Image Processing - Image Fundamentals

Institute for Biomedical Imaging, Hamburg University of Technology

- Lecture: Prof. Dr.-Ing. Tobias Knopp
- Exercise: Konrad Scheffler, M.Sc.

Table of Contents

Image Processing - Image Fundamentals
1. Mathematical Notation
2. Image Foundations
   2.1 Discrete Images
   2.2 Coordinate System
   2.3 Visualization
   2.4 Discretization
      2.4.1 Sampling
      2.4.2 Quantization
   2.5 Image Characteristics
      2.5.1 Spatial Resolution
      2.5.2 Contrast
      2.5.3 Noise
      2.5.4 Intensity Resolution
   2.6 Memory Footprint
3. Image Interpolation
4. Image Transformations
   4.1 Linear Transformations
   4.2 Shift Invariance
   4.3 Matrix Vector Notation
   4.4 Fast Transformations
5. Geometric Transformations
   5.1 Affine Linear Transformations
6. Wrapup

1. Mathematical Notation

In this course we use standard mathematical notation where small boldface letters denote vectors and capital boldface letters denote matrices. Moreover, we use the symbol * to denote complex conjugation of either a scalar, vector or a matrix. In the case of a vector or matrix, complex-conjugation is understood to be performed component-wise. For a matrix $\mathbf{A}$, we use $\mathbf{A}^\mathsf{T}$ to denote the transposed matrix and $\mathbf{A}^\mathsf{H} = (\mathbf{A}^*)^\mathsf{T}$ to denote the complex-conjugated and transposed matrix.

For a vector $\mathbf{x} \in \mathbb{R}^N$, the transposed vector $\mathbf{x}^\mathsf{T}$ refers to the corresponding row vector obtained when interpreting $\mathbf{x}$ as a $(N \times 1)$-matrix.

For two vectors or matrices, the Hadamard product is defined as

$$\mathbf{A} \odot \mathbf{B} = \begin{pmatrix} a_{1,1} \cdot b_{1,1} & \cdots & a_{1,N} \cdot b_{1,N} \\ \vdots & \ddots & \vdots \\ a_{M,1} \cdot b_{M,1} & \cdots & a_{M,N} \cdot b_{M,N} \end{pmatrix}$$

The Hadamard division is defined as

$$\mathbf{A} \oslash \mathbf{B} = \begin{pmatrix} a_{1,1} / b_{1,1} & \cdots & a_{1,N} / b_{1,N} \\ \vdots & \ddots & \vdots \\ a_{M,1} / b_{M,1} & \cdots & a_{M,N} / b_{M,N} \end{pmatrix}$$

2. Image Foundations

Images can be expressed mathematically as functions $f : \Omega \to \Gamma$ with image domain $\Omega \subseteq \mathbb{R}^D$, where $D \in \mathbb{N}$ is the dimensionality of the image, and the color space Γ. In most cases $D = 2$ but we note that images do not need to be 2D but can also be 3D, 4D, 3D+t, ...

Example

A classical JPEG image from the internet is 2D, a video shown on youtube is 2D+t.

We often write out the image function as

$$f(x, y) \text{ or } f(x, y, t)$$

or even more general just

$$f(\mathbf{r})$$

where $\mathbf{r} \in \Omega$ is then the (spatial) variable. Usually, Ω is a subset of the real numbers $\mathbb{R}^D$ in case of a continuous image or a subset of the natural numbers $\mathbb{N}^D$ in case of a discrete image. For instance continuous images are usually defined over $\Omega = [0, 1]^D$. If the image has sizes (e.g. $9\,\text{cm} \times 7\,\text{cm}$) one can map to the unit cube by scaling each dimension accordingly. Discrete images are defined over index sets

$$I_N = \{1, \dots, N\}$$

i.e. an $N_x \times N_y$ image could be defined as $f : I_{N_x} \times I_{N_y} \to \mathbb{R}$. Discrete images are usually not scaled to $[0, 1]^D$ but they are defined on the index coordinates.

Both continuous and discrete image can have a physical size measured in m (mm, cm, inch, ...). This size can be given by the vector $\mathbf{s} \in \mathbb{R}^D$ such that one can switch between the physical and the

normalized domain using the coordinate transform:

$$\mathbf{r}_{\text{physical}} = \mathbf{s} \odot \mathbf{r}_{\text{unit cube}}$$

if the image is defined on the unit cube $[0,1]^D$.

If the image is defined on the index sets we can switch back and forth between image coordinates and physical coordinates using

$$\mathbf{r}_{\text{physical}} = \mathbf{s} \odot (\mathbf{r}_{\text{normalized}} - 0.5) \oslash \mathbf{N}$$

This coordinate transform assumes that we are using a cell-centered grid (see section 2.2).

If required one can also extend these definitions with an additional offset, which is helpful when considering a series of images that needs to be stitched together.

The range is usually chosen to be $\mathbb{R}$, $[0,\infty)$ or $[0,C]$ when considering a continuous range. Most common is to normalize the range to the interval $[0,1]$ (i.e. $C=1$).

Pixel

If f is a discrete image it is composed of image *pixels* (*voxels* in 3D). The term pixel is short term for *picture elements* and describes both the coordinates $(\mathbf{r})$, the value $f(\mathbf{r})$ and the pixel size Δ. The later can be the dimensionless pixel size, which is $\Delta = \mathbf{1} \oslash \mathbf{N}$ for an image on the unit cube and $\Delta = \mathbf{1}$ for indexing coordinates. In case that the image has a physical size one uses the dimensional pixel size $\Delta = \mathbf{s} \oslash \mathbf{N}$.

Formally one can think of a pixel as a triple $(\mathbf{r}, f(\mathbf{r}), \Delta)$. But usually the term is used informally and can mean also a subset of this triple.

Intensity

Elements of the color space Γ are called *intensities*. They can have some physical meaning including a physical unit but to achieve abstraction, the actual unit is often ignored. If you have camera pictures usually it is the light captured with a sensor that is encoded in the image intensity.

> **Note**
>
> When *processing* an image one usually considers images having a discrete domain and a continuous range. When *storing* an image both the domain and the range are considered to be discrete.

2.1 Discrete Images

Matrix Representation

Two-dimensional discrete images can be either represented as a function $f(i,j)$ or as a matrix

$$\mathbf{f} = (f_{i,j})_{i=1,\ldots,N_x;j=1,\ldots,N_y} = \begin{pmatrix} f(1,1) & \ldots & f(1,N_y) \\ \vdots & \vdots & \vdots \\ f(N_x,1) & \ldots & f(N_x,N_y) \end{pmatrix}$$

Obviously the matrix entries $f_{i,j}$ correspond to the function values $f(i,j)$.

> **Note**
>
> In image processing one often switches back and forth between the function and the matrix representation. The function representation is more useful if one needs to evaluate an image at off-pixel positions such as in interpolation tasks. The matrix representation is very useful when investigating neighbors of pixels.

Array Representation

The third representation that we can use is a (multidimensional) array. An array is a programming data structure that can hold a certain number of elements and is usually layed out contiguous in the main memory leading to $\mathcal{O}(1)$ access times to individual array elements (read/write).

Multidimensional arrays are stored in linear main memory (or files) by simply traversing the array where the first index is usually the fastest and the last index is usually the slowest. We also name this *flattening* of a multidimensional array into a 1D array. Mathematically one can describe this for an $N_1 \times \cdots \times N_d$ array $\mathbf{f}$ like this:

$$f_{i_1+N_1(i_2+N_2(i_3+\ldots N_{d-1}i_d)} = f_{i_1,i_2,\ldots,i_d}$$

For instance for $d=2$ we have

$$f_{i_1+N_1 i_2} = f_{i_1,i_2}$$

This conversion between 1D and multidimensional arrays is also often done implicitly which brings us to the *indexing strategies*. We name the indexing on the right a *multidimensional* or *Cartesian indexing*. The indexing on the left is named *linear indexing*.

Which one to use depends on the application. If the order of the elements is irrelevant (under a sum) one can use a linear index and prevent some overhead associated with Cartesian indexing. In case of operations where neighborhood relations are important (e.g. a convolution) a Cartesian index is much more efficient.

Let us shortly look into what Julia does here:

```
B = 3×4 Matrix{Int64}:
 1  3   5   7
 2  6  10  14
 3  9  15  21
```

```
1 B = kron(1:3,(1:2:8)')
```

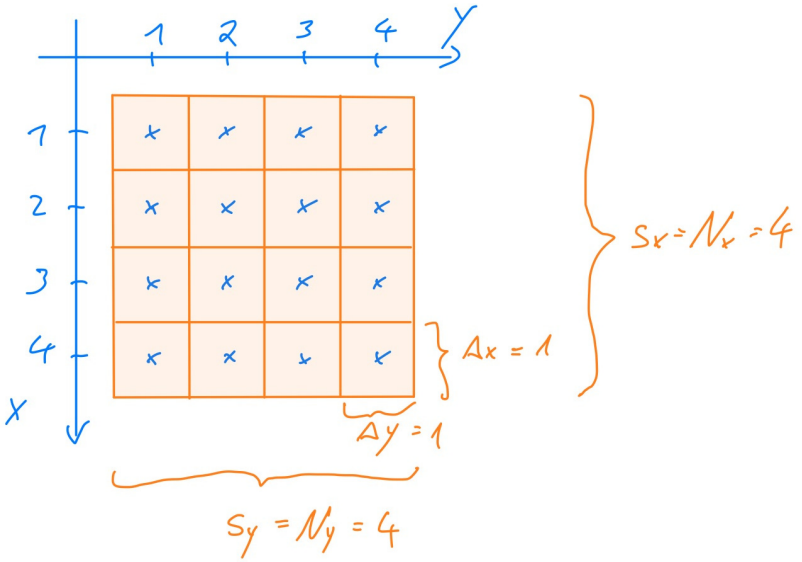
```
Cartesian indexing
1 1 1
2 1 2
3 1 3
1 2 3
2 2 6
3 2 9
1 3 5
2 3 10
3 3 15
1 4 7
2 4 14
3 4 21
Linear indexing
1 1
2 2
3 3
4 3
5 6
```

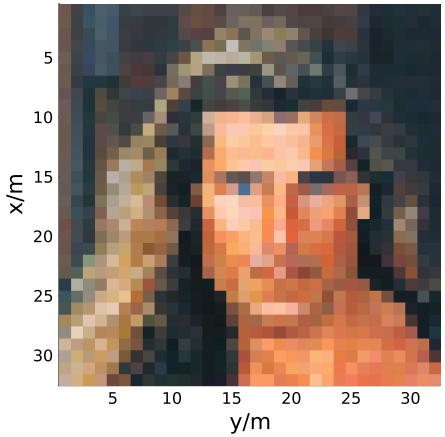
```
with_terminal() do
    println("Cartesian indexing")
    for i2 = 1:size(B,2)
        for i1 = 1:size(B,1)
           println("$i1 $i2 $(B[i1,i2])")
        end
    end

    println("Linear indexing")
    for i = 1:length(B) # = size(B,1)*size(B,2)
        println("$i $(B[i])")
    end
end
```

2.2 Coordinate System

The parameters N_x and N_y define the image resolution. Its fraction N_x/N_y defines the aspect ratio of an image. If $N_y > N_x$ then the image is in *landscape* format, if $N_y > N_x$ it is in *portrait* format.

The coordinate system that we will use most of the time looks like this:






```
begin
  fabio = testimage("fabio", nowarn=true)
  heatmap(imresize(fabio,32,32), legend=true, xlabel="y/m", ylabel="x/m", size=
(350,350))
end
```

Observations:

- upper left pixel is the origin (1,1)
- the x axis is the vertical axis and is directed downwards
- the y axis os the horizontal axis and is directed to the right
- the coordinate system is right-handed but just 90 degrees rotated to what you might have expected from a typical xy graph

> **Note**
>
> We use a one-based integer based indexing with $(1,1)$ being the upper left and (N_x, N_y) being the lower right pixel. This is useful since many programming languages use a 1-based indexing. Another convention is to use $(0,0)$ and $(N_x - 1, N_y - 1)$.

Image center

The image center is located at $\mathbf{r}_{\text{center}} = (\frac{N_x+1}{2}, \frac{N_y+1}{2})$. This just corresponds to a pixel location in case that both N_x and N_y are odd. One alternative is to define the image center as

$$\mathbf{r}_{\text{center}} = \left(\left\lfloor \frac{N_x + 1}{2} \right\rfloor, \left\lfloor \frac{N_y + 1}{2} \right\rfloor \right)$$

which ensures that always an image index is selected.

2.3 Visualization

We have already displayed an image using colors on the screen what makes somewhat sense since this is the representation that we associate with an image. But there are more ways. Let us first load a real valued image:

```
fabioReal = Float64.(Gray.(testimage("fabio", nowarn=true)));
```

Now we can visualize this image in the following ways:

Matrix View

First of all we can look at the image by showing the underlying values in their digit representation:

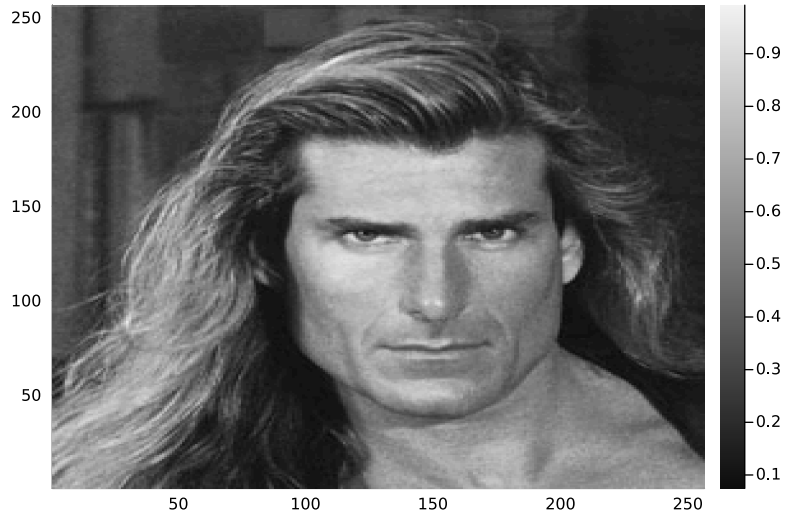
```
256×256 Matrix{Float64}:
 0.517647  0.494118  0.486275  0.466667  …  0.188235  0.184314  0.184314  0.188235
 0.372549  0.34902   0.384314  0.368627     0.180392  0.180392  0.184314  0.184314
 0.388235  0.360784  0.372549  0.364706     0.176471  0.176471  0.180392  0.180392
 0.368627  0.341176  0.376471  0.372549     0.168627  0.172549  0.176471  0.180392
 0.368627  0.345098  0.360784  0.356863     0.168627  0.168627  0.176471  0.176471
 0.380392  0.360784  0.364706  0.364706  …  0.172549  0.172549  0.176471  0.176471
 0.384314  0.372549  0.364706  0.364706     0.176471  0.176471  0.180392  0.180392
 ⋮                                    ⋱                                 ⋮
 0.443137  0.509804  0.580392  0.686275  …  0.623529  0.607843  0.611765  0.627451
 0.521569  0.52549   0.662745  0.752941     0.647059  0.627451  0.631373  0.643137
 0.517647  0.572549  0.705882  0.72549      0.654902  0.635294  0.631373  0.65098
 0.592157  0.666667  0.713725  0.647059     0.666667  0.662745  0.658824  0.658824
 0.690196  0.721569  0.678431  0.619608     0.705882  0.709804  0.694118  0.67451
 0.603922  0.666667  0.666667  0.698039  …  0.694118  0.678431  0.654902  0.639216
```

```
fabioReal
```

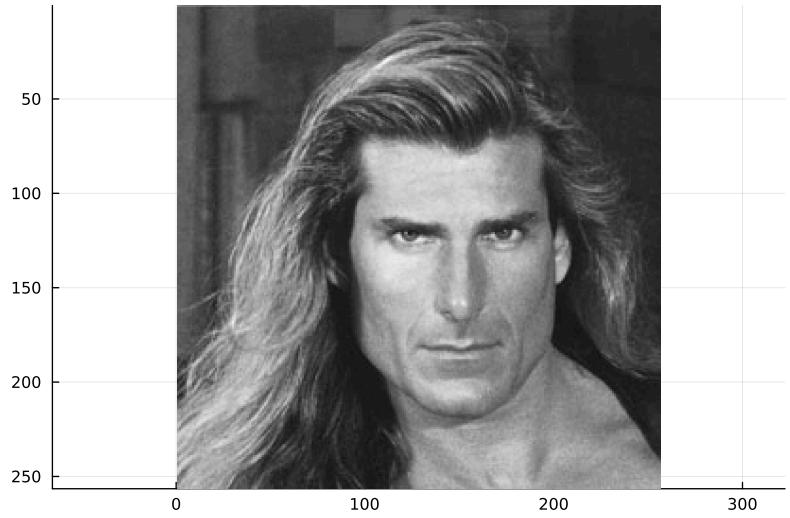
This is helpful when you need the precise quantitative values and want to compare. The human perception system is, however, not so good in interpreting this data efficiently.

Image View

Then of course we can show it as an image. The following shows two ways how you can do this with Julia's plotting package Plots.jl:




```
heatmap(reverse(fabioReal,dims=1),c=:grays)
```



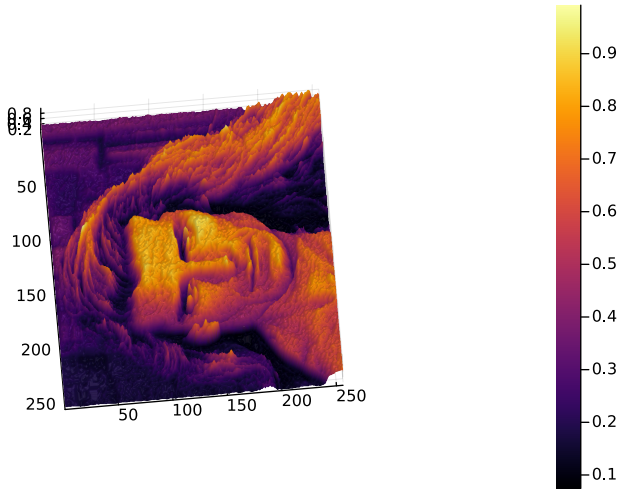

```
heatmap(Gray.(fabioReal))
```

When showing an image in a colorful representation it is important to take into account that there often is a mapping between the real values image and the color being shown. This *color mapping* will be discussed later in this course but you should be aware of it whenever you display images.

In the above code we once made the color mapping implicitly and once did it explicitly.

Surface View

The next possibility is to represent the image value $f(x,y) = z$ as the z value in a 3D plot, what we call a *surface plot*. This surface plot can also be colorized:



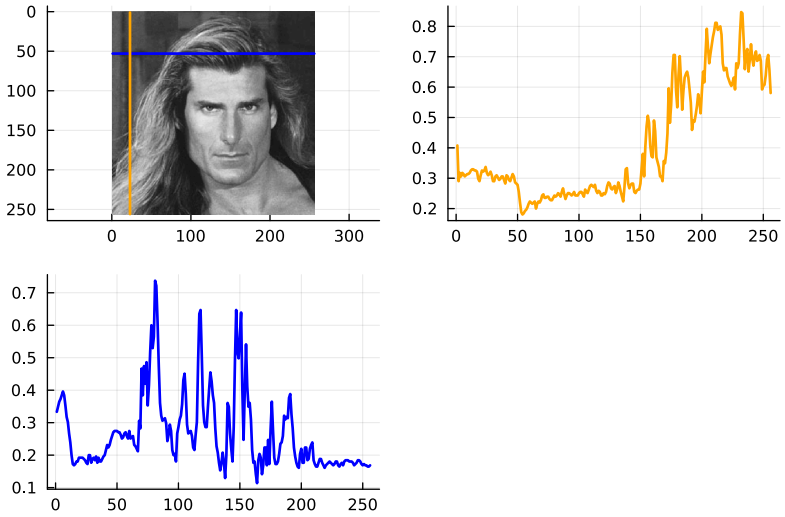
```
1 plot(fabioReal,st=:surface, cam=(85,85))
```

Profile View

Finally it is quite common to use 1D profiles of an image to showcase the progression of the underlying function along a certain direction. Often this is combined with the image view:

x = 53

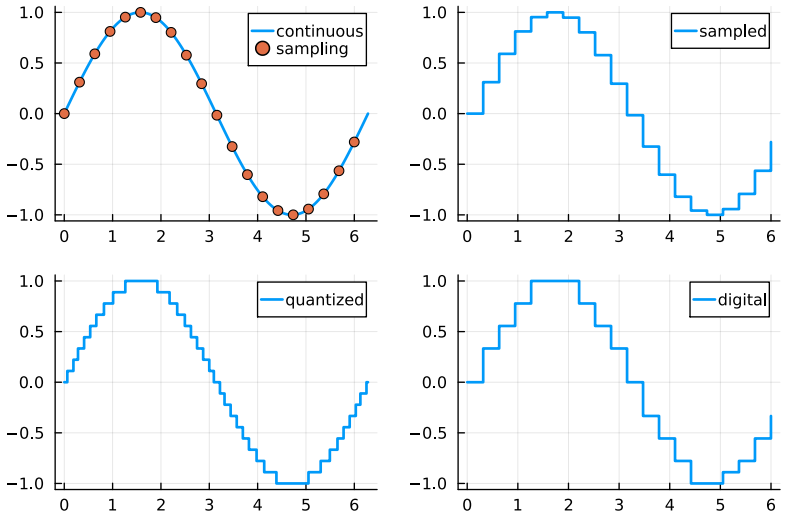
y = 23



2.4 Discretization

We next shortly review the discretization of a continuous image. We illustrate this using a 1D signal but the generalization to an image is analogous.

The following image shows how to convert an analog into a digital signal:



The discretization of the domain is called *sampling* whereas the discretization of the range is called *quantization*.

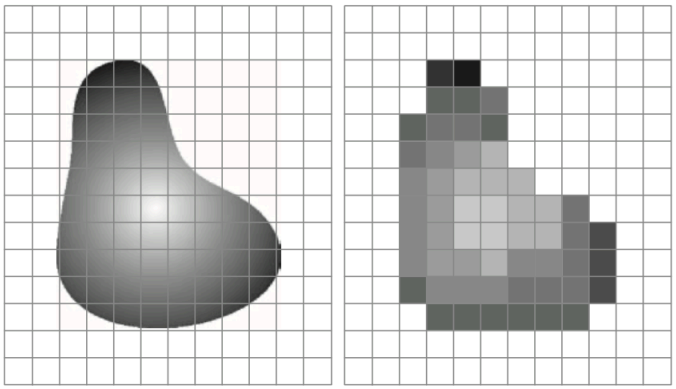
2.4.1 Sampling

Discretizing the domain of a function is called sampling. The following image shows on the left a continuous image $f(x, y)$ that is sampled at discrete positions

$$\{(x_1, y_1), \dots (x_{N_x}, y_{N_y})\}$$

i.e. the discrete image can be obtained by

$$f_{\text{discrete}}(i, j) = f(x_i, y_j) \quad \text{for} \quad i \in I_{N_x}, j \in I_{N_y}$$



Quite often, the sampling process has an *integrating* nature and thus, the sampling process is rather described by:

$$f_{\text{discrete}}(i,j) = \int_{x_i-\frac{\Delta_x}{2}}^{x_i+\frac{\Delta_x}{2}} \int_{y_i-\frac{\Delta_y}{2}}^{y_i+\frac{\Delta_y}{2}} f(x,y)\,\mathrm{d}x\,\mathrm{d}y$$

where Δ_x and Δ_y are the pixel sizes in x and y direction. Which form is used depends on the sensor being used. An integrating sensor is much more common since it maximizes the signal-to-noise ratio.

Note

Sampling needs to be done properly since otherwise one gets *artifacts* when *reconstructing* $f(x,y)$ from $f_{\text{discrete}}(i,j)$. We cover this topic later in this lecture and here for now just show you an image what happens if this is not done properly.

The following picture shows an aliasing artifact named the moire effect.

2.4.2 Quantization

Quantization means that a certain number of bits is chosen to store a real-valued variable. In practice there are two different forms.

Floating Point Values

When considering floating point values the quantization is done implicitly by the computer and one hardly needs to think about it. Floating point values allow for covering a huge dynamic range by storing mantissa and exponent individually. When choosing both large enough one can usually ignore rounding errors and can treat floating point numbers as real numbers. CPUs have special units to perform floating point calculations very efficiently.

➔ Floating point values such as `Float64` or `Float32` are usually used when *processing* images.

The downside of floating point values is that they require more bits than integer values. Thus, they are usually not used for storing images.

Integer Values

Integer values cover a range of 2^k where k is the number of bits. Since computers use byte addresses, k is usually a multiple of 8. In practice 8bit (one byte), 16bit (two bytes) and 32bit (four bytes) integers are used.

Note

At this point we primary consider *unsigned* integers that cover the range $0,\ldots,2^k-1$. When doing calculations one would rather use *signed integers* that cover a range $-2^{k-1},\ldots,2^{k-1}-1$. The reason is that subtractions can easily lead to surprising overflows when using unsigned integers. However, nowadays floating point units are fast enough such that usually integer arithmetic is not used at all.

Scaling/Offset

At some point one often wants to convert an integer value back to a floating point value representing the original range. Therefore, one often represents an image as:

$$f(x,y) = \alpha f_{\text{normalized}}(x,y) + \beta$$

where $f_{\text{normalized}} \in [0,1]$ are the normalized image values,

$$\beta = \min_{x,y}(f(x,y))$$

is the offset, and

$$\alpha = \max_{x,y}(f(x,y)) - \min_{x,y}(f(x,y))$$

is the scaling parameter.

The scaling/offset parameters are global values that are stored only once for the entire image.

The normalized image can be calculated by

$$f_{\text{normalized}}(x,y) = \frac{f(x,y)-\beta}{\alpha}.$$

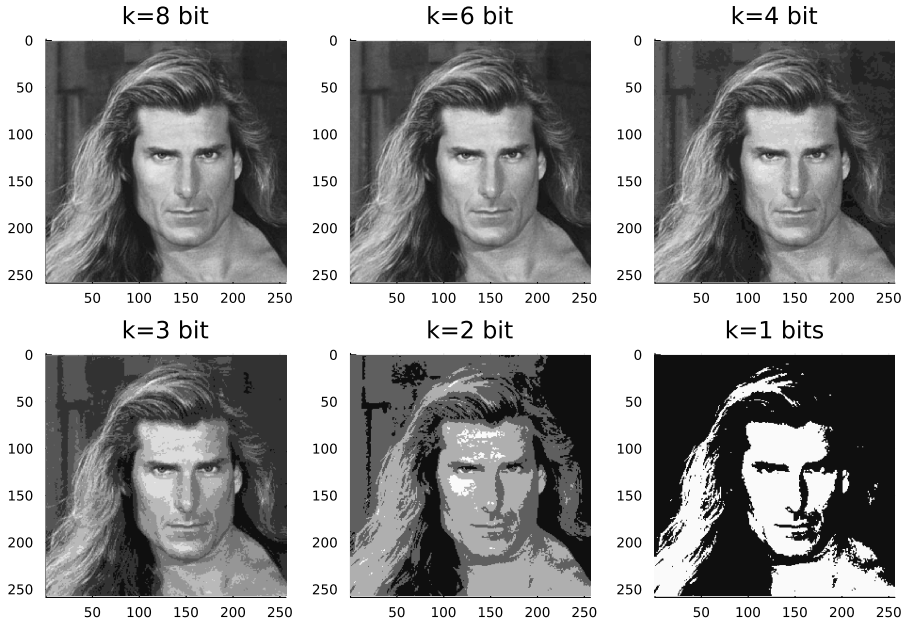
Performing Quantization

The quantization can be done in the following steps:

1. Calculate α and β.
2. Calculate $f_{\text{normalized}}(x,y)$
3. Calculate $f_{\text{integer}}(x,y) = \text{round}((2^k-1)f_{\text{normalized}}(x,y))$

Example

Let us have look at what quantization implies in practice. Next you can see a quantization of an image with different numbers of bits:



Observations:

- One can hardly see a difference between 8 and 6 bits.
- When using 4 and less bits one can clearly see quantization effects, which manifest themselves into larger regions having the same value.

2.5 Image Characteristics

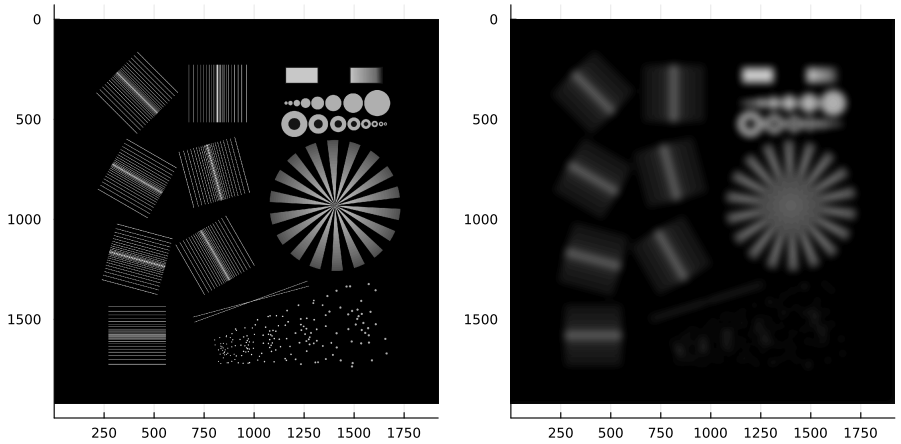
Images can be characterized using different metric that are discussed next.

2.5.1 Spatial Resolution

The resolution determines the number of details that can be distinguished. We differentiate between

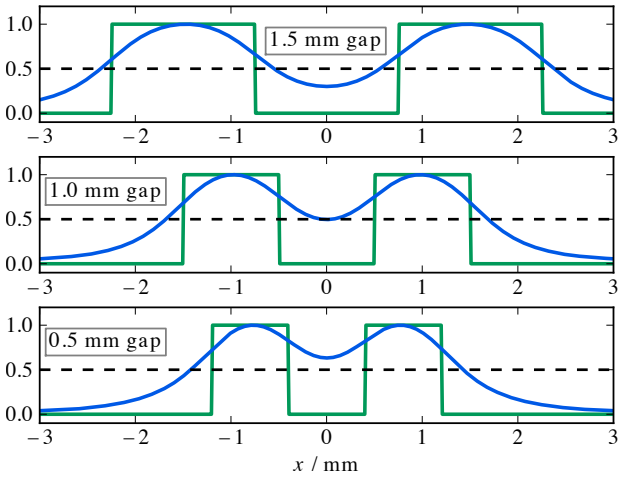
- The pixel resolution that is determined by N_x and N_y.
- The physical resolution of the underlying function $f(x, y)$ which can be lower (or higher) because of the physics during the acquisition process.

The following shows two images both having a pixel resolution of 1920×1920. The left has a full spatial resolution, the right is blurred and in turn the spatial resolution is much lower.



Physical Resolution

The (spatial) resolution is a measure defined either as a length or a spatial frequency. One can determine/define the resolution using the distance of two dots/lines that move to each other with a certain gap that has the same width as each dot. If the function value at the gap is less than 50% of the value at the dots, the dots count as resolved. This is illustrated in the next image where the green line shows the original dots with infinite resolution and the blue line shows the function after performing acquisition with a certain resolution. One can see that the spatial resolution is about 1 mm.



For further reading you can have a look at this wikipedia article on optical resolution, which is more tailored towards the resolution properties of the optical elements before digitization.

Image Resolution

(see also the wikipedia article)

The image resolution is given by the size of an image pixel. If the $N_x \times N_y$ image is associated with a certain image size (width w and height h) the resolution can be also reported as the pixel size

$$\Delta_x = \frac{h}{N_x}, \quad \Delta_w = \frac{w}{N_y}$$

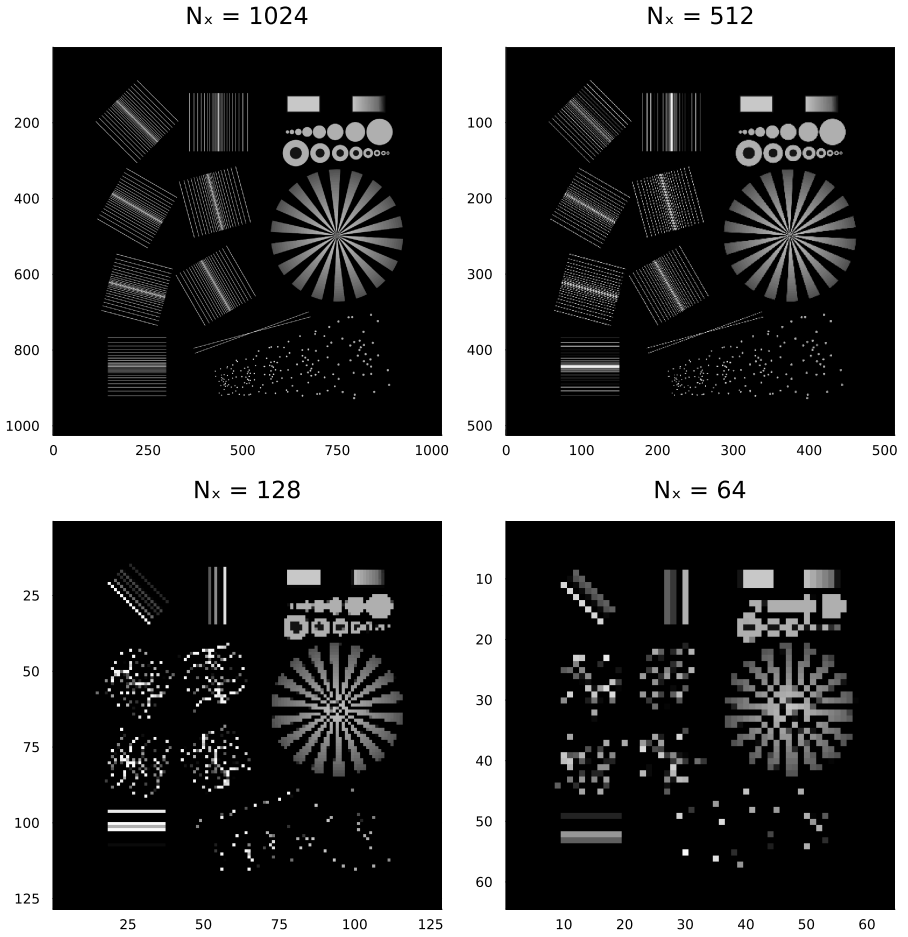
Quite often the reciprocal is reported as *lines per millimeter*, *line pairs per millimeter* or pixels per inch (ppi).

The term *resolution* is also often used for the number of pixels in the image, i.e. $N_x \times N_y$. This is of course just a simplification and requires the image size to translate this to a (dimensional) resolution.

Example

→ A typical movie in what is called *Full HD* has 1920×1080 pixels. Its image resolution cannot be stated because it depends on the output device.
→ An iphone 11 has an LCD screen with 1792×828 pixels, an image resolution of 326 ppi (12.834646 pixel/mm, 77.9141 µm pixel size), a screen size of (139.622 mm × 64.5129 mm) and an associated diagonal of 153.80 mm.

The following images show what happens if we decrease the image size:



One can see two effects:

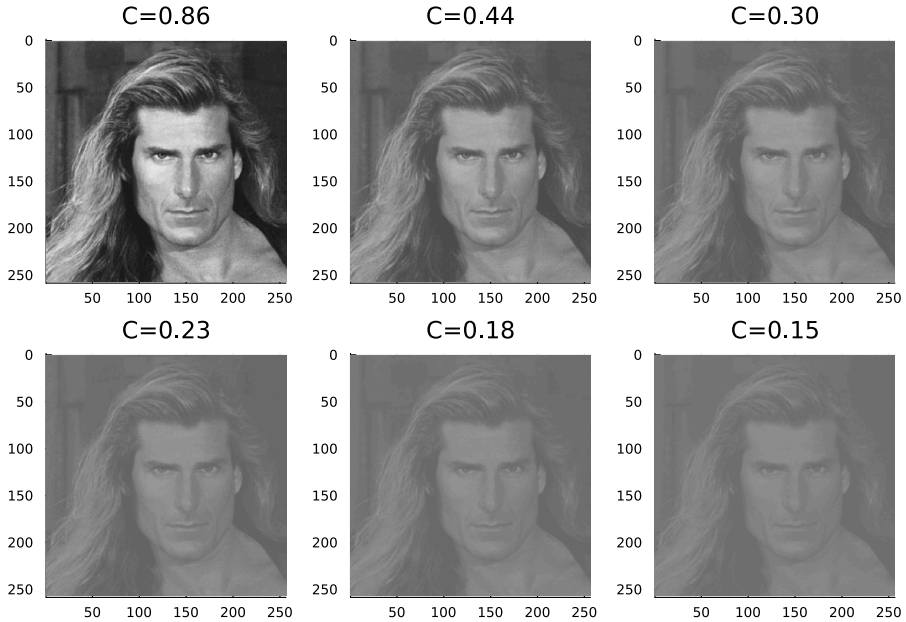
- Fine details cannot be shown anymore if the number of used pixels is too low.
- Without additional band limitation we obtain aliasing errors (discussed in upcoming lectures).

2.5.2 Contrast

The contrast of an image can be defined in different ways. It is quite common to use the *michelson contrast* that is defined as

$$C(f) = \frac{\max(f) - \min(f)}{\max(f) + \min(f)}$$

It basically measures how large the maximum variation in the image intensity is and relates that to twice of the mean value. Let us define it and have a look at the contrast of some images:



One can clearly see how it gets more difficult to distinguish parts of the image when the image contrast gets lower.

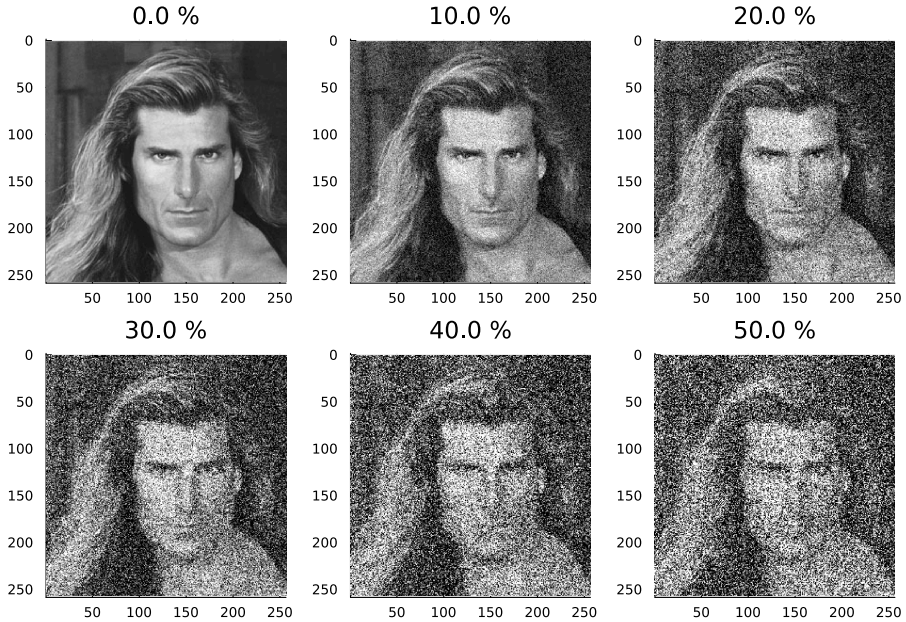
2.5.3 Noise

Often the image is contaminated by noise. Mathematically this can be modelled by:

$$f(\mathbf{r}) = f_{\text{true}}(\mathbf{r}) + \varepsilon(\mathbf{r})$$

where $f_{\text{true}}(\mathbf{r})$ is the true noise free image and $\varepsilon(\mathbf{r})$ is a noise image that (usually) contains in each pixel an uncorrelated random number with a certain mean and standard deviation.

The following pictures show the Fabio image with different amounts of noise being added. The standard deviation is given in percentage of the maximum value in the true image.



One can see how the noise degrades the overall image quality and makes it difficult to see the underlying image.

Note

The image characteristics resolution, contrast and noise are mostly independent of each other. But it is still much more challenging to inspect high resolution features in a noisy image. The contrast is usually increased when adding noise. But this increase in contrast does not match our visual impression and is thus somewhat artificial.

2.5.4 Intensity Resolution

Similar to the spatial resolution we can also define a resolution for the image intensity. This is named intensity resolution and specified by the bin size that is captured when changing the least significant bit. If the range is normalized the intensity resolution can also be given as the number of bits used for discrimination.

Note

Similar to that spatial resolution, an can image not always resolve the entire intensity resolution. There are two possible causes:
→ When storing an image with a higher bit depth one cannot resolve more intensities.
→ Most images contain a certain amount of noise. If this noise is larger than the bit resolution,

the noise limits the intensity resolution. Simple example is if you take a picture with your still camera in a dark room.

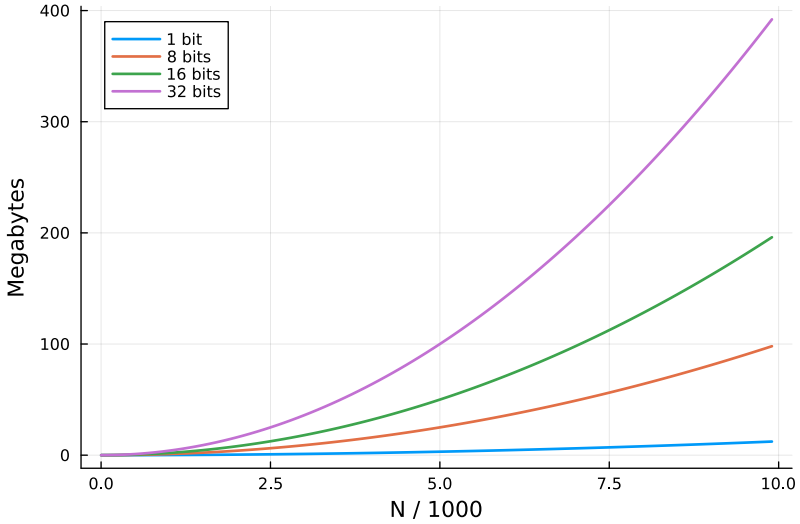
2.6 Memory Footprint

In uncompressed form an image requires to store

$$b = N_x N_y k$$

bits or $b/8$ bytes. An image data structure also needs to store the size (N_x, N_y) which can be stored in a *header*. Such a header is also used in image file formats. Consequently, the actual file size is usually slightly larger than in the above formula.

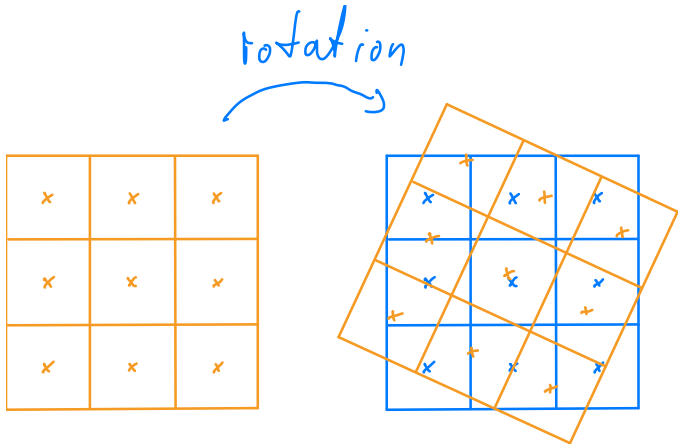
Here is a graph showing the number of bytes required to store an image depending on the image size $N = N_x = N_y$:



Note that 1 bit is often used to store text images (so-called binary images), 8 bits is a typical value for regular grayscale images and 4 bytes is typical for colored images (one byte for red, green, blue and alpha).

3. Image Interpolation

A fundamental operation that is required in image processing is (image) interpolation. As an example we consider the task of rotating an image and evaluating it on the original grid:



In such a situation we wish that the image would be continuous and could be evaluated at offgrid positions. And this is exactly what interpolation does.

Interpolation means that we take a discrete image $f_{\text{discrete}}(i, j)$ of size $N_x \times N_y$ that is available at positions $i \in I_{N_x}, j \in I_{N_y}$, and aim to evaluate this image at non-integer coordinates.

The most common method for interpolation is spline interpolation. Spline interpolation uses low-degree polynomials through given points, which can be evaluated at offgrid positions afterwards.

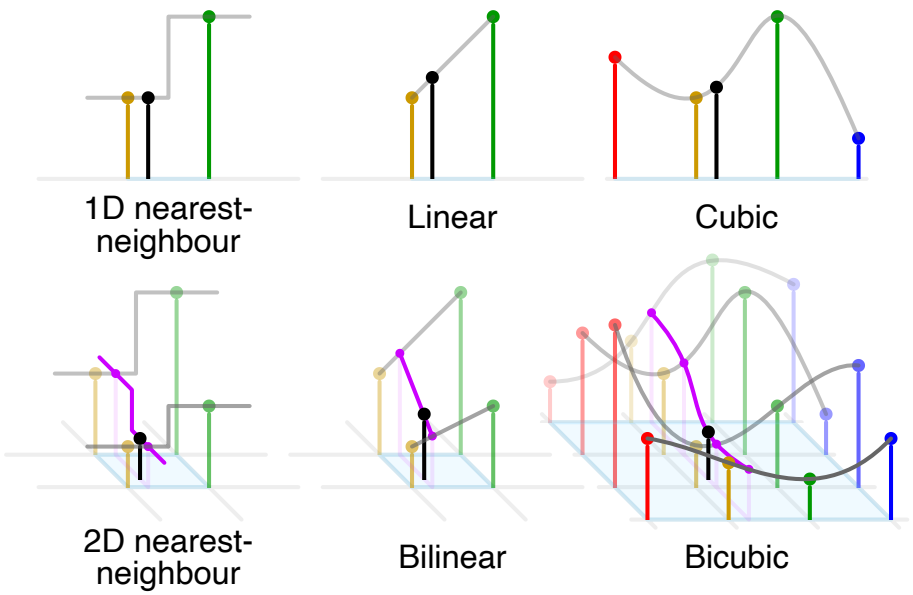
The interpolating function f_{interp} is then a function $f_{\text{interp}}(i, j) : [1, N_x] \times [1, N_y] \to \Gamma$.

Note

We name it *interpolation* if we get the exact function values back when evaluating f_{interp} at the indices (i, j), i.e. $f_{\text{interp}}(i, j) = f_{\text{discrete}}(i, j)$. Otherwise we call it *approximation*. Interpolation is usually done when the given data can be fully trusted. Approximation is done when the data has uncertainties since interpolation would then try to follow the progression of noise, which is undesired.

The difference between regular polynomial interpolation and spline interpolation is that the former uses a global polynomial with a high degree while the latter uses multiple low-degree polynomials in short intervals and chooses the polynomial pieces such that they fit smoothly together. Spline interpolation is much more common since it avoids large oscillations, which connot be avoided in regular polynomial interpolation.

The next image shows some 1D and 2D spline interpolations for polynomial degrees 0, 1 and 3:



Nearest-Neighbor

This is the most simple interpolation scheme, where the nearest point in space is taken as the function value. This is equivalent to setting up rectangular function around each pixel which has exactly the width of the pixel.

Linear

In linear interpolation one takes the neighborhood into account and takes a linear function that goes through neighboring pixels. In 1D this is two points while in 2D the interpolation takes $2 \times 2 = 4$ points.

The interpolation function for the area between the pixels (i, j) and $(i+1, j+1)$ is expressed as

$$f^{i,j}_{\text{linear}}(x, y) = a^{i,j}x + b^{i,j}y + c^{i,j}xy + d^{i,j}$$

where the coefficients $a^{i,j}$ to $d^{i,j}$ depend on the values $f(i, j)$, $f(i+1, j)$, $f(i, j+1)$, $f(i+1, j+1)$. Here, we assume that the function $f^{i,j}_{\text{linear}}(x, y)$ is shifted such that

$$f^{i,j}_{\text{linear}}(0, 0) = f_{\text{discrete}}(i, j).$$

Cubic

We skip quadratic interpolation since one usually directly goes to cubic interpolation when aiming for a more smooth transition between the points.

The issue with linear interpolation is that it is not differentiable at the grid points, which leads to visible interpolation artifacts. Higher degree polynomials do not suffer from this problem.

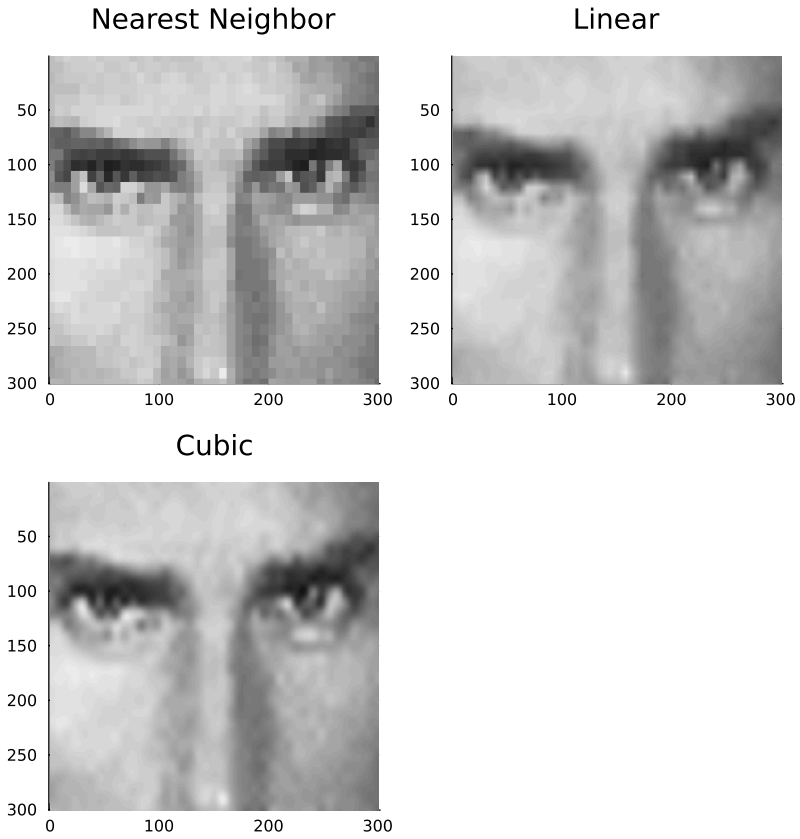
In cubic interpolation one needs to take even more neighbors, i.e. in 2D $4 \times 4 = 16$ points are required. The interpolation function then can be expressed as

$$f^{i,j}_{\text{linear}}(x, y) = \sum_{l=0}^{3}\sum_{k=0}^{3} a^{i,j}_{l,k} x^l y^k$$

Again, the coefficients depend on the local neighborhood. In cubic spline interpolation the local 4×4 patches are overlapping and they are chosen in such a way that the transition is smooth (i.e. twice differentiable).

Examples

Here are some examples of using different interpolation techniques, which are available in the *Interpolations.jl* package.



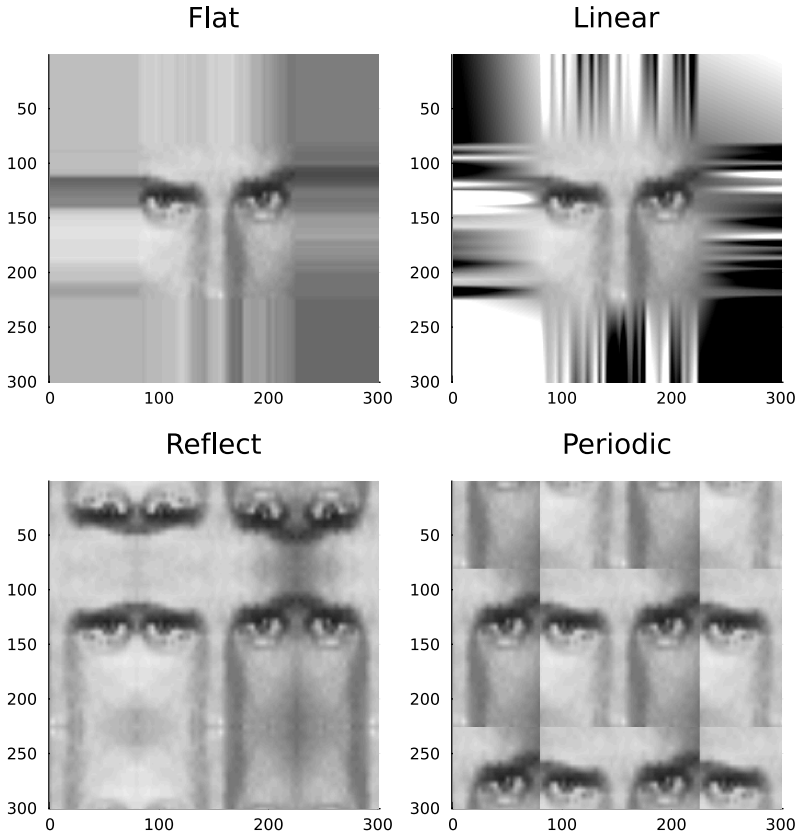
Boundary Conditions

For polynomial degrees larger than one, the boundary pixels have to be handled differently. There are different so-called boundary conditions that can be applied. Usually one *extrapolates* the missing pixels and then applies the regular interpolation scheme.

Here are some common extrapolation strategies:

- Flat: Take a constant value from the boundary and keep it constant when going outside.
- Linear: Calculate the derivative at the boundaries and let the value increase linearly.

- Reflect: Reflect the entire image and put the reflected images behind all boundaries. Behind the corners one needs to reflect twice.
- Periodic: Assume that the image is periodic and simply wrap around the indices.



4. Image Transformations

In image processing we usually take an input image $f(x, y)$ and process it resulting to an image $g(x, y)$. This is called *transformation*, *operator* (mathematics), or *system* (signal processing).

Mathematically the transformation can be defined as a function operating on images:

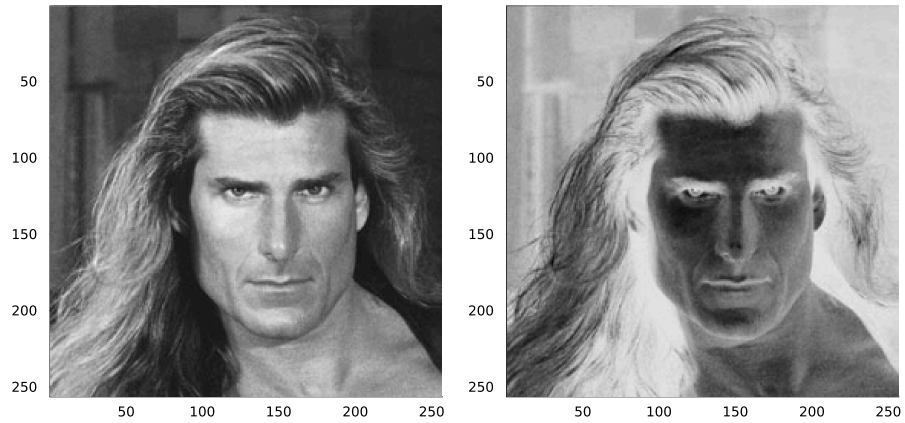
$$T : (\Omega_1 \to \Gamma_1) \to (\Omega_2 \to \Gamma_2)$$

Example

The operator

$$T_{\text{invert}}(f(x, y)) = \max(f(x, y)) - f(x, y)$$

inverts the intensity range.



4.1 Linear Transformations

A transformation T is linear if

$$T(\alpha f(x, y) + g(x, y)) = \alpha T(f(x, y)) + T(g(x, y))$$

In the discrete case, linear transformations can be expressed as

$$T(f(x, y)) = g(u, v) = \sum_{x=1}^{N_x} \sum_{y=1}^{N_y} f(x, y) s(x, y, u, v) \quad u \in I_{N_x}, v \in I_{N_y}$$

where $s(x, y, u, v)$ is the so-called *(forward) transformation kernel*.

Inverse Transformation

In many cases there exists an inverse transformation T^{-1} with $T^{-1}(T(f(x, y))) = f(x, y)$ that can be expressed as

$$T^{-1}(g(u, v)) = f(x, y) = \sum_{u=1}^{N_x} \sum_{v=1}^{N_y} g(u, v) r(x, y, u, v) \quad u \in I_{N_x}, v \in I_{N_y}$$

The function $r(x, y, u, v)$ is called the *inverse transformation kernel*.

Separable Kernel

An integral kernel is said to be separable if it can be expressed as

$$s(x, y, u, v) = s_x(x, u) s_y(y, v)$$

If $s_x = s_y$ the kernel is said to be symmetric. For separable kernels the transformation can be rearranged as

$$T(f(x,y))(u,v) = \sum_{y=1}^{N_y} s_y(y,v) \left(\sum_{x=1}^{N_x} f(x,y) s_x(x,u) \right) \quad u \in I_{N_x}, v \in I_{N_y}$$

Thus, the 2D transformation can be carried out by first doing N_y 1D transformations in x direction and then N_x 1D transformations in y direction. This requires $\mathcal{O}(N_x^2 N_y + N_y^2 N_x)$ operations instead of $\mathcal{O}(N_x^2 N_y^2)$ which are required for a non-separable kernel. This lowers the time complexity considerably.

4.2 Shift Invariance

A transformation is shift-invariant if

$$T(f(x - x_0, y - y_0))(u,v) = T(f(x,y))(u - x_0, v - y_0)$$

Shift invariance plays an important role in many image processing algorithms. One important characteristic of linear shift invariant operators is that one can express them as a convolution

$$T(f(x,y)) = g(u,v) = \sum_{y=1}^{N_y} \sum_{x=1}^{N_x} f(u-x, v-y) h(x,y) = (f * h)(u,v)$$

Here $h(x,y)$ is the so-called impulse response or point-spread function (PSF), convolution kernel or filter kernel.

We will discuss convolution in more depth in one of the following lectures.

> **Note**
>
> In practice the kernel h can also have a different size then the input image f. In these cases h is of size $M_x \times M_y$ and the above definitions have to be adapted accordingly.

4.3 Matrix Vector Notation

We have expressed linear transformations using summations. Alternatively one can exploit/reuse the linear algebra formalisms. To this end, we first need to treat images as vectors. This can be done by using the linear indexing discussed before, i.e. we consider $\mathbf{f} \in \mathbb{R}^N$ to be our image.

Then a linear transformation can be expressed as the matrix vector operation

$$\mathbf{T}\mathbf{f}$$

where $\mathbf{T} \in \mathbb{R}^{M \times N}$ is the system matrix.

This means that most of the time in image processing we take an image, multiply a matrix from the left, and then often proceed with another operation. A chain of D transformations can be compactly written as

$$\mathbf{T}_D \cdots \mathbf{T}_1 \mathbf{f}.$$

4.4 Fast Transformations

A regular transformation of an image with $N = N_x N_y$ pixels requires $\mathcal{O}(N^2)$ operations if we assume $M = N$. In many cases this is a high computational effort. Fortunately for many transformations, such as the Fourier transform or the Wavelet transform, there are faster algorithms that can carry out a transformation in only $\mathcal{O}(N \log N)$ operations. Pointwise transformations can be even carried out in only $\mathcal{O}(N)$ operations.

> **Note**
>
> When designing an image processing pipeline you always should take care of the computational complexity of the pipeline and try avoiding quadratic cost if possible.

5. Geometric Transformations

Some very simple but useful transformations are geometric transformations, where the intensity itself is kept as it is but instead the spatial variable is transformed. In the most general form we can express this as

$$T_{\text{geom}}(f(\mathbf{r})) = f(\varphi(\mathbf{r})) = g(\mathbf{r})$$

where $\varphi : \Omega \to \Omega$ is the coordinate transform. Often this transform has additional restrictions, i.e. φ is usually considered to be bijective.

> **Note**
>
> Since φ changes the position $\mathbf{r}$ the image f is usually evaluated at off-grid positions. Thus, performing a geometric transformation usually involves interpolation.

5.1 Affine Linear Transformations

A very important class of geometric transformations are the so-called affine linear transformations. They can be expressed as

$$\varphi(\mathbf{r}) = \mathbf{A}\mathbf{r} + \mathbf{b}$$

where $\mathbf{A} \in \mathbb{R}^{2\times 2}$ and $\mathbf{b} \in \mathbb{R}^2$. Here are some typical transformations

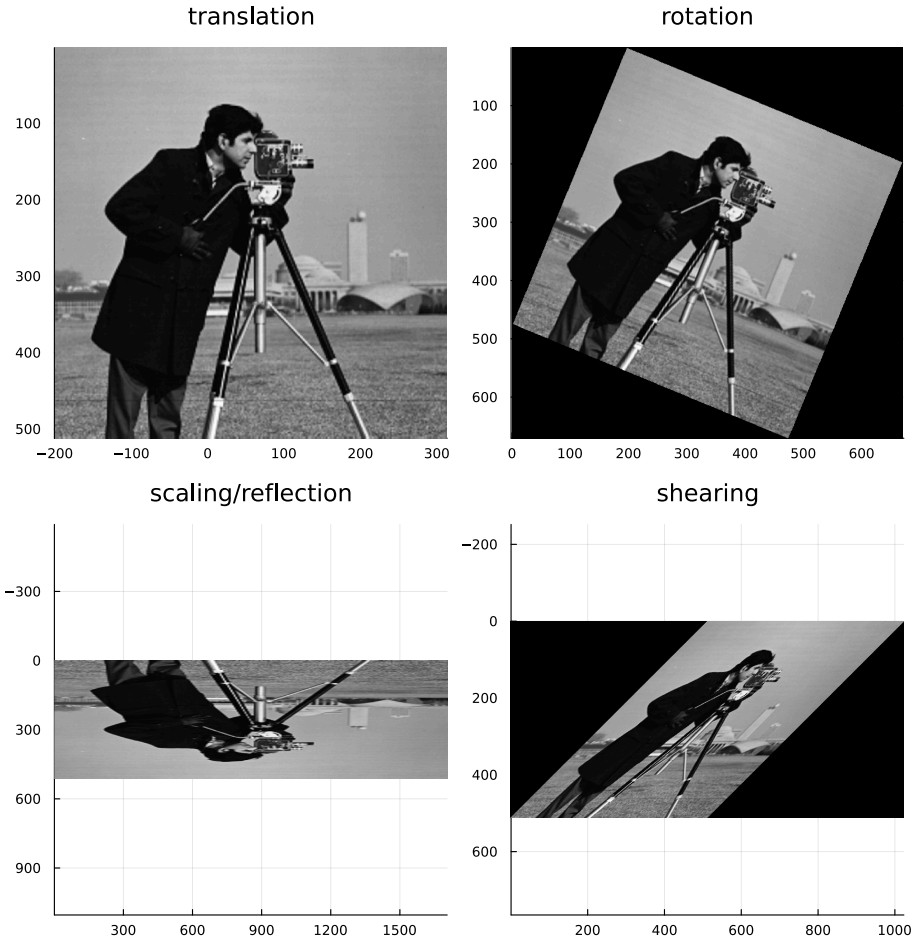
Name	$\mathbf{A}$	$\mathbf{b}$
Translation	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} t_x \\ t_y \end{pmatrix}$
Scaling / Reflection	$\begin{pmatrix} c_x & 0 \\ 0 & c_y \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$
Rotation (clockwise)	$\begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$
Shearing	$\begin{pmatrix} 1 & s_v \\ 0 & 1 \end{pmatrix}$ or $\begin{pmatrix} 1 & 0 \\ s_h & 1 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$

One can combine multiple transformation my multiplying the transformation matrices $\mathbf{A}_1\mathbf{A}_2\ldots\mathbf{A}_d$. The rightmost transformation is applied first to the spatial variable.

> **Note**
>
> One can embed the 2×2 matrix and the translation into a 3×3 matrix and use so-called homogeneous coordinates. This has the advantage that the translation needs not to be handled separately and the transformation becomes linear in the higher dimensional space.

Examples

Here are some examples of standard affine linear transformations using the ImageTransformations.jl and the CoordinateTransformations.jl packages:



6. Wrapup

In this lecture you have learned how images are represented mathematically as functions and what the typical characteristics of image functions are. In addition you have learned how images can be processed using transformations.