3D Visual Reconstruction: A Practical Approach

Vahagn Yeghikyan

2023

Contents

Part I

Introduction to Image Processing

Chapter 1

Understanding Digital Images

1.1 Introduction to Digital Imaging

A digital image is a representation of a visual image (like a photograph or scene) in the form of a two-dimensional array composed of pixels. Each pixel, or picture element, in this array contains information about the brightness and color at a specific point in the image. Typically, each pixel represents 1 (grayscale), 3 (rgb) or 4 (rgba) values.

In digital imaging, a standard coordinate system is used. The origin (0,0) is located at the top-left corner of the image. The x-coordinate increases horizontally to the right, and the y-coordinate increases vertically downwards. This coordinate system is essential for accessing and manipulating pixels within an image.

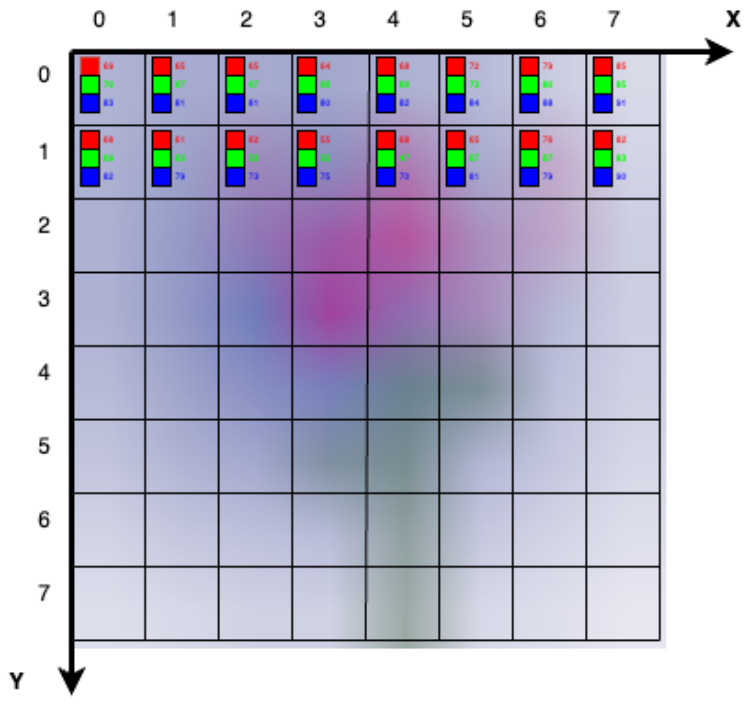

Figure 1.1: A simple 2D representation of a digital image showing its coordinate system.

Throughout this book, we will consider pixel components values encoded in 8 bits, hence, ranging from 0 (no color) to 255 (max color). There are sensors that produce higher color resolution that are encoded into 12 and even 20 bits (expensive production photo-cameras), however, although crucial for certain applications, such a high color resolution is of no use for our purposes.

1.2 Understanding Image Formats and Compression

Since, throughout the book we will have practical examples that use actual images, we should understand, that storing the image data as is on the hard drive is inefficient. In the world of digital imaging, different formats are used to store images, each with its unique characteristics. Some of the most common formats include BMP, JPEG, and PNG. Understanding these formats is crucial for image processing, as the choice of format can affect the quality and size of the image.

Below are 3 most common image formats that are widely used and supported.

- **BMP (Bitmap Image File)** The BMP format, also known as bitmap, is a simple, uncompressed format native to the Windows environment. It stores color data for each pixel in the image without any compression. As a result, BMP files are typically large but maintain high image quality.
- **JPEG (Joint Photographic Experts Group)** JPEG is a widely used format for photographic images. It employs lossy compression, which reduces file size by selectively discarding less important data. While this compression can significantly reduce file sizes, it can also lead to a loss of image quality, especially at higher compression levels.
- **PNG (Portable Network Graphics)** PNG is a format designed for the web, offering lossless compression. It provides a good balance between image quality and file size and supports transparency, making it suitable for graphics and web images.

While understanding image formats and compression is vital, the implementation details of these compression algorithms are beyond the scope of this book. Our focus will primarily be on the processing and manipulation of images, regardless of their format. However, knowledge of these formats is essential when choosing the right format for storage and processing tasks in image processing and 3D reconstruction applications.

1.3 Practical Application: The Image Class

Moving from the theoretical understanding of digital images, let's delve into a practical implementation. This will enable us to understand the digital images better by playing with the actual data. In our accompanying library, we have an 'Image' class, defined in 'image.h', which encapsulates the properties and functionalities of a digital image in C++.

The 'Image' class serves as a fundamental building block in our library. It holds the image data and provides methods to access and manipulate pixels. The class is designed to be flexible and efficient, suitable for a range of image processing tasks.

Here is the code listing for the 'Image' class:

Listing 1.1: Image class implementation

```
#ifndef VISQ_IMAGE_H
#define VISQ_IMAGE_H

#include <memory> // for shared_ptr

namespace visq {

template<typename T>
class Image {
 public:
  Image(size_t width, size_t height, size_t channels)
    : width_(width), height_(height), channels_(channels),
      stride_(width * channels), offset_(0),
      data_(new uint8_t[width * height * channels]) {}
```

```
  Image(const Image<T>& other, size_t new_width, size_t new_height, size_t offs
    : width_(new_width), height_(new_height), channels_(other.channels_),
      stride_(other.stride_), offset_(offset),
      data_(other.data_) {}

  void Set(T value, size_t y, size_t x, size_t c) {
    data_.get()[(y * stride_) + (x * channels_) + c + offset_] = value;
  }

  [[nodiscard]] const T& At(size_t y, size_t x, size_t c) const  {
    return data_.get()[(y * stride_) + (x * channels_) + c + offset_];
  }

  T* Data() {
    return data_.get();
  }

  [[nodiscard]] const T* Data() const {
    return data_.get();
  }

  template<typename R>
  bool CopyTo(Image<R> &other) const;

  [[nodiscard]] bool IsContinuous() const {
    return stride_ == width_ * channels_;
  }

  // Getters
  [[nodiscard]] size_t GetWidth() const { return width_; }
  [[nodiscard]] size_t GetHeight() const { return height_; }
  [[nodiscard]] size_t GetChannels() const { return channels_; }
  [[nodiscard]] size_t GetStride() const { return stride_; }
  [[nodiscard]] size_t GetOffset() const { return offset_; }

 private:
  size_t width_;
  size_t height_;
  size_t channels_;
  size_t stride_;
  size_t offset_;
  std::shared_ptr<T[]> data_;
};

}  // namespace visq

#endif  // VISQ_IMAGE_H
```

Let us quickly review the basic properties of the class.

- **width**_ The width of the image in pixels.
- **height**_ The height of the image in pixels.
- **channels**_ The number of channels (typically 1, 3, or 4).

- **data_** A shared pointer to the pixel data itself. We will keep the pixel data in the most common - interleaved format, i.e. the first $channels\_$ bytes will represent the top left pixel components, the next $channels\_$ bytes the pixel to the right and so on until we reach the end of the top row. Then comes the second row with the exact same format.

We will speak about the $stride\_$ and $offset\_$ variables a bit later. For now, we will think about the simplest case where $offset\_ = 0$ and $stride\_ = width\_ * channels\_$.

- **Pixel Access**: The 'At' method provides direct access to the pixel values, enabling both read and write operations. An implementation of such a method can look as follows:

Listing 1.2: An implementation of the method At

```
uint8_t & Image::At(size_t y, size_t x, size_t channel) {
  return data_[y_ * width_ * channels_ + x_* channels_ + channel];
}
```

Here and there we would want, however, to access not the entire image, but a small part of it represented by a rectangle with its top-left corner located at some point $(left\_x, top\_y)$ and which has some $width$ and $height$. This is why we need the second constructor. For multiple purposes, we do not want to copy the pixel data, so we will just take the pointer to it. This, however, leads to the problem that the current implementation of the method At will not work for such sub-images. Indeed $y * width\_ * channels\_$ no more represents the offset of the row. We need to now the width of the initial image and that is what we will keep in our stride variable. We will also need the $offset\_$ variable that represents the offset of the top-left pixel of the sub-image in the data array. Hence, the modified method will read:

Listing 1.3: An implementation of the method At

```
uint8_t & Image::At(size_t y, size_t x, size_t channel) {
  return data_[offset_ + y_ * stride_ + x_* channels_ + channel];
}
```

This class forms the foundation for more advanced image processing and 3D reconstruction techniques that will be explored later in the book.

As soon as we have our 'Image' class we would want to load an actual image and explore its contents. However, as already mentioned, images are stored and distributed in special formats. For the implementation of image loading and saving, we utilize the 'stb' library, a popular set of single-file public domain libraries for C/C++. The 'stb' library provides simple and efficient solutions for handling common operations in image processing, such as reading and writing different image file formats.

Note: The 'stb' library is external to our library and must be included separately. It offers functions like 'stbi_load' and 'stbi_write_xxx' for dealing with various image formats.

The 'io.h' file in our library interfaces with the 'stb' library functions and provides a simplified and unified API for our image processing tasks. It abstracts the complexities involved in directly using the 'stb' library functions, thereby making image I/O operations more accessible and straightforward for users.

Now, we are ready to implement our first application. The following program will load an image and print the values of the provided channel:

Listing 1.4: A simple application that prints some pixel values

```
#include <iostream>
#include "image.h"
#include "io.h"

int main(int argc, char *argv[]){

  char * imname = argv[1];
  size_t channel, print_square_size;
```

```
  char * error;
  channel = strtol(argv[2], &error);

  if(error){
    std::cout << error << std::endl;
    return 1;
  }

  print_square_size = strtol(argv[3], &error);
  if(error){
    std::cout << error << std::endl;
    return 1;
  }

  auto [error_code, Image] = LoadImage(im_path);
  if(error_code != OK)
    return 1;

  // some checks here.

  for(size_t y = 0; y < print_square_size; ++y){
    for( size_t x = 0; x < print_square_size; ++x){
      std::cout << image.At(y,x,channel) << '\t';
    }
    std::cout << std::endl;
  }

  return 0
}
```

1.4 Images as Signals

Signal processing is a field that deals with the analysis, modification, and synthesis of signals. A signal, in a broad sense, is a function conveying information about the behavior or attributes of some phenomenon. In most cases signals are continuous smooth functions of time. Take, for example, the sound wave emitted by a source(say, speaker) or the distance of the car from its initial point through a trip.

In the digital world, however, we cannot possibly keep the whole signal because it will potentially require memorizing an infinite (continuum or even countable) set of pairs (t, f). Typically, in the digital world we take discrete values and store them with a certain precision (8-64bit float/integer). This process is called "sampling". Figure 1.3 shows an example of a real world signal (the solid graph line) and the sampled digital values (black circles).

It is clear, that during the sampling process we lose a huge amount of information. (In fact, in some hard-core mathematical we lose almost all the information). Should we try to recover the original signal, we will figure out that it can be done in infinitely many ways. Indeed, there are infinitely many ways to connect 2 dots with continuous functions.

In the context of digital systems, signals are typically represented in discrete form.

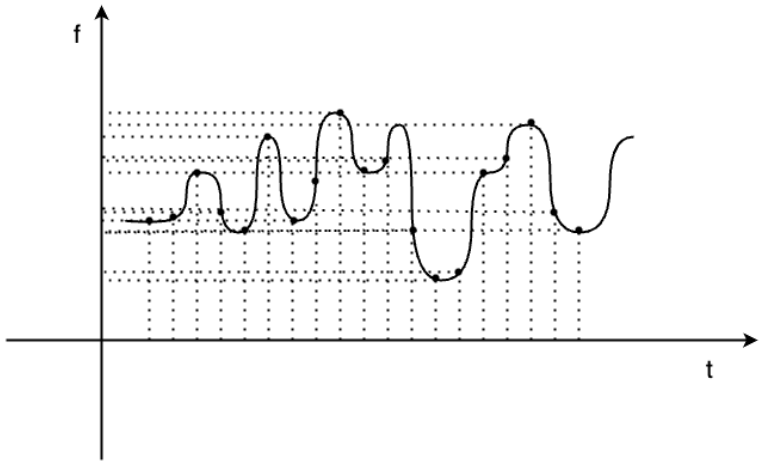

Figure 1.2: Example of signal with its discretization

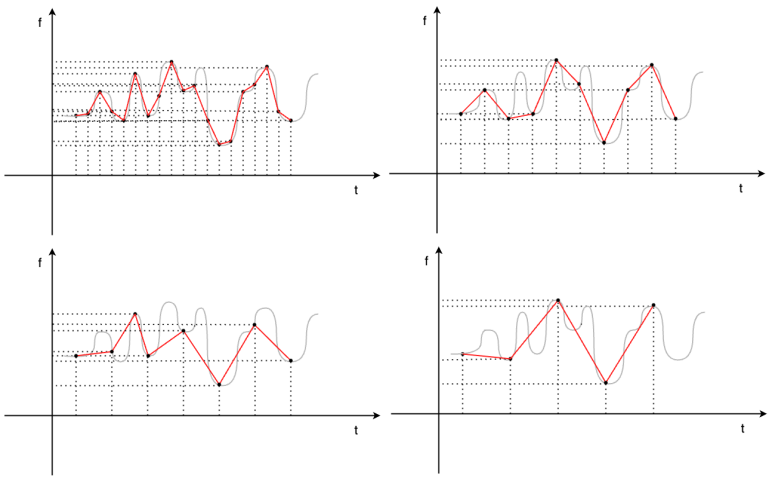

Figure 1.3: Example of signal with its discretization

As you see, although the figure composed of the black circles resembles the original signal, the details between the dots are omitted and lost. Restoring the values at any real time-point t can be done in infinitely many ways, and, strictly speaking is not a well-defined problem.

In many applications, however, this does not matter much. The most common example of this is the sound. A typical human ear can hear sounds with frequencies in range 20Hz-20kHz. So,

Signal processing techniques are fundamental in various scientific and engineering domains, including communications, audio processing, and image analysis.

Traditionally, signals are thought of as 1D functions, such as audio signals, which vary over time. However, images can be conceptualized as 2D signals. In this perspective, an image is a function that varies over two spatial dimensions, representing the intensity or color of light at each point.

This conceptualization allows us to apply signal processing techniques to image analysis. For instance, filtering, which is widely used in audio processing, can be applied to images for enhancement, noise reduction, or feature extraction. In this prospect we can think about the image as a 2D function $Im(y, x)$. Here and further in this section we will neglect the channels and consider only monochromatic images.

1.4.1 Resampling of Digital Signals

One of the most common signal processing techniques is the so-called resampling. It involves changing the sampling rate or dimensionality of a signal.

Consider a 1D signal, such as an audio waveform. Resampling might involve increasing the sampling rate (interpolation) to make the signal smoother or decreasing it (decimation) to reduce the size of the data. In both cases, we should assume that the discrete data we have, is some digitalization of a continuous signal and we should "guess" the values of the original signal in between the value with integer coordinates. We will see many examples of resampling below. A first and the most obvious example is image resize.

1.4.2 Image resize

Suppose, we want to transform an image of size $M \times N$ to the new size $M' \times N'$. The point with integer coordinates (x', y') on the resized image naturally corresponds to the point

$$x = \frac{M}{M'}x', \quad y = \frac{N}{N'}y' \tag{1.1}$$

of the original image. Notice, that x and y are not necessarily integers. We know the values of the pixels in the original image only at integer coordinates. In order to guess the value at point with arbitrary real coordinates, we should interpolate the value of the continuous function $Im(x, y)$ based on the integer sampling.

The image Fig 1.4 image illustrates resizing a 6×6 image to 5×5 pixels. The point with coordinates (1,1) corresponds to the point (1.2,1.2) in the original image.

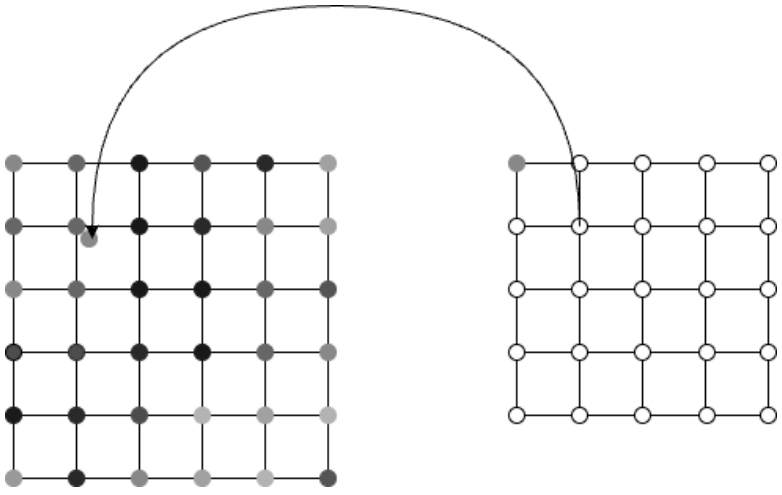

Figure 1.4: A simple 2D representation of a digital image showing its coordinate systvem.

There are various techniques for interpolating the value of the original image at non-integer points. Below we will describe some of them separately. The follwoing is an abstract interface that represents an interpolated image:

Listing 1.5: The interface for image sampler

```
namespace visq {

/*!
 * Interface that represents an image sampler.
 */
template<typename T>
class Sampler {
public:
  Sampler(const Image<T> & image): image_(image) {};
  virtual ~Sampler() = default;

  /*!
   * Returns the interpolated value of provided image volume at real point (x,y
   */
  [[nodiscard]] virtual T Interpolate(double y,
                                      double x,
                                      size_t channel) const = 0;

protected:
  const Image<T> image_;
};

}
```

Having defined this interface we can easily implement a generic resize logic as follows:

Listing 1.6: The implementation of resizer using sampler

```
#include <visq/image.h>
#include <visq/transform/sampler.h>

namespace visq {
/*!
 * @brief Resizes image from Sampler
 */
template<typename T>
class Resizer {
public:
  /*!
   * @brief Resizes the image into the provided one.11
   * @param sampler The sampler of original image.
   * @param image_to The target image to resize image to.
   */
  static void Resize(const Sampler<T> * const sampler, Image<T> * image_to) {
    double scale_x = static_cast<double>(sampler->GetImage().GetWidth()) /
image_to->GetWidth();
    double scale_y = static_cast<double>(sampler->GetImage().GetHeight()) /
image_to->GetHeight();
    for(size_t y = 0; y < image_to->GetHeight(); ++y) {
      double y_o = scale_y * y;
      for(size_t x = 0; x < image_to->GetWidth(); ++x) {
```

```
      double x_o = scale_x * x;
        for(size_t c = 0; c < image_to->GetChannels(); ++c) {
          image_to->Set(sampler->Interpolate(y_o, x_o, c), y,x,c);
        }
      }
    }
  }
};
} // namespace visq
```

Nearest-neighbor interpolation

This is the simplest and roughest interpolation. It simply returns the value of the pixel that is nearest to the point of interest.

The following is a self-explanatory implementation:

Listing 1.7: An implementation of the Nearest Neighbor interpolation

```
#include <visq/transform/sampler.h>

namespace visq {

template<typename T>
class NearestNeighborInterpolation: public Sampler<T> {
public:
  NearestNeighborInterpolation(const Image<T> & image): Sampler<T>(image) {}

  [[nodiscard]] uint8_t Interpolate(double y, double x, size_t channel) const {
    y = std::max(0., y);
    y = std::min(static_cast<double>(this->image_.GetHeight()), y);

    x = std::max(0., x);
    x = std::min(static_cast<double>(this->image_.GetWidth()), x);

    size_t y_int = std::round(y);
    size_t x_int = std::round(x);

    return this->image_.At(y_int, x_int, channel);
  }
};

}
```

Bi-linear interpolation

The idea behind this kind of interpolation is as old as the calculus itself. Assuming we have sampled a data from a 1-dimensional signal and got some number N of data-points (x_n, y_n). As the most simplistic interpolation we can simply connect the data points on the graph with lines thus filling the gaps between them.

Namely, we interpolate the original data with a set of linear functions as follows:

$$y(x) = \left\{ y_k + \frac{y_{k+1} - y_k}{x_{k+1} - x_k}(x - x_k), \quad x_k \le x \le x_{k+1} \right. \tag{1.2}$$

Fig.1.5 illustrates this.

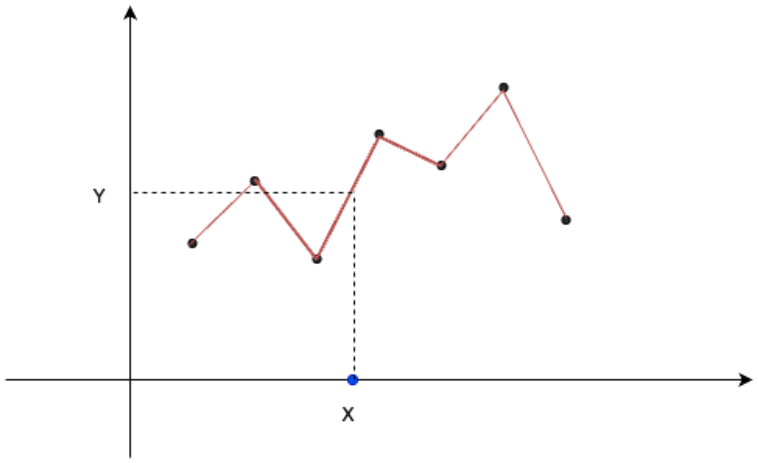

Figure 1.5: Linear interpolation

The bilinear interpolation is done in 2 steps, see Fig 1.6. First The values of the points E and F are interpolated from the points A, B and C, D respectively. Then the value of the point is interpolated from the point E and F.

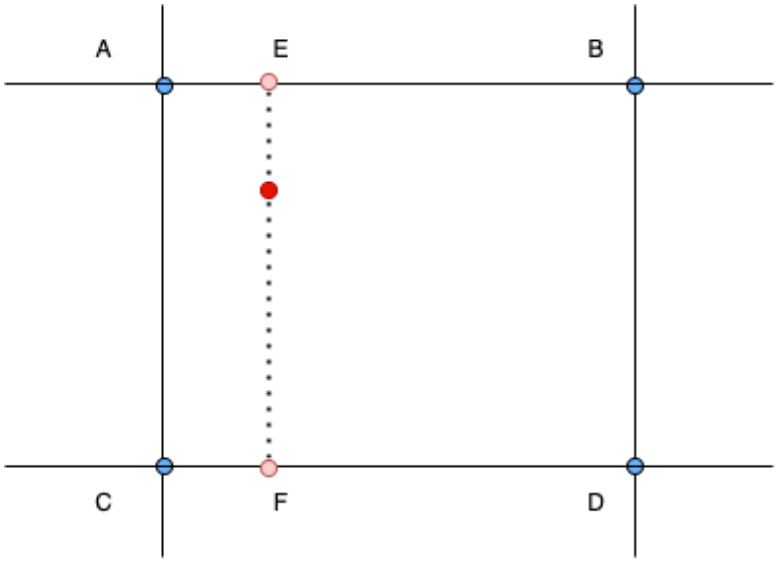

Figure 1.6: Bilinear interpolation

It is an exercise to check that the result of interpolation does not depend on the order of interpolation, i.e. we could have performed the same steps by interpolating points on the vertical axis from point A, C and B, D respectively and then perform the second interpolation between those points.

Bi-Cubic Interpolation

Cubic interpolation uses the values of neighboring points to compute a cubic polynomial for each interval between known points. In images, bicubic interpolation, which applies cubic interpolation in both dimensions, provides smoother results than linear interpolation but at a higher computational cost. ...

Higher-Order Interpolation

More sophisticated interpolation methods, such as spline interpolation or Lanczos resampling, use higher-order polynomials or specially designed kernels. These methods can provide superior results, especially for high-resolution images, but are computationally more intensive.

Choosing the Right Interpolation Method

The choice of interpolation method depends on various factors, including:

- The nature of the data or image being processed.
- The desired balance between image quality and computational efficiency.
- The specific requirements of the application, such as real-time processing constraints or the need for high-quality visual results.

Each method has its advantages and trade-offs. Nearest-neighbor interpolation, while less computationally demanding, may not be suitable for high-quality image scaling. On the other hand, methods like bicubic interpolation or Lanczos resampling, while offering higher quality, might be overkill for simple applications or when computational resources are limited.

In the following sections, we will explore these interpolation methods in more detail, particularly in the context of image processing tasks such as resizing and geometric transformations.

1.5 Fourier transform

It is impossible to underestimate the importance of Fourier transform in signal processing. Here we assume that the reader is familiar with the theory from a basic course of calculus and hence, to keep the context we are going just to copy the definitions here and highlight properties that are important for our purposes.

1.5.1 Finite case

Let us have a function $f(t)$ defined on an interval $0 \leq t \leq T$. Then, given the function satisfy certain conditions (which are presumed) it can be represented as an infinite sum of harmonic functions:

$$f(t) = \sum_{n=0}^{\infty} \left(A_n \cos \omega_n t + B_n \sin \omega_n t\right), \text{ where } \omega_n = \frac{2\pi n}{T} \tag{1.3}$$

The coefficients A_n and B_n can be computed from the initial function as follows:

$$A_n = \frac{2}{T} \int_{t=0}^{T} f(t) \cos \omega_n t dt, \quad B_n = \frac{2}{T} \int_{t=0}^{T} f(t) \sin \omega_n t dt \tag{1.4}$$

Given these integrals exist, the proof follows from the simple properties of harmonic functions

$$\int_{t=0}^{T} \cos \omega_n y \cos \omega_k t dt = \int_{t=0}^{T} \sin \omega_n y \sin \omega_k t dt = \frac{T}{2} \delta_{nk}, \quad \int_{t=0}^{T} \cos \omega_n y \sin \omega_k t dt = 0 \tag{1.5}$$

The formula (1.3) can be interpreted as any finite smooth function $f(t)$ can be represented as a superposition of monochromatic waves with certain amplitude and phase. This fact as we will see later is of crucial importance in the study of the properties of a signal. Indeed, since both expressions (1.3) and (1.4) are linear by $f(t)$ allows us in many cases to investigate the changes of well known harmonic functions and then assemble back to the to the otherwise complex function f.

The formula (1.3) can be written in complex form:

$$f(t) = \sum_{n=0}^{\infty} C_n e^{\imath \omega_n t}, \tag{1.6}$$

where complex coefficients C_n play the role of complex amplitudes:

$$C_n = \frac{1}{T} \int_{0}^{T} f(t) e^{-\imath \omega_n t} dt \tag{1.7}$$

1.5.2 Discrete case

In the digital world the signals cannot be stored continuously. In other words, of all continuum set of pairs (t, f) we reduce to a finite set of pairs (t_n, f_n). The price is that we lose a lot of information about the initial signal. We would like to be able to extend formulae (1.3) and (1.4) for this case. Before we obtain a correct modification, let us observe some peculiar effect that comes from sampling.

Aliasing

Let us consider a harmonic signal on interval $[0, T]$ sampled with N values separated with $N-1$ intervals with length T/N each. Among all possible frequencies we are interested in one of those from (1.3). In the most general case the signal has the following form:

$$f(t) = A \sin\left(\frac{2\pi n}{T} t + \varphi\right). \tag{1.8}$$

Of all points on the interval we are particularly interested in the values of the function in the sampled points

$$t_k = T\frac{k}{N}, \quad k = 0, 1 \ldots, N$$

$$f_k = \sin\left(\frac{2\pi k n}{N} + \varphi\right) \tag{1.9}$$

Let us observe, that if we restrict ourselves by only the values if the function in the sampled point, then all signals with $k = n \mod N$ are indistinguishable, i.e. 2 function with the same amplitude and phase and frequencies which differ by $2\pi m/T$, $\forall m \in \mathbb{N}$, give exactly the same values on the sampled points.

This leads us to a conclusion, that for digital signals the infinite sum in (1.3) or (1.6) can be replaced with a finite one from $n = 0$ to $n = N - 1$, with the maximal frequency being the one with $n = N - 1$. This means, that all Fourier components of a signal with frequency higher than ω_{N-1} will contribute to the coefficients of the components with $k = n \mod N$. This effect is called *aliasing*. We will talk about it in more details later.

Discrete Fourier transform

After replacing the sum in (1.6) with a little effort we arrive to the Fourier transform formulae for sampled signal f_k:

$$f_k \equiv f\left(\frac{Tk}{N}\right) = \sum_{n=0}^{N-1} f_n e^{\imath \frac{2\pi k n}{N}} \tag{1.10}$$

and

$$C_n = \frac{1}{N} \sum_{k=0}^{N-1} f_k e^{-\imath \frac{2\pi k n}{N}} \tag{1.11}$$

One can simply substitute (1.11) into (1.10) and prove that this transformation holds exactly.
Hint It is necessary to notice that

$$\sum_{n=0}^{N-1} e^{\imath \frac{2\pi k n}{N}} = 0, \quad k \in \mathbb{N}, -N < k < N \tag{1.12}$$

This is a very important result which is widely used in signal analysis.

Fast Fourier transform

The formulae (1.10) and (1.11) give us the direct way to calculate direct and inverse Fourier transformation. It is easy to see that a straightforward implementation, however, will have $O(N^2)$ complexity (N coefficients N terms in sum each). For practical applications this is very slow. Good news, however, is that it is possible to do better. There are many algorithms that can calculate the Fourier transform in $O(N \log N$. The most famous of them is the so-called Cooley-Tukey algorithm. It is based on the following observation:

$$f_k = \sum_{n=0}^{N-1} f_m e^{\imath \frac{2\pi kn}{N}} = \sum_{m=0}^{N/2-1} f_{2m} e^{\imath \frac{2\pi k 2m}{N}} + \sum_{m=0}^{N/2-1} f_{2m+1} e^{\imath \frac{2\pi k(2m+1)}{N}} \tag{1.13}$$

Factoring out a from the second term we get

$$f_k = \sum_{m=0}^{N/2-1} f_{2m} e^{\imath \frac{2\pi km}{N/2}} + e^{\imath \frac{2\pi k}{N}} \sum_{m=0}^{N/2-1} f_{2m+1} e^{\imath \frac{2\pi km}{N/2}} \tag{1.14}$$

or

$$f_k = E_k + e^{\imath \frac{2\pi k}{N}} O_k, \tag{1.15}$$

where,

$$E_k = \sum_{m=0}^{N/2-1} f_{2m} e^{\imath \frac{2\pi km}{N/2}}, \quad O_k \sum_{m=0}^{N/2-1} f_{2m+1} e^{\imath \frac{2\pi km}{N/2}} \tag{1.16}$$

For $k < N/2$, E_k and O_k represent the Fourier tranform of the subarrays consisting of only even and odd elements respectively. It is an exercise to prove that

$$f_{N/2+k} = E_k - e^{\imath \frac{2\pi k}{N}} O_k \tag{1.17}$$

and, therefore, it is enough to compute the Fourier tranform of 2 the subarrays with half the size of the original one and then perform N operations (1.15) and (1.17). Since we have $log_2(N)$ levels until we reach the end of recursion the complexity of this algorithm is $N \log_2(N)$.

Clearly, in order to be able to reach the end of recursion the initial length of the array should be a power of 2. This is a limitation of the Cooley-Tukey algorithm. However, in practice the choice of the signal length often lies on us, so we can chose it to satisfy this condition. Alternatively, we can pad the array. Other, more correct methods have been developed to overcome this issue. Because of the importance of FFT in digital signal processing, a lot of research has been done on its efficient calculation which is beyond the scope of this work.

1.6 Filters

1.7 Filters in Signal Processing and Computer Vision

1.7.1 Definition of a Filter in Signal Processing

In the context of signal processing, a *filter* is a mathematical operation or system designed to selectively enhance or suppress certain components of a signal. Typically, filters are used to modify signals based on specific criteria, such as frequency, amplitude, or phase. A filter processes an input signal to produce a desired output signal, often removing unwanted noise or emphasizing useful features.

The term *filter* originates from its analogy to physical filters used in everyday life, such as coffee filters or air filters. Just as these physical filters allow desired substances to pass through while blocking unwanted materials, signal processing filters allow desired components of a signal to pass through while attenuating or removing undesired components.

Filters are ubiquitous in various domains of signal processing. Below are a few examples unrelated to computer vision:

- **Audio Signal Processing:** Low-pass filters are used in audio systems to remove high-frequency noise, while high-pass filters can eliminate low-frequency hums.
- **Telecommunications:** Band-pass filters are employed to isolate a specific range of frequencies in a communication channel, enabling selective reception of signals.
- **Electrical Circuits:** Analog filters in electronic circuits are used to smooth voltage waveforms, such as removing ripple from a power supply.
- **Biomedical Signal Processing:** Filters are applied to ECG (electrocardiogram) data to remove noise caused by muscle movement or electrical interference.

Often filters are implemented as the so-called integral operaors. Integral operators, which underpin many filters in signal processing, arise both mathematically and from physical systems. Mathematically, integral operators are derived from the theory of convolution, a fundamental operation used to combine functions. In physical systems, integral operations naturally occur in circuits such as RC (resistor-capacitor) circuits, which integrate or differentiate input signals depending on their configuration. These circuits form the basis for early analog filters, where voltage and current behavior emulate mathematical integration.

An integraloperator applies a mathematical function, represented by a kernel, to modify or transform a signal. In continuous signal processing, this operation often takes the form of an *integral transform*, where the output signal is derived by integrating the product of the input signal and the kernel function over a specified range. Mathematically, for a signal $f(t)$ and a kernel $k(t, \tau)$, the output $g(t)$ is given by:

$$g(t) = \int_{-\infty}^{\infty} k(t, \tau) f(\tau)\, d\tau. \tag{1.18}$$

Kernel operators are essential because they provide a structured way to manipulate signals for tasks like smoothing, sharpening, and frequency filtering. By carefully designing the kernel, specific signal characteristics can be amplified or attenuated, enabling efficient analysis and processing.

The generalization of (1.18) for 2-dimensional case is straightforward:

$$g(x, y) = \int_{x'=-\infty}^{\infty} \int_{y'=-\infty}^{\infty} k(x', y') f(x, y) dx' dy'$$

An essential aspect of many integral operators is the choice of the kernel, the function used in the operation. Kernels that depend on the distance between points, such as Gaussian kernels, are particularly important because they emphasize locality. This property makes them ideal for capturing and processing spatial or temporal relationships in signals. We will focus on this property below.

In practical applications, signals are often discretized through a process called *sampling*, where the continuous signal is represented by discrete data points. When signals are sampled, integral operations are approximated by summations. For a sampled signal $f[n]$ and a discrete kernel $k[m]$, the output $g[n]$ is computed as:

$$g_n = \sum_{m=-M}^{M} k_m f_{n+m},$$

where M determines the size of the kernel.

Despite the transition from integration to summation, the principles of kernel operations remain the same: the kernel weights the contributions of neighboring values to compute the output. Summing over a small window is particularly valuable in discrete settings, as it localizes computation and enables real-time or efficient processing. Small windows often suffice for many applications, as they capture local structures or features while maintaining computational efficiency.

In computer vision, the signals are 2-dimensional. These filters operate by convolving a small, fixed-size matrix, known as a *kernel* or *filter*, over an image. This operation is called *convolution* and involves the following steps:

1. The kernel is positioned over a specific region of the image (typically, starting at the top-left corner).
2. The element-wise product of the kernel values and the corresponding image pixel intensities within the kernel's footprint is computed.
3. The resulting values are summed to produce a single output value, which is assigned to the corresponding pixel in the output image.
4. The kernel is slid (translated) across the image, repeating the above steps for every position.

Mathematically, for an image $I(x, y)$ and a kernel $K(u, v)$ of size $m \times n$, the output image $O(x, y)$ is given by:

$$O(x, y) = \sum_{u=-\lfloor m/2 \rfloor}^{\lfloor m/2 \rfloor} \sum_{v=-\lfloor n/2 \rfloor}^{\lfloor n/2 \rfloor} K(u, v) \cdot I(x + u, y + v),$$

where $\lfloor \cdot \rfloor$ denotes the floor function.

Linear sliding window filters are used to perform various operations in image processing:

- **Smoothing:** Averaging filters (e.g., mean filter) reduce noise by averaging pixel intensities within a neighborhood.
- **Edge Detection:** Filters like the Sobel or Prewitt kernels emphasize edges by computing gradients of pixel intensities.
- **Sharpening:** Laplacian filters highlight fine details and edges by emphasizing high-frequency components.

These filters are foundational in computer vision as they enable preprocessing, feature extraction, and enhancement of image data, paving the way for more complex analyses and applications.

Before passing to examples and implementation of filters, let us stop on an important property of local filters—the *multiplicative property of Fourier images*. This property states that convolution in the spatial domain corresponds to multiplication in the Fourier domain. Mathematically, for an image $I(x, y)$ and a filter kernel $K(x, y)$, their convolution $O(x, y)$ can be represented as:

$$O(x, y) = I(x, y) * K(x, y),$$

where $*$ denotes convolution. Taking the Fourier transform of both sides, we have:

$$\mathcal{F}\{O(x, y)\} = \mathcal{F}\{I(x, y)\} \cdot \mathcal{F}\{K(x, y)\}.$$

To prove the multiplicative property, we begin with the definition of convolution in two dimensions:

$$O(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(\xi, \eta) K(x - \xi, y - \eta)\, d\xi\, d\eta.$$

Taking the Fourier transform of $O(x, y)$, we apply the Fourier transform definition:

$$\mathcal{F}\{O(x, y)\}(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} O(x, y) e^{-j2\pi(ux+vy)}\, dx\, dy.$$

Substituting the expression for $O(x, y)$ into this equation:

$$\mathcal{F}\{O(x, y)\}(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \left[\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(\xi, \eta) K(x - \xi, y - \eta)\, d\xi\, d\eta \right] e^{-j2\pi(ux+vy)}\, dx\, dy.$$

Rearranging the order of integration:

$$\mathcal{F}\{O(x, y)\}(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(\xi, \eta) \left[\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} K(x - \xi, y - \eta) e^{-j2\pi(ux+vy)}\, dx\, dy \right] d\xi\, d\eta.$$

Changing variables in the inner integral by letting $x - \xi = x'$ and $y - \eta = y'$ gives:

$$\mathcal{F}\{O(x,y)\}(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} I(\xi,\eta)e^{-j2\pi(u\xi+v\eta)}\mathcal{F}\{K(x,y)\}(u,v)\,d\xi\,d\eta.$$

Recognizing the remaining integral as the Fourier transform of $I(x,y)$, we obtain:

$$\mathcal{F}\{O(x,y)\}(u,v) = \mathcal{F}\{I(x,y)\}(u,v) \cdot \mathcal{F}\{K(x,y)\}(u,v).$$

This completes the proof of the multiplicative property.

Here, $\mathcal{F}$ denotes the Fourier transform, and the operation in the Fourier domain is element-wise multiplication. This equivalence simplifies computations significantly, especially for large filters and images, as multiplication is computationally more efficient than direct convolution in the spatial domain.

This property is essential for several reasons:

- **Efficiency:** Using the Fast Fourier Transform (FFT), the Fourier transform and its inverse can be computed efficiently, making this approach practical for large-scale problems.
- **Filter Design:** Filters can be directly designed in the frequency domain to target specific frequency components, such as low-pass, high-pass, or band-pass filters.
- **Theoretical Insight:** The Fourier domain provides a clear perspective on the frequency characteristics of images and filters, aiding in the understanding and design of processing techniques.

For example, Gaussian filters, which are often used for smoothing, have a particularly simple representation in the Fourier domain. A Gaussian filter in the spatial domain corresponds to another Gaussian in the Fourier domain, emphasizing its isotropic smoothing properties across all frequencies. This makes Gaussian filters computationally and analytically attractive in both domains.

In summary, the multiplicative property of Fourier images bridges spatial and frequency domain analysis, offering both computational advantages and deeper insights into the behavior of filters.

Now let us consider some examples:

1.7.2 The Sobel Filter

The Sobel filter, introduced by Irwin Sobel and Gary Feldman in the 1960s, is one of the most widely used edge detection techniques in image processing and computer vision. It was developed during the early days of digital image processing at the Stanford Artificial Intelligence Laboratory as part of efforts to enable computers to analyze visual information effectively. The Sobel operator is foundational in the study of edge detection and has inspired numerous advancements in image analysis over the decades.

The primary goal of the Sobel filter is to highlight regions of an image where intensity changes abruptly, which often correspond to edges. Detecting edges is a critical preprocessing step in many computer vision tasks, such as object recognition, segmentation, and motion tracking.

To achieve this, the Sobel filter computes the gradient of image intensity at each pixel, which indicates the rate and direction of change in intensity. The gradient magnitude reveals the strength of the edge, while the gradient direction specifies its orientation.

The design of the Sobel operator reflects the need for:

- **Gradient Estimation:** A robust measure of intensity change across pixels.
- **Noise Resistance:** Reduced sensitivity to small variations and noise.
- **Simplicity:** Efficient computation for real-time applications.

The Sobel operator employs two convolution kernels, one for detecting horizontal edges (G_x) and another for vertical edges (G_y). These kernels are designed to emphasize changes in intensity along their respective directions while applying a slight smoothing effect to reduce noise.

The horizontal and vertical Sobel kernels are defined as:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

These kernels are convolved with the input image $I(x, y)$ to calculate the gradient components:

$$S_x(x, y) = G_x * I(x, y), \quad S_y(x, y) = G_y * I(x, y),$$

where $*$ denotes convolution. The magnitude of the gradient is given by:

$$|S(x, y)| = \sqrt{S_x(x, y)^2 + S_y(x, y)^2},$$

and the gradient direction is:

$$\theta(x, y) = \arctan\left(\frac{S_y(x, y)}{S_x(x, y)}\right).$$

In practice, the magnitude calculation is often approximated for efficiency as:

$$|S(x, y)| \approx |S_x(x, y)| + |S_y(x, y)|.$$

The Sobel filter has several notable properties:

- **Edge Localization:** It provides good localization of edges in the image.
- **Noise Reduction:** The inherent smoothing due to averaging across the kernel mitigates noise to some extent.
- **Directional Sensitivity:** Separate kernels for horizontal and vertical edges allow for detailed analysis of gradient directions.

Applications of the Sobel filter include:

- Detecting object boundaries in images.
- Preprocessing for algorithms such as Hough transform and feature extraction.
- Motion detection by analyzing gradients between consecutive frames in video sequences.

While the Sobel filter is simple and effective, it has limitations. It is sensitive to noise and does not perform well on images with low contrast or highly textured regions. Modern edge detection methods, such as the Canny edge detector, address these issues by incorporating multi-stage processes and adaptive thresholding.

Nevertheless, the Sobel filter remains a valuable tool for understanding the fundamentals of edge detection and is often used in educational contexts and as a baseline in research.

The Sobel filter stands as a cornerstone in the history of image processing. Its simplicity and effectiveness have made it a staple in computer vision, offering insights into the principles of edge detection and gradient analysis. Despite the advent of more advanced techniques, the Sobel operator continues to play a critical role in both foundational learning and practical applications.

1.7.3 The Canny Filter

The Canny filter, developed by John F. Canny in 1986, is a sophisticated edge detection algorithm that remains widely used in computer vision. Canny's work was driven by the need to formalize the criteria for optimal edge detection, which he addressed by framing the problem mathematically. His approach established a solid theoretical foundation for edge detection and has profoundly influenced subsequent advancements in image analysis.

The Canny edge detection algorithm aims to identify edges in images with precision while minimizing false positives and negatives. To achieve this, Canny proposed three main criteria for optimal edge detection:

1. **Good Detection:** The algorithm should identify as many true edges as possible.
2. **Good Localization:** The detected edges should be as close as possible to the true edges.
3. **Minimal Response:** Each edge should be marked only once, avoiding spurious or multiple detections.

To meet these criteria, the Canny filter employs a multi-stage process that integrates gradient analysis, non-maximum suppression, and hysteresis thresholding. This process balances the need for sensitivity to genuine edges with robustness against noise.

The Canny edge detection process consists of the following stages:

1. Noise Reduction Images often contain noise that can lead to false edge detection. The first step in the Canny algorithm is to apply a Gaussian filter to smooth the image:

$$I_{smooth}(x, y) = G(x, y; \sigma) * I(x, y),$$

where $G(x, y; \sigma)$ is a Gaussian kernel with standard deviation σ, and $*$ denotes convolution.

2. Gradient Computation The gradient magnitude and direction are calculated using Sobel filters:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

The gradient magnitude $|G(x, y)|$ and direction $\theta(x, y)$ are given by:

$$|G(x, y)| = \sqrt{G_x^2 + G_y^2}, \quad \theta(x, y) = \arctan\left(\frac{G_y}{G_x}\right).$$

3. Non-Maximum Suppression To thin the edges and improve localization, non-maximum suppression is applied. This step retains only the local maxima in the gradient magnitude along the gradient direction. A pixel $P(x, y)$ is preserved if its magnitude is greater than its neighbors along the gradient direction.

4. Hysteresis Thresholding To finalize edge detection, the algorithm applies two thresholds: a high threshold T_h and a low threshold T_l. Pixels with gradient magnitudes above T_h are classified as strong edges, while those between T_l and T_h are classified as weak edges. Weak edges are retained only if they are connected to strong edges, ensuring continuity and reducing noise-induced false edges.

The Canny filter is renowned for its:

- **Precision:** Effective detection of true edges with minimal noise interference.
- **Adaptability:** Adjustable thresholds and Gaussian smoothing make it versatile for various image conditions.
- **Continuity:** Hysteresis thresholding ensures connected edges, which is crucial for object segmentation and boundary tracing.

Applications of the Canny filter include:

- Object detection and segmentation.
- Feature extraction for machine learning models.
- Preprocessing in tasks like optical character recognition (OCR) and facial recognition.

Despite its strengths, the Canny filter has limitations. It can be computationally expensive due to the multi-stage process, and its performance is sensitive to parameter choices (e.g., thresholds and Gaussian smoothing). Modern edge detection methods, including machine learning-based techniques, offer greater robustness and adaptability.

Nevertheless, the Canny filter remains a cornerstone in edge detection due to its balance of simplicity, effectiveness, and theoretical grounding.

The Canny filter exemplifies the intersection of theory and practice in computer vision. Its mathematically grounded design and robust performance have made it a benchmark for edge detection, influencing generations of research and applications.

1.7.4 The Gaussian Filter

The Gaussian filter is a fundamental tool in image processing, primarily used for smoothing and noise reduction. It is named after Carl Friedrich Gauss, the mathematician who introduced the Gaussian function, also known as the normal distribution. The use of the Gaussian filter in image processing became prominent in the mid-20th century as researchers sought effective ways to reduce noise and preserve image details for better analysis. The Gaussian filter's mathematically grounded properties and computational efficiency have made it a cornerstone in image preprocessing.

The Gaussian filter is designed to smooth an image by reducing high-frequency noise while preserving important low-frequency components. This makes it particularly useful for:

- **Noise Reduction:** Removing unwanted variations in pixel intensity caused by sensor errors or environmental factors.
- **Preprocessing:** Preparing images for subsequent tasks like edge detection, segmentation, and feature extraction.
- **Blurring:** Creating a softer version of an image for artistic or technical purposes.

The Gaussian filter achieves these objectives by convolving the image with a Gaussian kernel, which ensures that the smoothing effect is isotropic (uniform in all directions) and localized.

The Gaussian function in two dimensions is defined as:

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right),$$

where x and y are spatial coordinates, and σ is the standard deviation, controlling the extent of smoothing.

The Gaussian kernel is a discretized version of this function, typically normalized so that the sum of its elements equals 1. The filter is applied to an image $I(x, y)$ using convolution:

$$I_{smooth}(x, y) = G(x, y; \sigma) * I(x, y),$$

where $*$ denotes the convolution operation. Larger values of σ result in greater smoothing.

In the frequency domain, the Gaussian filter acts as a low-pass filter. This can be understood through the Fourier transform, where the Gaussian function remains Gaussian but in an inverted scale. The Fourier transform of the Gaussian function is given by:

$$\mathcal{F}\{G(x, y; \sigma)\} = \exp\left(-2\pi^2\sigma^2(u^2 + v^2)\right),$$

where u and v are frequency components. This property ensures that high-frequency noise is effectively attenuated while preserving the essential low-frequency components of the image.

Key properties of the Gaussian filter include:

- **Isotropy:** The smoothing effect is the same in all directions.
- **Separability:** The 2D Gaussian kernel can be decomposed into two 1D kernels, reducing computational cost.
- **Smoothness:** The Gaussian function is infinitely differentiable, ensuring smooth transitions in the filtered image.

Applications of the Gaussian filter include:

- Noise suppression in medical imaging, satellite imagery, and photography.
- Preprocessing for edge detection algorithms like Sobel and Canny filters.
- Gaussian pyramids for multi-scale analysis in computer vision.

While the Gaussian filter is highly effective for smoothing, it has limitations:

- **Loss of Detail:** Excessive smoothing can blur fine details and edges.
- **Fixed Smoothing Scale:** The choice of σ affects the trade-off between noise reduction and detail preservation.
- **Not Edge-Preserving:** The Gaussian filter does not distinguish between noise and important structures like edges.

Modern techniques, such as bilateral and guided filters, address these limitations by combining smoothing with edge-preserving properties.

The Gaussian filter is a foundational tool in image processing, valued for its simplicity and effectiveness. Its mathematical properties and compatibility with Fourier analysis make it a versatile choice for noise reduction and image preprocessing. Despite its limitations, it continues to be widely used and serves as a stepping stone to more advanced filtering techniques.

1.7.5 Techniques for Applying a Convolutional Kernel to an Image

Below, we discuss three prominent techniques for applying a convolutional kernel to an image: straightforward convolution, separable convolution, and convolution using the Fourier transform. Additionally, we highlight some alternative methods.

Straightforward Convolution

The most intuitive way to apply a convolutional kernel to an image is through straightforward convolution, also known as direct convolution. This process involves:

1. Sliding the kernel over the image spatially, pixel by pixel.
2. At each position, performing an element-wise multiplication between the kernel elements and the corresponding image region elements.
3. Summing the resulting values to produce a single output value at that position.

Mathematically, for a 2D image I and a kernel K, the output O at position (i, j) is computed as:

$$O(i,j) = \sum_m \sum_n K(m,n) \cdot I(i+m, j+n), \tag{1.19}$$

where m and n iterate over the kernel dimensions.

Straightforward convolution is simple to understand and implement but can be computationally expensive, especially for large kernels and high-resolution images.

Separable Convolution

Separable convolution is an optimization technique applicable when a kernel can be decomposed into the outer product of two smaller 1D kernels. For example, a 2D kernel K of size $N \times N$ can be represented as:

$$K = k_1 \otimes k_2, \tag{1.20}$$

where k_1 and k_2 are 1D kernels of size N. The convolution operation can then be performed in two steps:

1. Convolve the image with k_1 along one dimension (e.g., rows).
2. Convolve the intermediate result with k_2 along the other dimension (e.g., columns).

This reduces the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(2N)$ for each pixel. Separable convolution is particularly efficient for kernels like Gaussian filters, which are inherently separable.

Convolution Using the Fourier Transform

The convolution theorem states that convolution in the spatial domain is equivalent to element-wise multiplication in the frequency domain. This property can be leveraged to perform convolutions efficiently, particularly for large kernels:

1. Compute the Fourier transform of the image I and the kernel K to obtain $\mathcal{F}(I)$ and $\mathcal{F}(K)$.
2. Multiply the transformed image and kernel: $\mathcal{F}(O) = \mathcal{F}(I) \cdot \mathcal{F}(K)$.
3. Compute the inverse Fourier transform of $\mathcal{F}(O)$ to obtain the convolved output O.

For an image of size $M \times M$ and a kernel of size $N \times N$, the computational complexity is dominated by the Fourier transform, which is $\mathcal{O}(M^2 \log M)$. This approach is efficient for large kernels where N is comparable to M.

Other Techniques

Several other methods for convolution exist, including:

- **Strided Convolution:** Involves skipping pixels during the convolution process to reduce the output size. This is common in deep learning for downsampling operations.
- **Dilated Convolution:** Introduces gaps between kernel elements to capture a larger receptive field without increasing kernel size.
- **Winograd Convolution:** An algorithm designed to reduce the number of multiplications, particularly useful for small kernel sizes in deep learning.
- **FFT-Based Convolution for GPUs:** Specialized for hardware accelerators, optimizing FFT operations for parallelism.

Each technique has its trade-offs in terms of computational complexity, memory requirements, and suitability for different applications. Understanding these methods allows practitioners to choose the most appropriate approach for their specific use case.

Chapter 2

Features

It is intuitively understood (and we will address this problem in the upcoming chapters) that 3D reconstruction involves determining corresponding physical points on images that are spatially or temporally shifted and then applying geometry, thus recovering the 3D coordinates of these physical points. While it would be ideal to recover all points in the scene and construct a dense depth map, it is more practical to start with a sparse set of points—the keypoints. Here is where the so-called *features* come into play.

In 3D reconstruction, *features* play a pivotal role in bridging the gap between raw visual data and meaningful geometric understanding. These features, which are distinct and repeatable patterns in images, enable the identification of corresponding points across multiple views of a scene. This correspondence is fundamental for estimating the 3D structure of a scene using techniques such as Structure-from-Motion (SfM) and Multi-View Stereo (MVS). In this section, we discuss why features such as ORB, SIFT, and SURF are crucial in the 3D reconstruction pipeline.

Features are distinctive elements in an image that are invariant or robust to various transformations such as scaling, rotation, and changes in illumination. They often include points, edges, or regions that can be reliably detected across different images of the same scene.

- **Keypoints**: Points of interest in an image, often located at corners or regions with high gradient variation.
- **Descriptors**: Numerical representations of the local image patch around a keypoint, used to match features across images.

By detecting features in images and matching them across views, it is possible to recover the spatial relationships and relative camera poses, which are essential for reconstructing a scene in three dimensions.

A typical 3D reconstruction involves several steps where features are indispensable:

1. **Feature Detection:** Algorithms such as Scale-Invariant Feature Transform (SIFT), Speeded-Up Robust Features (SURF), and Oriented FAST and Rotated BRIEF (ORB) detect keypoints in images. These keypoints serve as the foundation for establishing correspondences across multiple views.
2. **Feature Matching:** After detection, descriptors generated for the keypoints are compared across images to find matches. Robust matching ensures that corresponding points are identified correctly, even in the presence of noise, occlusions, or perspective distortions.
3. **Camera Pose Estimation:** Matched features enable the estimation of relative camera positions and orientations using techniques such as epipolar geometry. Accurate pose estimation is crucial for constructing a coherent 3D representation of the scene.
4. **3D Point Cloud Generation:** Triangulation of matched points from multiple views produces a sparse 3D point cloud, representing the scene's structure.

5. **Dense Reconstruction:** Features often guide dense matching algorithms, allowing for the generation of a detailed 3D model by interpolating or densifying the sparse point cloud.

We will delve into the details of each step deeper later. In this chapter, however, we will focus mainly on the first 2 steps.

Usually the first step can be split into 2 substeps - keypoint detection and descriptor calculation. There exist various algorithms for both substeps which can normally combined with each other depending on the needs and tradeoffs you make.

Keypoints

As previously mentioned, keypoints are points of interest in an image that are characterized by certain properties. The most common properties include strength and orientation. Strength represents the robustness of the detected keypoint, i.e., how likely it is to be detected in similar images. Orientation, on the other hand, indicates a specific approximate direction that should remain relatively consistent even after slight distortions to the image.

To represent a point let us introduce a tructure as follows:

Listing 2.1: Point2D class

```
template<typename T>
struct Point2D {
  Point2D() = default;
  Point2D(T _x, T _y) : x(_x), y(_y) {}
  T x;
  T y;

  bool operator==(const Point2D<T> & other) const{
    return x == other.x && y == other.y;
  }
};
```

Using this definition we can declare the struture for Key Points:

Listing 2.2: KeyPoint class

```
struct KeyPoint{
  geometry::Point2D<double> pt;
  double angle;
  double strength;
};
```

Having defined the Keypoint structure, we can now define an interface for Keypoint detector:

Listing 2.3: IKeypointDetector interface

```
template<typename T>
class IKeypointDetector {
 public:
  virtual ~IKeypointDetector() = default;
  virtual std::vector<KeyPoint>
        ExtractKeyPoints(const Image<T> & image) = 0;
};
```

We will implement this interface for several algorithms in the sections below.

Descriptors

Extracting keypoints alone is not very useful by itself unless we can match them between two spatially separated images. Descriptors are numerical properties of keypoints, typically represented

as vectors of some length D. Together with a distance function $L(d_1, d_2) \geq 0$, the descriptors enable us to match keypoints from 2 images.

We will discuss the matching process in more detail later. For now, to provide some intuition, consider two sets of features, $(k_i^{(1)}, d_i^{(1)})$ and $(k_i^{(2)}, d_i^{(2)})$ extracted from 2 images. During matching, we assume that the corresponding keypoint for $(k_i^{(1)}, d_i^{(1)})$ in the second image is $(k_k^{(2)}, d_k^{(2)})$, which is the closest in terms of the distance function L.

We will not introduce a new structure for storing descriptors; instead, we will reuse our existing Image class.

2.1 FAST - Features from accelerated segment test

Although FAST is not the first keypoint detection algorithm historically, we will begin with it due to its simplicity. Its computational efficiency makes it an excellent choice for applications that require high performance, such as mobile apps where resources are limited.

Consider a circle with radius 3 around a pixel as in Fig 2.1. The circle contains 16 pixels with brightness values denoted by I_i, where $i = 0, \ldots, 15$.

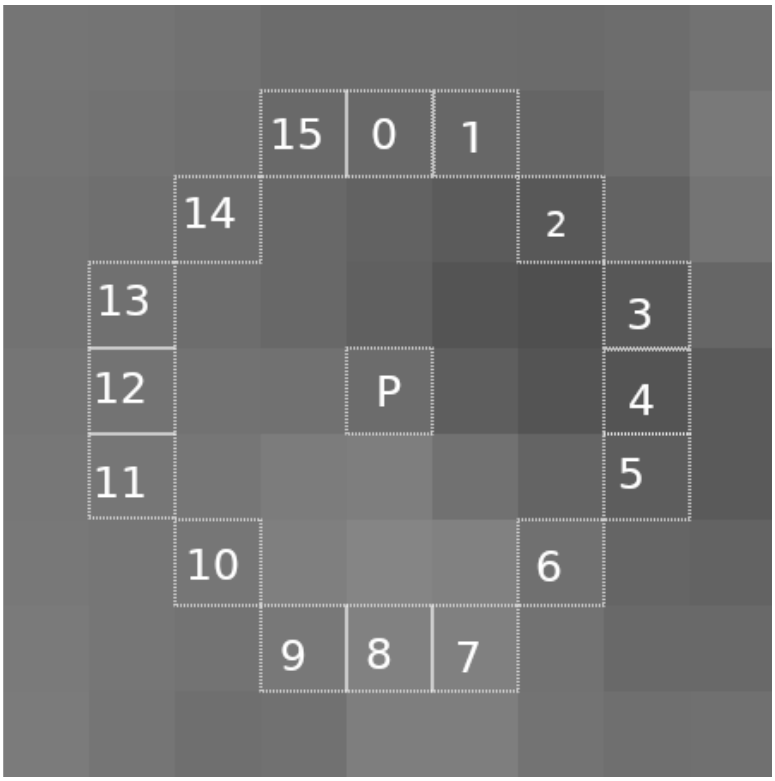

Figure 2.1: Illustration of FAST.

A pixel is selected as a *keypoint* if there are $= 12$ **sequential** pixels on the circle such that all these pixels are either brighter or darker than the central pixel.

$$I_k \geq P + t, \quad \forall k = k_0, \ldots, k_n, \qquad n \geq 12 \tag{2.1}$$

or

$$I_k \leq P + t, \quad \forall k = k_0, \ldots, k_n, \quad n \geq 12 \tag{2.2}$$

Here the parameter t is a threshold that determines the sensitivity of the keypoint detection. The choice of t depends on the specific application, the trade-offs between sensitivity and specificity, and the characteristics of the image dataset. For sharper images, a low threshold (even $t = 0$) may suffice, while for applications requiring the identification of only the most prominent keypoints, a higher threshold value can be chosen.

The strength of the keypoint is defined as the sum of absolute differences between the brightness of pixels on the chain and the central pixel's.

$$s = \sum_{k=k_0}^{n} \text{abs}\left(I_k - P\right) \tag{2.3}$$

2.1.1 Non maximum suppression