that a bit in $B_3$ assigned to two bits in $B$ is set to `1` when both the $B$ bits are set to `1`, zero otherwise. This is done because if we know $B_3$, we need to save only bits from $B$ that have their respective bit in $B_3$ set to zero. It is likely that zero bits will prevail in $B_3$ after this transformation, so that we can use the similar decomposition of $B_3$ to $B_2$ but for zeros in $B_3$. We can go on subsampling the higher hierarchies up to the decomposition to lower hierarchies still decreases the total bits for representing $B$. In our case we stop after $B_2$. Then the output of coder is a bit sequence from the lowest hierarchy up to $B_3$ and all characters for which $B_3$=`0`:

$$M = \text{“}\texttt{abaaaaaacdaaaaca}\text{”} \quad \Longrightarrow \quad M_{\text{RLE}} = \text{“}\texttt{(10101110) abaacdaaca}\text{”} \qquad (7.3)$$

Note that the information about runs in $M$ is encoded only to 8–bits, what is usually a size of one character within the message. The bits written to output are shown in

brackets in 7.3. The results achieved by this method depends highly on basic heuristic what decides whether it is better to suppress `0`'s or `1`'s from a subsequent hierarchy. Also different decimation by factor of 4 or 8 can be used in the decomposition in order to improve compression.

## 7.2 Burrows–Wheeler Transform

This method based on block–sorting was first invented by D.J. Wheeler in 1983, but was first published in 1994 [1]. The principle of the Burrows–Wheeler transform (BWT) is that a block of data to be transformed is sorted lexicographically to put similar contexts present within a data block together to be better encoded by entropy coders. Thus BWT is not a compression method itself but leads to better performance for further applied compression. The essential property of BWT is that the data can be reconstructed without any losses to its original form.

### 7.2.1 Forward Burrows–Wheeler Transform

To demonstrate the forward BWT itself we use encoding of a message $M =$"`abaca`". First, one has to do a number of string rotations that equals the total characters within the message (table 7.2a). The $I_f$ is a position of the first character of the rotated

a) rotations of the input message

| $I_f$ | rotated $M$ |
|---|---|
| 0 | `abaca` |
| 1 | `bacaa` |
| 2 | `acaab` |
| 3 | `caaba` |
| 4 | `aabac` |

b) sorted $M$ and BWT output

| $I$ | $I_f$ | rotated, sorted $M$ | output $M_{\mathrm{BWT}}$ |
|---|---|---|---|
| 0 | 4 | `aabac` | `c` |
| 1 | 0 | `abaca` | `a` |
| 2 | 2 | `acaab` | `b` |
| 3 | 1 | `bacaa` | `a` |
| 4 | 3 | `caaba` | `a` |

Table 7.2: Illustration of the forward Burrows–Wheeler transform.

message in the original string $M$. Then the rotated strings are lexicographically sorted (table 7.2a) in the way that we keep an information of the original index $I$

in $I_f$ (table 7.2b). The output is then the last characters from the sorted strings followed by the index $I$ where $I_f = 1$. So the output is $M_{\text{BWT}}$ ="`cabaa`" and index $I_1 = 3$.

### 7.2.2 Backward Burrows–Wheeler Transform

The original message can be reconstructed from the input sequence $M_{\text{BWT}}$ ="`cabaa`", and index $I_1 = 3$ because of properties of lexicographycal sorting. We are able to reconstruct the original message if we assign an index $I$ to each character of the message and perform lexicographical sorting in the way that we store the sorted indices to $I_b$. Then the reconstruction can be done by algorithm described in 7.3b and the interpreation of it is shown in table 7.3a.

a) backward BWT demonstration

| $I$ | | $M_{\text{BWT}}$ | | $I_b$ | sorted $M_{\text{BWT}}$ |
|---|---|---|---|---|---|
| 0 | $\overset{4}{\Longrightarrow}$ | `c` | $\overset{4}{\Longrightarrow}$ | 1 | `a` |
| 1 | $\overset{5}{\Longrightarrow}$ | `a` | $\overset{5}{\Longrightarrow}$ | 3 | `a` |
| 2 | $\overset{2}{\Longrightarrow}$ | `b` | $\overset{2}{\Longrightarrow}$ | 4 | `a` |
| 3 | $\overset{1}{\Longrightarrow}$ | `a` | $\overset{1}{\Longrightarrow}$ | 2 | `b` |
| 4 | $\overset{3}{\Longrightarrow}$ | `a` | $\overset{3}{\Longrightarrow}$ | 0 | `c` |

b) algorithm of backward BWT

1 $i = I_1$
2 $M_{\text{BWT}}(i) \rightarrow$ output
3 $i = I_b(i)$
4 if $i \neq I_0$ go to 2

Table 7.3: Illustration of the backward Burrows–Wheeler transform.

### 7.2.3 Why BWT Transformed Data Compresses Better

To understand this fact we have to define what a **context** is. A context of $n$-th order in a given message is a selection of $n + 1$ characters from the message where order of the characters is conserved. If we consider message "`data`" then all possible contexts[1] are "`d`", "`a`", "`t`", "`da`", "`at`", "`ta`", "`dat`", "`ata`". The high compressability of BWT transformed data can be then illustrated in the way that BWT actually gathers characters from a message with the same or similar contexts what decreases the local

[1] Of the first, second and third order.

entropy of a non–random message as there is a high probability that in a sufficiently long message the characters with similar contexts will be same or less different in comparison with the original message. This leads to significantly better performance of entropy coders.

## 7.3 Concept of Entropy (Redundancy Removal) Coders

In order to understand methods of lossless compression it is needed to define a term of entropy $H$ used in field of information theory. The Shannon's theorem [13] is frequently used for it so it is presented here.

It seems quite natural that there exist some limiting rate of lossless compression achievable by alphabet substitutions. For instance, let us have a message $M = (M_0, M_1, M_2, ...)$ defined by an alphabet with finitely many characters $M_j \in a = \{a_0, a_2, a_3, ..., a_{n-1}\}, \forall j \in \{0, 1, 2, ..., N-1\}$. Coding of one character apparently requires $\log_2 n$ bits so for all the characters in $M$ is required $N \log_2 n$ bits to express it without losses. We can denote the number of bits required to express the first $N$ characters as $B(a, N)$, then we can write that

$$\lim_{N\to\infty} \frac{B(a, N)}{N} = \log_2 n. \tag{7.4}$$

Shannon's theorem asserts that there exists a nonnegative number $H$, the entropy of probability distribution of the original alphabet, such that a new alphabet $b = \{b_0, b_2, b_3, ..., b_{n-1}\}$ consisting of variable length characters satisfies:

$$H \leq \lim_{N\to\infty} \frac{B(b, N)}{N}, \tag{7.5}$$

and for every $\epsilon > 0$ there exists a particular alphabet $b^\epsilon$ which satisfies:

$$\lim_{N\to\infty} \frac{B(b^\epsilon, N)}{N} \leq H + \epsilon. \tag{7.6}$$

Let us assume that the characters in $M$ occurs with probability $P(a_i) = p_i$, then the entropy of message is:

$$H = -\sum_{i=0}^{n-1} p_i \log_2 p_i. \tag{7.7}$$

The entropy $H$ actually says that the average bit-size of code in an optimal alphabet to encode the message $M$ is $H$. It should be apparent from the inequality:

$$0 \leq H \leq \log_2 n, \tag{7.8}$$

which can be interpreted in a way that the entropy of message is minimal, $H = 0$, in case the input message consists of the same characters repeated forever, or that the entropy is maximal, $H = \log_2 n$, if all the characters in the message are equally probable. Hence a possible measure of message redundancy could be a value $R = \log_2(n) - H$.

## 7.4 Approaches to Entropy Coding

There exist various approaches to the problem of prediction of unknown incoming characters in an encoded message. They are based on statistical modelling and can be classified to three groups proportionaly to their performance and increasing calculation expense:

- **static coding** assumes that an input message has a fixed probability distribution without any analysis of the message. This probability is not being changed (is static) during encoding. This method reaches the worst compression ratios.
- **semi-adaptive coding** is a two pass method, where in the first pass probability distribution of the message is gathered, in the second pass the message is encoded with respect to this distribution. It is needed to save the gathered probability distribution in order to decode the message.
- **adaptive coding** changes the prediction model while the message is being encoded in the way that the model is updated after encoding of each character.

This is the most efficient approach in terms of compression, but also the most computationally expensive.

## 7.5 Shannon–Fano Coding

The Shannon–Fano coding is based on a binary tree tracing, where fixed bit sized codes are emitted by encoder based on the position of a particular character within the tree. It specifies how to construct such a tree from a known statistics of characters occurence from a message in order to emit nearly optimal bit–sized codes. It constructs the binary tree in a way that it splits the incoming set of characters into two halves with approximatelly 0.5 probability of occurence each. A tree node is created

a) message statistics and codes

| character | $n$ | $P$ | code | total bits |
|---|---|---|---|---|
| `1` | 6 | 0.3750 | `00` | 12 |
| `a` | 4 | 0.2500 | `10` | 8 |
| `b` | 1 | 0.0625 | `1100` | 4 |
| `6` | 1 | 0.0625 | `1101` | 4 |
| `c` | 2 | 0.1250 | `01` | 4 |
| `d` | 1 | 0.0625 | `1110` | 4 |
| `4` | 1 | 0.0625 | `1111` | 4 |
| $\sum$ | 16 | 1.0000 | n/a | 40 |

b) Shannon–Fano tree

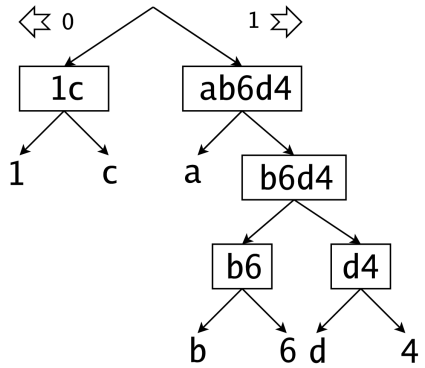



Table 7.4: Shannon–Fano coding demonstration of message "`1a1b6a1c1d4a1c1a`".

above such a split. The two sets are lately decomposed in the same way up to only one character remains. Then the creation of binary tree is completed. The construction of bit codes assigned to each character is done in the way that we define that we move from top to left in the tree when '`0`' and to right when '`1`' is emitted. This design assigns less sized codes to the most probable characters in the message and wider for less probable ones. This is why it achieves compression, because the most of messages have significantly different distribution than uniform. We can see from the

table 7.5 that the example message was encoded to 40 bits (5 bytes), but distribution of the characters must be known to the decoder to reconstruct the message in order to let it create the decoding tree.

## 7.6 Huffman Coding

Huffman coding is very similar to Shannon–Fano coding in the way that the output from the encoder are integer–sized bit sequences for each character to be encoded. The principal difference is that it uses a different method of binary tree construction. At the beginning it sorts all characters in order of occurence. Then it merges two characters of the smallest occurence what forms a Huffman tree node and a new character. In the next passes the process is repeated up to the top of the tree, i.e.

a) message statistics and codes

| character | $n$ | $P$ | code | total bits |
|---|---|---|---|---|
| `1` | 6 | 0.3750 | `1` | 6 |
| `a` | 4 | 0.2500 | `01` | 8 |
| `b` | 1 | 0.0625 | `00010` | 5 |
| `6` | 1 | 0.0625 | `00011` | 5 |
| `c` | 2 | 0.1250 | `001` | 6 |
| `d` | 1 | 0.0625 | `00000` | 5 |
| `4` | 1 | 0.0625 | `00001` | 5 |
| $\sum$ | 16 | 1.0000 | n/a | 40 |

b) Huffman tree

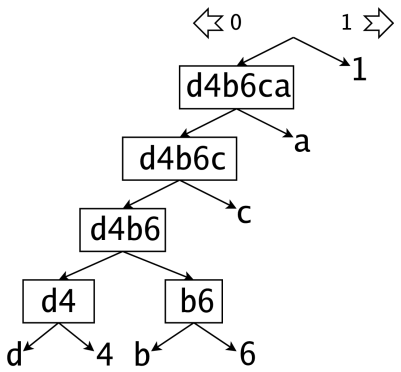



Table 7.5: Huffman coding demonstration of message "`1a1b6a1c1d4a1c1a`".

when only two characters are remaining. Then the tree is complete and codes can be generated by moving in the Huffman tree by two bit values either to left or right respectively.

| tree level | characters/counts | | | | | | |
|---|---|---|---|---|---|---|---|
| 6. | 1 | a | c | b | 6 | d | 4 |
| | 6 | 4 | 2 | 1 | 1 | 1 | 1 |
| 5. | 1 | a | c | d4 | b | 6 | |
| | 6 | 4 | 2 | 2 | 1 | 1 | |
| 4. | 1 | a | c | d4 | b6 | | |
| | 6 | 4 | 2 | 2 | 2 | | |
| 3. | 1 | a | d4b6 | c | | | |
| | 6 | 4 | 4 | 2 | | | |
| 2. | 1 | d4b6c | a | | | | |
| | 6 | 6 | 4 | | | | |
| 1. | d4b6ca | 1 | | | | | |
| | 10 | 6 | | | | | |

Table 7.6: Connection of characters to form nodes in Huffman tree.

## 7.7 Arithmetic Coding

The basic principle of arithmetic coding is that an interval $I = \langle 0, 1)$ is decomposed to subintervals proportional to a probability of each character in the message. For each character $c_n$ in the message we will reduce range of $I$ to interval $I = \langle I_n, I_{n+1})$, where $I_n$ position is defined by the model cummulative histogram. Now the model can be eventually updated and the interval $I$ is further decomposed to a smaller intervals proportional to the model. This is done for all the characters in the message. At the end of encoding we have some result interval $I_N$. The output of arithmetic coder is a smallest binary number, scaled to the original interval $\langle 0, 1)$, whose value belongs to the interval $I_N$.

### 7.7.1 Semi–Adaptive Arithmetic Coding Demonstration

To demonstrate the algorithm itself we will use semi–adaptive arithmetic encoding of message $M =$"`abaca`". If we gather a statistics of each character, we see that $P_{\texttt{a}} = \frac{3}{5}$, $P_{\texttt{b}} = \frac{1}{5}$, $P_{\texttt{c}} = \frac{1}{5}$. Then we set left $L_n$ and right $R_n$ probability limits to $L_0 = 0$, $R_0 = 1$, so that interval $\langle L_0, R_0)$ will be further decomposed while encoding $n$–th character of the message. Before encoding we decompose the interval to subintervals

proportional to probability of characters, so for $I_{\mathtt{a}} = \langle I^L_{\mathtt{a}}, I^R_{\mathtt{a}}) = \langle 0, \frac{3}{5})$, $I_{\mathtt{b}} = \langle \frac{3}{5}, \frac{4}{5})$, $I_{\mathtt{c}} = \langle \frac{3}{5}, 1)$. When the first character $c_0$ is encoded, the starting interval $\langle L_0, R_0)$ is shortened to $I_{c_0} = \langle L_1, R_1)$, which is then successively decomposed as the next characters are encoded.

| $n$ | $c_n$ | $L_n$ | $R_n$ | $\delta_n = R_n - L_n$ | $I^L_{c_n}$ | $I^R_{c_n}$ | $L_{n+1} = L_n + \delta_n I^L_{c_n}$ | $R_{n+1} = L_n + \delta_n I^R_{c_n}$ |
|---|---|---|---|---|---|---|---|---|
| 0 | a | 0 | 1 | 1 | 0 | $\frac{3}{5}$ | 0 | $\frac{3}{5}$ |
| 1 | b | 0 | $\frac{3}{5}$ | $\frac{3}{5}$ | $\frac{3}{5}$ | $\frac{4}{5}$ | $\frac{9}{25}$ | $\frac{12}{25}$ |
| 2 | a | $\frac{9}{25}$ | $\frac{12}{25}$ | $\frac{3}{25}$ | 0 | $\frac{3}{5}$ | $\frac{9}{25}$ | $\frac{54}{154}$ |
| 3 | c | $\frac{9}{25}$ | $\frac{54}{125}$ | $\frac{9}{125}$ | $\frac{4}{5}$ | 1 | $\frac{261}{625}$ | $\frac{54}{125}$ |
| 4 | a | $\frac{261}{625}$ | $\frac{54}{125}$ | $\frac{9}{625}$ | 0 | $\frac{3}{5}$ | $\frac{261}{625}$ | $\frac{1332}{3125}$ |

Table 7.7: Example of semi–adaptive arithmetic coding of message "abaca".

We can see that the final interval calculated from table 7.7 is $\langle \frac{261}{625}, \frac{1332}{3125})$. For the semi–adaptive coding holds:

$$\prod_{n=0}^{N-1} P_{c_n} = \delta_N,$$

thus the size of this interval is

$$P_a P_b P_a P_c P_a = \frac{3}{5}\frac{1}{5}\frac{3}{5}\frac{1}{5}\frac{3}{5} = \delta_5 = \frac{1332}{3125} - \frac{261}{625} = \frac{27}{3125}.$$

We are able to reconstruct the original message from a sole knowledge of the interval $\langle L_5, R_5)$. We now need to encode the interval to a binary form. A single binary number that belongs to this interval is sufficient for the lossless reconstruction. So the problem is to find a binary number $\mathcal{B}$:

$$\frac{\mathcal{B}}{2^k} \in \langle L_N, R_N), \tag{7.9}$$

such that the $k$ is minimal, i.e. $\mathcal{B}$ has to have the smallest possible number of bits to fit in $\langle L_N, R_N)$. This criterion ensures reaching maximal compression for the semi–adaptive scheme. The encoding and finding $\mathcal{B}$ is shown in fig. 7.1.

So we encoded the message "abaca" to the number $\mathcal{B} = 27$, i.e. to the 6–bit sequence 011011. The $k = 6$ is the minimal number of bits to encode the message because there would be no number to fit in $\langle L_N, R_N)$ if we make the scale in fig. 7.1 less dense.

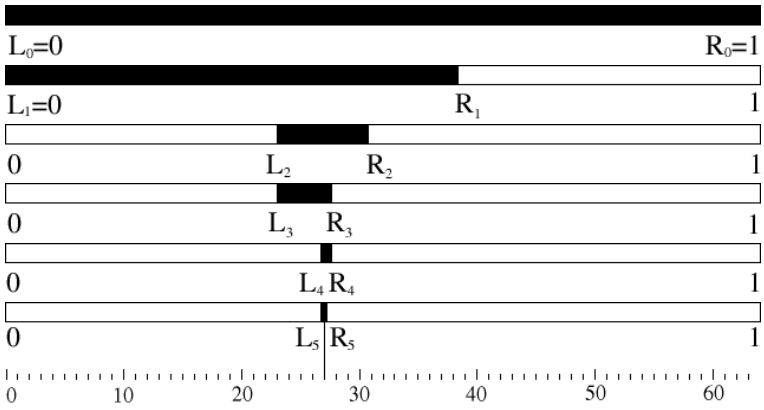



Figure 7.1: Graphical illustration of arithmetic coding.

So we reached the mean number of bits per character, entropy $H(\text{“}\texttt{abaca}\text{”}) = 1.2$ bits. Note that it is needed to know the probability distribution in order to reconstruct the message.

### 7.7.2 Adaptive Arithmetic Coding

Up to a recent time the usage of most efficient adaptive arithmetic coding was considered unpractical because of its high computational expense. This is because for every encoded character we have to do $O(N)$ operations to update our prediction model. This is not true any more as $O(\log N)$ algorithm is presented here. Even if this metod was originally designed, it was found that a similar method has been already published in [9].

### 7.7.3 $O(\log N)$ Algorithm for Cummulative Histogram Updating

It is quite intuitive that for calculation of cummulative histogram $c(x)$ of histogram $h(x)$, which describes probability distribution of $N$ character alphabet message is $O(N)$, because it is needed to sum all the probabilities present in $h(x)$. It can be simplified to binary tree tracing problem. Let us have a binary tree $T_l(x), l \in$

$\{0, 1, ...L\}, x = \{0, 1, ...X\}$, which has $L = \log_2 N$ levels and at each level $X = 2^{l+1}$ nodes, this is shown for $N = 8$ in the table 7.8. If we put the whole $h(x)$ to the

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $T_0(0)$ | | | | $\|T_0(1)$ | | | |
| $T_1(0)$ | | $\|T_1(1)$ | | $\|T_1(2)$ | | $\|T_1(3)$ | |
| $T_2(0)$ | $\|T_2(1)$ | $\|T_2(2)$ | $\|T_2(3)$ | $\|T_2(4)$ | $\|T_2(5)$ | $\|T_2(6)$ | $\|T_2(7)$ |

Table 7.8: The form of $N = 8$ binary tree used for cummulative histogram calculation.

highest level of $T$, then we can calculate successive additions at higher levels of $T$ as:

$$T_l(x) = T_{l+1}(2x) + T_{l+1}(2x + 1), \quad \forall x \in \{0, 1, ...2^{l+1} - 1\}, \forall l \in \{L - 2, L - 3, ...0\},$$

then the value of cummulative histogram for example for $c(5) = T_0(0) + T_2(4)$. In general the total number of additions in worst case is equal to $\log_2 N$, thus the complexity of the algorithm was reduced to $O(\log N)$, what is the significant speed boost which makes adaptive arithmetic coding practical.

### 7.7.4 Finite Context Prediction

A smart approach can be used to yet improve the adaptive arithmetic encoding scheme to result in even better results than predicted by Shannon's entropy (7.7), but in agree with (7.8). The idea is in exploiting the nature of adaptive arithmetic coding, where the prediction model is updated after encoding of each character of a message. The point is that a better prediction during encoding can be made if a model, which predicts the next character in message to be encoded is not based on statistics from all the encoded characters but only on a statistics of a few characters close to encoding position (a finite context) in message. This is because there can be found contexts within a message that have different distribution that the message as the whole. Unfortunately a prediction of optimal finite context[2] size is difficult to find in a straight way, so iterative algorithms are used. In fig. 7.2 can be seen results of encoding fig. 7.4, considered as message of $2^{20} = 1048576$ characters of 135 stated alphabet, where the optimal context size is 4096 characters to reach minimal $H = 3.387b$.

[2]Also called a finite window.

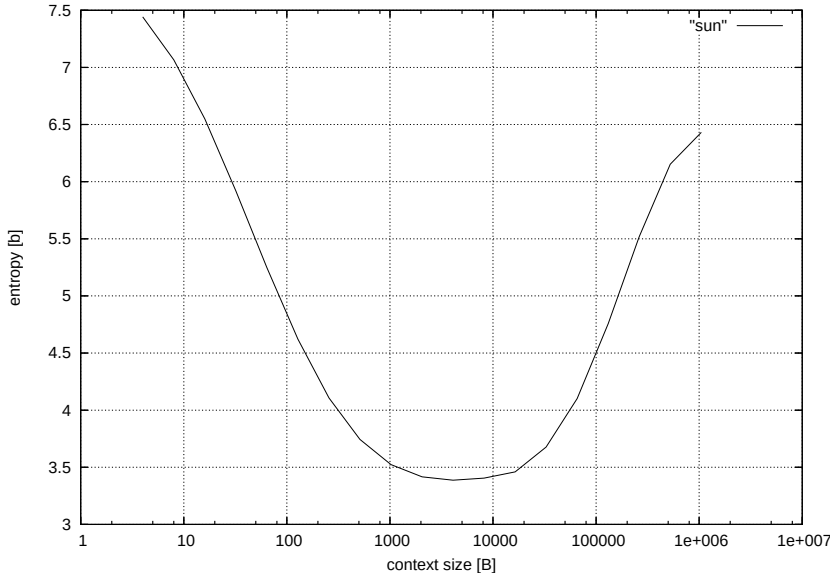



Figure 7.2: Entropy dependence on context size.

## 7.8 Lossless Compression Algorithm Proposal

The proposed algorithm for lossless image compression is based on Burrows-Wheeler block-sorting algorithm [1], [10], hierarchycal RLE coding (sec. 7.1.4) and adaptive arithmetic coding (sec. 7.7) with finite context prediction.

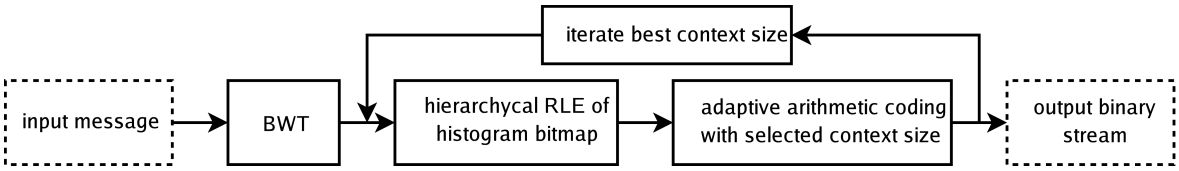



Figure 7.3: Scheme of the proposed compression algorithm.

The characters within a message are first reordered using the BWT to gather characters of similar contexts close together. Then the message is separated to blocks of specified size. Then for each block a histogram bitmap is calculated to reduce message alphabet states in the block passed to the entropy coder. The histogram bitmaps are encoded by the hierarchycal RLE encoding. Multiple iteration passes are then performed in order to find an optimal size of context to compress the message

to a minimal size. The context starts at size of 256 characters and is doubled in each iteration up to 16384 characters. Usual range for the optimal context size is 1024 up to 8192 characters. The reason for doubling the context size is time expense of the compression. It could be found in a precission up to one character. The compression software supports up to 16bit per letter characters in a message. The purpose of the hierarchycal RLE coding in the compression scheme in table 7.3 is that for 16-bit characters we have to store histogram bitmap for 65536 characters, what is the bitmap of size 8192 bytes what is frequently reduced below 1/100 of its original size. In case of 8-bit characters in message, the histogram size is mostly halved. A comparison of effectivity of the proposed algorithm and other commercial and non-commercial lossless compression formats can be seen from table 7.9.

| Name | File size [B] | Entropy [b] | Algorithm |
|---|---|---|---|
| GIF | 700 206 | 5.342 | LZW |
| TIFF | 621 772 | 4.743 | LZ77 |
| GNU zip v1.3.3 | 608 016 | 4.638 | LZ77 |
| UNIX compress v4.2.4 | 604 225 | 4.609 | LZW |
| RAR v3.3 beta 1 | 566 518 | 4.322 | unknown |
| PNG | 542 779 | 4.141 | LZ77 |
| bzip2 v1.0.2 | 470 925 | 3.592 | BWT, Huffman |
| proposed algorithm | 427 934 | 3.265 | BWT, Arit. |

Table 7.9: Results and comparison of the proposed compression algorithm.

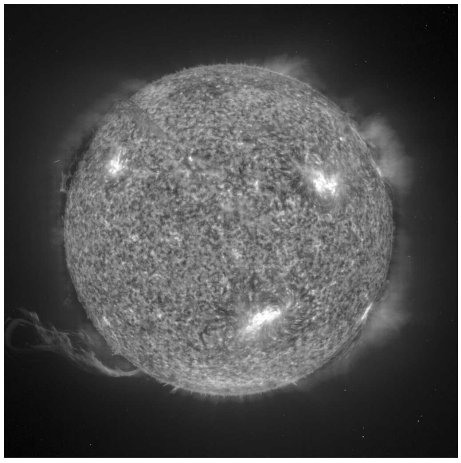


Figure 7.4: Sample 1024 × 1024 grayscale image used for compression.

## 7.9 Achievements and Original Contributions

This section is added to comprehensively summarize what was done originally in what chapter to make an evaluation easier for oponents. These areas are summarized in the following list.

- **chapter 1, section 1.8**: Algorithm of black body color calculation at various temperatures was created and implemented, based on integration of spectral colors derived from the CIE diagram and Planck black body irradiation in the visible spectrum.

- **chapter 2, section 2.2**: Signal sample arithmetics is described from the lowest level point of view in order to make a slight insight to how a digital signal is actually handled by a computer what should demystify many things to non-programming scientists and theoreticians.

- **chapter 2, section 2.4**: Various types of discrete convolution are described with wide range of applications, particularly non–cyclic convolution ones.

- **chapter 2, section 2.4.5**: A new adaptive kernel convolution method was developed for a purpose of noise, bad pixel detection and filtration from ortho–elevation surface maps. This method was developed for TU Wien, dept. of Geography for filtering MOLA mars elevation maps also with an interactive software.

- **chapter 3, sections 3.1.1, 3.1.2, 3.1.3**: Original comprehesive implementations of both DIT and DIF FFT algorithms and optimized bit reversal routines are presented there.

- **chapter 3, section 3.2.2**: Original software based on phase correlation techniques for shift detection of slices in volume rendering was implemented and used to correct slices acquired from the Visible Human Project to make the data actually usable for volume rendering.

- **chapter 4**: An introduction to one and two dimensional wavelet analysis is presented here with particular aim to comprehense. Original software was implemented for searching all possible 4–tap filters forming a wavelet base for a given coefficient quantization, what is presented in table 4.1 for 4–bit and in appendix for 7–bit coefficient quantization.

- **chapter 5**: This chapter is written originally from scratch in order to describe how raycasting volume rendering is implemented. It can be shown on interactive volume rendering software that was implemented. Some sample renderings are shown in appendix.

- **chapter 6**: Introduction to decorrelating transforms is presented here again with particular aim to comprehense. Optimal decorrelating basis vetors are shown for a sample image calculated by self-implemented software and are then compared with other decorrelating transforms, where similarity of basis vectors can be compared also visually. Then the proposed DCT based progressive lossy compression software is described.

- **chapter 7**: Originally written guide to lossless compression methods is presented here, new are approaches to RLE encoding and arithmetic entropy coding. It demonstrates the methods on particular examples which are to be easily generalized by reader to avoid a need to study hard theory behind these methods. The original lossless compression software design is then presented based on described compression methods including comparison with other lossless compression codecs.

# Appendix

| $n$ | $h_k$ unnormalized | $h_k$ normalized | zeroed | MSE |
|---|---|---|---|---|
| 1 | $+0+0-63-63$ | $+0.000000+0.000000-0.707107-0.707107$ | 95.49% | 45.17 |
| 2 | $-18+10-35-63$ | $-0.240149+0.133416-0.466957-0.840523$ | 92.39% | 66.15 |
| 3 | $-35+10-18-63$ | $-0.466957+0.133416-0.240149-0.840523$ | 90.02% | 77.28 |
| 4 | $-63+0+0-63$ | $-0.707107+0.000000+0.000000-0.707107$ | 89.12% | 81.16 |
| 5 | $-12+8-40-60$ | $-0.163178+0.108786-0.543928-0.815892$ | 93.40% | 60.38 |
| 6 | $-15+9-36-60$ | $-0.207973+0.124784-0.499134-0.831890$ | 92.82% | 64.01 |
| 7 | $-20+10-30-60$ | $-0.282843+0.141421-0.424264-0.848528$ | 91.85% | 68.87 |
| 8 | $-30+10-20-60$ | $-0.424264+0.141421-0.282843-0.848528$ | 90.34% | 75.58 |
| 9 | $-36+9-15-60$ | $-0.499134+0.124784-0.207973-0.831890$ | 89.83% | 78.50 |
| 10 | $-40+8-12-60$ | $-0.543928+0.108786-0.163178-0.815892$ | 89.56% | 79.54 |
| 11 | $-8+6-42-56$ | $-0.113137+0.084853-0.593970-0.791960$ | 94.04% | 55.90 |
| 12 | $-42+6-8-56$ | $-0.593970+0.084853-0.113137-0.791960$ | 89.36% | 80.69 |
| 13 | $-5+4-36-45$ | $-0.086233+0.068986-0.620874-0.776093$ | 94.39% | 53.43 |
| 14 | $-36+4-5-45$ | $-0.620874+0.068986-0.086233-0.776093$ | 89.26% | 80.93 |
| 15 | $+6-8-56-42$ | $+0.084853-0.113137-0.791960-0.593970$ | 96.22% | 38.35 |
| 16 | $-7+5-30-42$ | $-0.133777+0.095555-0.573330-0.802662$ | 93.77% | 57.75 |
| 17 | $-30+5-7-42$ | $-0.573330+0.095555-0.133777-0.802662$ | 89.42% | 80.12 |
| 18 | $-56-8+6-42$ | $-0.791960-0.113137+0.084853-0.593970$ | 89.35% | 80.28 |
| 19 | $+8-12-60-40$ | $+0.108786-0.163178-0.815892-0.543928$ | 96.38% | 36.98 |
| 20 | $-60-12+8-40$ | $-0.815892-0.163178+0.108786-0.543928$ | 89.55% | 79.21 |
| 21 | $+9-15-60-36$ | $+0.124784-0.207973-0.831890-0.499134$ | 96.38% | 36.18 |
| 22 | $+4-5-45-36$ | $+0.068986-0.086233-0.776093-0.620874$ | 96.08% | 39.29 |
| 23 | $-45-5+4-36$ | $-0.776093-0.086233+0.068986-0.620874$ | 89.29% | 80.85 |

Table 7.10: 4–tap $h_k$ filters generating orthogonal wavelet bases for 7–bit coefficient quantization calculated from 268 435 456 possible candidates, part I.

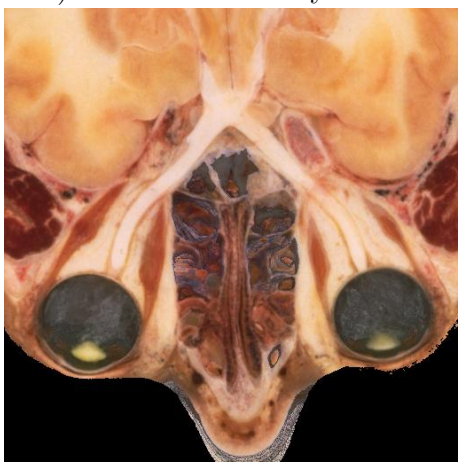
| $n$ | $h_k$ unnormalized | $h_k$ normalized | zeroed | MSE |
|---|---|---|---|---|
| 24 | $-60-15+9-36$ | $-0.831890-0.207973+0.124784-0.499134$ | 89.82% | 78.15 |
| 25 | $+10-18-63-35$ | $+0.133416-0.240149-0.840523-0.466957$ | 96.36% | 36.37 |
| 26 | $-14+6-15-35$ | $-0.341362+0.146298-0.365745-0.853405$ | 91.18% | 72.20 |
| 27 | $-15+6-14-35$ | $-0.365745+0.146298-0.341362-0.853405$ | 90.89% | 73.15 |
| 28 | $-63-18+10-35$ | $-0.840523-0.240149+0.133416-0.466957$ | 90.05% | 77.20 |
| 29 | $+10-20-60-30$ | $+0.141421-0.282843-0.848528-0.424264$ | 96.31% | 37.03 |
| 30 | $+5-7-42-30$ | $+0.095555-0.133777-0.802662-0.573330$ | 96.28% | 37.49 |
| 31 | $-42-7+5-30$ | $-0.802662-0.133777+0.095555-0.573330$ | 89.40% | 79.66 |
| 32 | $-60-20+10-30$ | $-0.848528-0.282843+0.141421-0.424264$ | 90.42% | 75.81 |
| 33 | $+10-30-60-20$ | $+0.141421-0.424264-0.848528-0.282843$ | 96.01% | 41.11 |
| 34 | $-60-30+10-20$ | $-0.848528-0.424264+0.141421-0.282843$ | 91.93% | 68.89 |
| 35 | $+10-35-63-18$ | $+0.133416-0.466957-0.840523-0.240149$ | 95.90% | 42.26 |
| 36 | $-63-35+10-18$ | $-0.840523-0.466957+0.133416-0.240149$ | 92.47% | 66.09 |
| 37 | $+9-36-60-15$ | $+0.124784-0.499134-0.831890-0.207973$ | 95.78% | 42.74 |
| 38 | $+6-14-35-15$ | $+0.146298-0.341362-0.853405-0.365745$ | 96.20% | 38.51 |
| 39 | $-35-14+6-15$ | $-0.853405-0.341362+0.146298-0.365745$ | 90.98% | 73.35 |
| 40 | $-60-36+9-15$ | $-0.831890-0.499134+0.124784-0.207973$ | 92.89% | 63.71 |
| 41 | $+6-15-35-14$ | $+0.146298-0.365745-0.853405-0.341362$ | 96.14% | 39.31 |
| 42 | $-35-15+6-14$ | $-0.853405-0.365745+0.146298-0.341362$ | 91.25% | 72.23 |
| 43 | $+8-40-60-12$ | $+0.108786-0.543928-0.815892-0.163178$ | 95.65% | 43.61 |
| 44 | $-60-40+8-12$ | $-0.815892-0.543928+0.108786-0.163178$ | 93.46% | 59.93 |
| 45 | $+35+63+18-10$ | $+0.466957+0.840523+0.240149-0.133416$ | 96.43% | 36.10 |
| 46 | $+30+60+20-10$ | $+0.424264+0.848528+0.282843-0.141421$ | 96.36% | 36.84 |

Table 7.11: 4–tap $h_k$ filters generating orthogonal wavelet bases for 7–bit coefficient quantization calculated from 268 435 456 possible candidates, part II.

| $n$ | $h_k$ unnormalized | $h_k$ normalized | zeroed | MSE |
|---|---|---|---|---|
| 47 | $+20+60+30-10$ | $+0.282843+0.848528+0.424264-0.141421$ | 95.97% | 40.95 |
| 48 | $+18+63+35-10$ | $+0.240149+0.840523+0.466957-0.133416$ | 95.83% | 42.04 |
| 49 | $+36+60+15-9$ | $+0.499134+0.831890+0.207973-0.124784$ | 96.48% | 36.22 |
| 50 | $+15+60+36-9$ | $+0.207973+0.831890+0.499134-0.124784$ | 95.74% | 42.78 |
| 51 | $+6-42-56-8$ | $+0.084853-0.593970-0.791960-0.113137$ | 95.57% | 44.38 |
| 52 | $-56-42+6-8$ | $-0.791960-0.593970+0.084853-0.113137$ | 94.09% | 55.31 |
| 53 | $+40+60+12-8$ | $+0.543928+0.815892+0.163178-0.108786$ | 96.45% | 36.73 |
| 54 | $+12+60+40-8$ | $+0.163178+0.815892+0.543928-0.108786$ | 95.65% | 43.97 |
| 55 | $+5-30-42-7$ | $+0.095555-0.573330-0.802662-0.133777$ | 95.61% | 44.13 |
| 56 | $-42-30+5-7$ | $-0.802662-0.573330+0.095555-0.133777$ | 93.84% | 57.29 |
| 57 | $+42+56+8-6$ | $+0.593970+0.791960+0.113137-0.084853$ | 96.32% | 38.44 |
| 58 | $+15+35+14-6$ | $+0.365745+0.853405+0.341362-0.146298$ | 96.21% | 38.49 |
| 59 | $+14+35+15-6$ | $+0.341362+0.853405+0.365745-0.146298$ | 96.13% | 39.13 |
| 60 | $+8+56+42-6$ | $+0.113137+0.791960+0.593970-0.084853$ | 95.56% | 44.94 |
| 61 | $+4-36-45-5$ | $+0.068986-0.620874-0.776093-0.086233$ | 95.59% | 45.10 |
| 62 | $-45-36+4-5$ | $-0.776093-0.620874+0.068986-0.086233$ | 94.42% | 52.73 |
| 63 | $+30+42+7-5$ | $+0.573330+0.802662+0.133777-0.095555$ | 96.39% | 37.69 |
| 64 | $+7+42+30-5$ | $+0.133777+0.802662+0.573330-0.095555$ | 95.60% | 44.61 |
| 65 | $+36+45+5-4$ | $+0.620874+0.776093+0.086233-0.068986$ | 96.18% | 39.53 |
| 66 | $+5+45+36-4$ | $+0.086233+0.776093+0.620874-0.068986$ | 95.52% | 45.04 |
| 67 | $+0-63-63+0$ | $+0.000000-0.707107-0.707107+0.000000$ | 95.47% | 45.25 |
| 68 | $-63-63+0+0$ | $-0.707107-0.707107+0.000000+0.000000$ | 95.51% | 44.85 |

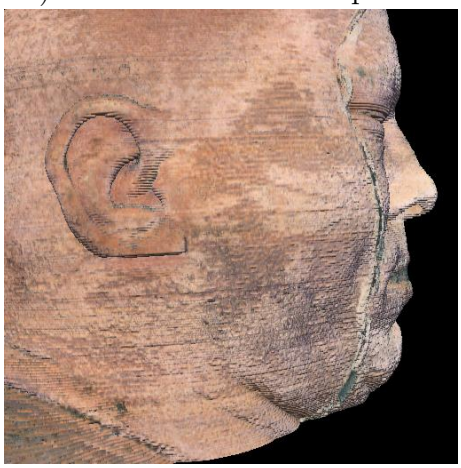
Table 7.12: 4–tap $h_k$ filters generating orthogonal wavelet bases for 7–bit coefficient quantization calculated from 268 435 456 possible candidates, part III.

a) cross section in eyes area

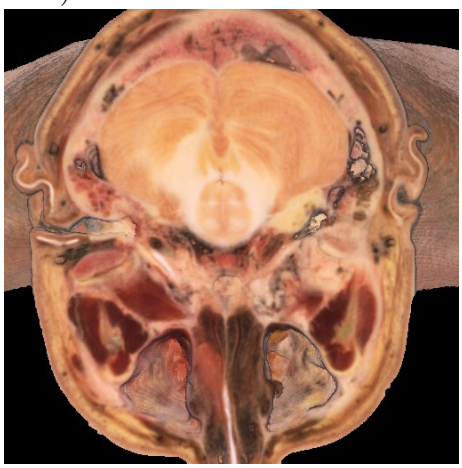


b) face of the cadaver

c) cadaver viewed from profile



d) nose hollow and ear area



e) photography of Joseph Paul Jernigan

f) area with cervical, pith and vocal chord

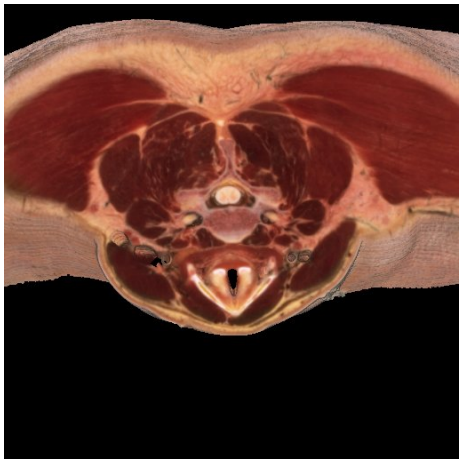


Figure 7.5: Application to the Visible Human Project.

# Bibliography


[1] Burrows M., Wheeler D.J. *A Block-sorting Lossless Data Compression Algorithm*. Digital System Research Center, 1994.

[2] CCIR Rec.709. *Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange*. ITU-R Recommendation BT.709, 1990.

[3] Daubechies, I. Orthonormal bases of compactly supported wavelets. *Comm. Pure Appl. Math 41* (1988), 909–996.

[4] Druckmüller M. *Opacity simulation technique for confocal microscope image processing*, vol. 1. Journal of Applied Biomedicine, 2003.

[5] Engineering Productivity Tools Ltd. *The FFT Demystified V2.1*. Engineering Productivity Tools Ltd., http://www.epstools.com/, 1999.

[6] Harna Z. *Přesná mechanika*. Vysoké učení technické v Brně, Kounicova 67a, Brno, 1996.

[7] Huffman D.A. *A method for the construction of minimum-redundancy codes*, vol. 40. Proceedings of the IRE, 1952.

[8] Martišek D. *The 2D and 3D Processing of Images Provided by Conventional Microscopes*, vol. 24. Scanning, 2002.

[9] Nelson, M. Arithmetic coding + statistical modellig = data compression. *Dr. Dobb's Journal 1* (1991).

[10] Nový J. *Numerické metody rekonstrukce 3D modelu zemského povrchu.* ÚFI FSI VUT Brno - diplomová práce, 2002.

[11] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[12] Rogers D. F. *Procedural Elements for Computer Graphics.* McGraw-Hill Book Company, New York, 1985.

[13] Shannon C. E., Weaver W. *The Mathematical Theory of Communication.* The University of Illinois Press, Urbana, 1964.

[14] Smith S. W. *The Sciencist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, P.O. Box 502407, San Diego, CA 92150-2407, 1999.

[15] Uytterhoeven G. *Wavelets: Software and Application.* Katholieke Universiteit Lueven, Faculteit Toegepaste Wetenschappen Arenbergkasteel, B-3001 Haverlee, Belgium, 1999.

[16] Wallace G. K. *The JPEG still picture compression standard.* Communications of the ACM, 1991.

[17] Ziv, Lempel. *A universal algorithm for sequential data compression*, vol. 23. IEEE Transactions in Information Theory, 1977.

[18] Ziv, Lempel. *Compression of individual sequences via variable-rate coding*, vol. 24. IEEE Transactions in Information Theory, 1978.