APPLIED ROBOT VISION WITH I-SEE-BYTES

İbrahim Cem Baykal

APPLIED ROBOT VISION WITH I-SEE-BYTES

İBRAHİM CEM BAYKAL

1st EDITION
April 2025

- Copyrights owned by İbrahim Cem Baykal.
- Turkish R. Culture & Turism Ministry Directorate of Copyrights Registration No: 2025/13419
- Digital copies can be distributed freely without altering.
- Printed copies can only be produced once for personal use but can not be sold/distributed.
- Cannot be quoted without citing the source.

A publication of Otoidrak Software Ltd. ISBN:

INDEX

Preface ______ 1

Chapter 1: Terminology & Philosophy ______ 3

1.1. Robot Vision ______ 7

1.2. The Philosophy of Engineering ______ 9

1.3. The Design of the I-See-Bytes ______ 11

1.3.1. High Quality GUI ______ 12

1.3.2. Direct Hardware Access ______ 13

1.3.3. Fastest, Reliable and Newest Vision Algorithms ______ 13

Chapter 2: Digital Signal Processing ______ 15

2.1. Removing Signals From Each Other ______ 16

2.2. FILTERING ______ 19

2.3. Filtering the Second Signal ______ 20

2.4. What Does That All Mean? ______ 22

2.4.1. Notation ______ 24

2.5. Filters: What are They Good for? ______ 26

2.6. Wrong Coefficients ______ 32

2.7. High-Pass Filtering ______ 34

2.8. Down Sampling ______ 35

Chapter 3: Thresholding & Segmentation ______ 37

3.1 Raw Image Types ______ 37

3.1.1 Gray level Images ______ 41

3.2 Color Thresholding ______ 41

3.3 Gray Level Thresholding ______ 45

3.4 Pixel Segmentation 48

Chapter 4: Actuator Control 51

4.1. Definition of an Actuator 51

4.2. Relay Control 51

4.3. A Simple Mobile Robot 53

4.4. A Simple Robot Vision Project 56

4.5. Robot Kinematics 59

4.6. Servo Motors 60

4.7. Controlling Servos 62

4.8. Building a Robot Arm 63

Chapter 5: Device Access 69

5.1 Querying Hardware Info 69

5.1.1. System Information Query 69

5.1.2. Querying Peripheral Devices 70

5.1.3. Querying Logical Drives 71

5.1.4. Querying COM Ports 72

5.2 Communicating with the COM Ports 73

5.3 Network Communication 75

5.4 Sound Output 78

5.5 Reading Sound Files 81

5.6 Sound Recording 82

5.7 Camera Access 83

Chapter 6: Edge Detection ____________________________ 87

6.1. Image Derivative ____________________________ 87

6.2. What is an Edge? ____________________________ 89

6.3. The Prewitt Operator ____________________________ 90

6.4. A Philosophical Question ____________________________ 94

6.5. Thresholding ____________________________ 94

Chapter 7: Spectral Transform ____________________________ 97

7.1. The Fourier Transform ____________________________ 97

7.2. The Discrete Cosine Transform ____________________________ 98

7.3. Precision Loss ____________________________ 101

7.4. DCT of 8×8 Pixels ____________________________ 103

7.5. Image Compression ____________________________ 103

Chapter 8: The Affine Transform ____________________________ 105

8.1. Transforming Coordinates ____________________________ 106

8.2. Affine Transform on Images ____________________________ 109

Chapter 9: Texture Analysis ____________________________ 113

9.1. The Cooccurrence Matrices ____________________________ 114

9.2. The Sum and Difference Histograms ____________________________ 118

Preface

Every graduate student who had to deal with the OpenCV realized that it is a horribly impractical library to write programs with. First of all, it has almost no support for the GUI widgets such as the text edit boxes, buttons, list boxes, menus etc. It doesn’t create a window frame for your application; it just shows pictures. The Python version is easier to use but it is very slow compared to the C++ version and if you don’t want to be an academician but work in the industry, you need to learn C++ because the industry uses it for real-time computer vision products and Python is too slow for that.

The fact that there are frustrating number of different objects and structures in the OpenCV library pushed me to think about this issue, and over the years, I started to design a way to keep many different variable types and data structure definitions in a compressed form. The I-See-Bytes library emerged as a result of this effort. The main object of this library, the ICBYTES class (pronounced I-See-Bytes), is used like a variable of Python. Not only can it hold int, float, double, etc. type variables but also complex data structures such as matrices and vector lists and it can alter its contents during runtime. That is why the library is called I-See-Bytes. This variable, which is called a “**fluid**” (to distinguish it from ordinary C/C++ variables) holds bunch of bytes and the way it interprets (sees) those bytes at runtime changes its contents radically. Every variable type in the I-See-Bytes library, other than the basic C++ data types are of type ICBYTES. You never need to deal with other variable types such as:

```
Mat image(Y,X, CV_8UC1);
vector<Point2f> corners;
```

The other superiority of the I-See-Bytes library is its support for hardware. The OpenCV library has support for only USB and IP cameras. On the other hand, the I-See-Bytes library has support for the sound card, servo motor controllers, relay cards, etc. You need to use another class (object) to access them: ICDEVICE. You can use this object to establish internet sockets and files as well. All of the functions in the I-See-Bytes library are smart; meaning that they can figure out what type of data structures there are in an ICBYTES type or ICDEVICE type fluid and operate accordingly. The user is not bothered by figuring out what type of input a function needs as an argument. In open CV, if you send a wrong type of data structure to a function, it crashes without telling you what went wrong. In

the I-See-Bytes, the worst that can happen is that the function returns without doing nothing and returns an error value or a message explaining what went wrong. For example, in OpenCV, in order to convert a raw image format, you need to know the input type of your image. If you don't, then you can't use the following functions:

```
cvtColor(input,output,COLOR_RGB2ARGB); //from 24 bit color depth to 32 bit
cvtColor(input, output,COLOR_ARGB2GRAY);//from 32 bit color depth to 8 bit
```

To be able to use this function, you also need to know the following table:

COLOR_BGR2BGRA	COLOR_BGR2XYZ	COLOR_YUV2RGB_NV12
COLOR_RGB2RGBA	COLOR_BGR2YCrCb	COLOR_RGB2HLS_FULL
COLOR_BGRA2RGBA	COLOR_BGR2HSV	COLOR_YUV2RGBA_NV12
COLOR_BGRA2RGBA	COLOR_RGB2Lab	COLOR_YUV2BGR_I420
COLOR_GRAY2RGBA	COLOR_RGB2Luv	COLOR_BayerRGGB2RGBA
COLOR_BGR5552RGBA	COLOR_YUV2BGR	COLOR_BayerRGGB2RGB_EA

Unfortunately, OpenCV has a very bad documentation and good luck finding this table on the web. Finding an explanation for each of them??? On the other hand, in the I-See-Bytes library, you don't need to know what type of raw image you are sending. All you care is the output. The library is smart enough to figure out the image type and employ the appropriate conversion mechanisms itself:

```
ToRGB32(input, output); //From any color depth to 32 bit color
ToRGB24(input, output); //From any color depth to 24 bit color
Luma(input, output);   // From any color depth to 8 bit grayscale color depth
```

To sum up, I-See-Bytes is a far superior library to OpenCV because of its Python like ease of use, tremendous GUI support and ability to interface with wide variety of hardware, enabling the user to write Robot Vision programs without the need to learn or include any other APIs or libraries for the hardware or the GUI interface.

Finally, the I-See-Bytes library uses the lowest level of Windows Kernel API, the WINAPI. That not only makes this library very fast, but also enables the compiled .exe programs to run on LINUX operating system (the system must have Intel compatible CPU) when used with the WINE (**W**ine **I**s **N**ot an **E**mulator) compatibility layer. At the moment, the I-See-Bytes library does not have as many different vision algorithms as the OpenCV, but given time, it will and it will contain the newest, fastest and most dependable of them.

Ibrahim Cem (Gem) BAYKAL

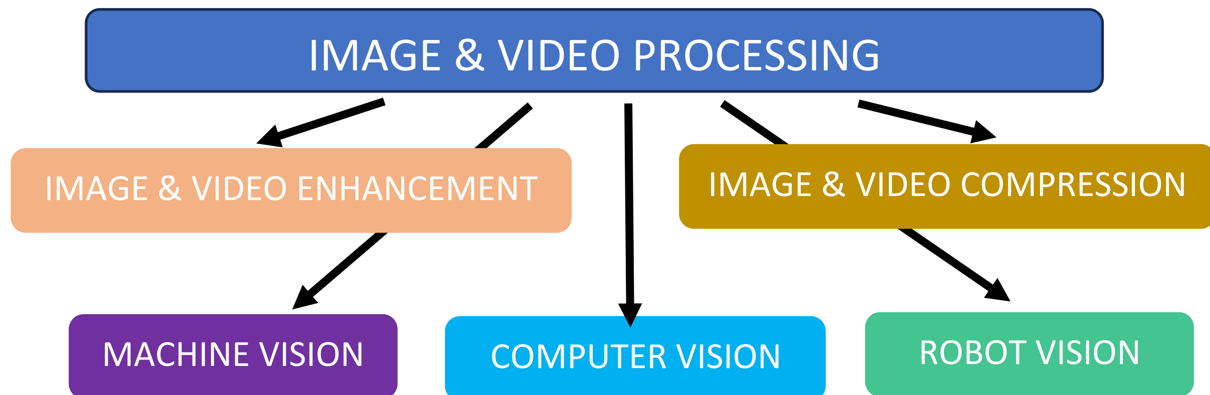
April 2025, Adana.

1. Terminology & Philosophy

Computer Vision (CV) is one of the most popular topics in the Computer Science. However, there is a confusion about what it really means and what its difference is compared to the other similar fields such as Machine Vision (MV), Image Processing and Robot Vision (RV).

Some people think that CV is the combination of Image Processing with Artificial Intelligence (AI) or Machine Learning (ML). Some people think that there is no difference between CV and MV. The main reason for this confusion is the CV researchers in academia who has never worked in MV or RV industry. Through their naivety, they think that because the same algorithms are used in these fields, they are the same. For MV experts who worked in the industry for years, distinction is clear. Unfortunately, the academia does not follow the industry. They coin their terminologies as they please and knowing that they are the real experts they expect the rest of the world to follow their lead. That is why you can see CV research articles in MV Journals.

First of all, Image and video processing is the field that generates tools for all the other related fields. Areas such as “Image Enhancement” and “Feature Extraction” are used by all the other fields. Since there is no restriction for Image Processing on the use of AI and ML, we can say that Image Processing is the name of the general field, and the rest of them are subfields of it. We may also include Feature Extraction and Pattern Recognition as subfields.



The main difference between the MV and the CV is that, basically, MV is CV with strict environmental and quality definitions. In Machine Vision, illumination conditions, the quality of the camera and even the objects that can come in front of the camera are strictly controlled. Machine vision is usually employed in industrial or commercial settings and used generally for purposes such as quality or process control. Machine vision systems have special lights that are specifically designed for a single task. The wavelength, intensity, dispersion and even the angle of the lighting is strictly controlled. On top of all these, their cameras are different too. They are called "Industrial Machine Vision Cameras" and not only have they very high quality, but also dozens of parameters that can be precisely altered by the user.

A researcher who has never used an industrial machine vision camera may not even get a proper image from an "Industrial" machine vision camera because

Figure 1.1. Three Industrial Machine Vision Cameras and a lens. From left to right: A CMOS line-scan Camera, its lens, a zoom controllable area-scan camera with integrated optics and a USB high-resolution auto-focus camera.

Figure 1.2. The back sides of the cameras shown in Figure 1.1.

of those parameters. Industrial Machine Vision cameras are so specialized that they are usually sold without lenses. The user must contact another vendor, an optics company, and purchase a specific lens that is most suitable for the application. He/she must know how to calculate even the lens parameters. On top of these, the user must know different camera technologies such as the CCD, CMOS and Time-Delay and Integration. He/she must decide whether to buy an area-scan or a line-scan camera. On the other hand, a cheap ordinary security camera is enough for a computer vision application. Figure1.1 shows three very different industrial MV cameras that are used for entirely different applications. For a seasoned MV expert, these are only a few of the types available among hundreds of cameras if not thousands.

These cameras not only have very different properties and parameters, but their connections are usually very different as well. While majority of the industrial MV cameras have USB or Gigabit ethernet connections (GigE, including 5 or 10 GigE), some cameras have so much data output that they have to be connected through different interfaces such as Camera Link or CoaXPress (CXP). Their power consumption may be too high for the USB or ethernet cable to handle, therefore, they might require extra power connections. On top of that they might require synchronization signals generated by shaft encoders or pulsed illumination sources. Such signals are vital especially for Line-Scan Time Delay Integration (TDI) cameras.

The resolution and frame rate of the machine vision cameras are nonstandard. Vendors can produce cameras with arbitrary resolutions. They can be extremely large or very small for high frame rate cameras. Frame rates can

reach over 200fps. As a result of these, they require special expertise, otherwise, the image quality can be extremely poor.

Machine vision systems are usually employed in factories where a conveyor belt constantly spews out a single product. The objects that the camera sees can be strictly controlled. In a factory where bottles are produced, there is almost no chance of a bird flying in front of your camera. Because of such high-quality demands and strict environmental controls, MV systems have surpassed the human visual perception long ago. They can see better, faster and never get tired. On the other hand, majority of the Computer Vision systems are far from reaching the level of human visual abilities.

Figure 1.3 shows typical security cameras and a USB web cam. All of these cameras have a very simplistic interface (almost plug and play) and can be used for computer vision. Many of them doesn’t even need an external power supply, the 5V through the USB or the voltage through the ethernet cable (if supports POE) is enough.

However, there is one crucial difference between these cameras and the machine vision ones that students must know: MV cameras have a fixed pixel gain while the ordinary ones have adaptive. When you first turn on an ordinary camera, the picture looks dark and it gradually gets brighter or when an ordinary camera is in the dark and you suddenly turn on the lights its picture gets ultra

Figure 1.3. Three security cameras and a webcam. From left to right: A Logitech USB webcam, a Wisenet, a Dahua and a HikVision outdoor security cameras.

bright, almost completely white and then slowly darkens and gets normal. That is because ordinary cameras adjust their pixel gains based on the ambient light in order to keep their dynamic range wide. On the other hand, MV cameras have fixed pixel gain (brightness) set by the user. Combined with precisely adjusted illumination conditions, this allows the user to employ simple pixel thresholding to detect colors. You cannot do that with an ordinary camera because its gain floats.

Some of the industrial machine vision cameras are so advanced that they can adjust the gain and bias of every individual pixel on their sensor. Combined with a color calibration process, their pixels will generate exactly the same pixel values when they see the same color. This means that, when you put a sheet of single-color paper in front of them, all of the pixel values in the image are the same number (say 197). Computer researchers who are unaware of that capability of such cameras will strongly object to using simple pixel thresholds out of their ignorance. Some of them are so fanatical about using a fixed threshold that they will accuse MV researchers of being unscientific. Using fixed thresholds are so common that some industrial MV cameras have that implemented inside them to reduce the data bandwidth. These cameras exclude the background (normal) pixels and send only the images of defective area by simply thresholding the image.

On the image processing side, MV researchers/engineers must have a better understanding of 2-D and statistical signal processing compared to CV ones. We can say that an MV engineer must know the mathematical basis of Image Processing better than the CV engineers. MV researchers are more like industrial R&D researchers rather than academical ones. They must be more hands on, and familiar with the hardware technologies. An industrial R&D engineer is more like a carpenter while an academician is more like a furniture designer. You may have graduated from the best furniture design school in Italy but that doesn't mean that you have hands on skills to build a dining table. These two skills are quite different.

1.1. Robot Vision

Having seen the difference between the Computer and Machine Vision, we can now ask what the difference of Robot Vision (RV) is. There are two main differences of this field: First of all, as the name implies, the vision is performed for a robot, meaning that the researcher or the programmer must command an

electromechanical, hydraulic, pneumatic of even organic actuator. In other words, the vision process must move a robot at the end of its completion. This requires some knowledge of hardware along with Control Theory and similar fields.

The second difference is that RV is almost exclusively performed on 3D data and results are in 3D too. A robot must always be aware of the depth dimension. Otherwise, it will crash into objects or break them. It may harm itself or people around it. This means that robot vision uses depth sensors other than cameras. Most common and popular of those sensors are LIDARs. They may also employ stereo cameras with integrated depth calculators or similar devices, along with other, simpler sensors such as ultrasonic, pressure or light reflection sensors.

WARNING: OPERATING ROBOTS MAY BE EXTREMELY DANGEROUS!

- A powerful robot must always be operated inside a metal cage where humans are not allowed to get inside. Otherwise, it may hit humans and injure them.
- A mobile robot must have fail-safe sensors that protects them from crashing into objects. Otherwise, it may crash into humans/pets and injure them.
- Remove all living beings, containers of flammable or corrosive liquids and fragile objects around the path of the mobile robots.
- Mobile robots contain explosive batteries like lithium-ion. These batteries will burst into flames spontaneously or explode when punctured. You must take necessary precautions.

THE READER BEARS ALL RESPONSIBILITY WHILE CONDUCTING ANY EXPERIMENTS USING THE PROGRAMS IN THIS BOOK OR THE I-SEE-BYTES LIBRARY.

1.2. The Philosophy of Engineering

There are thousands of engineering faculties at the universities all around the world. These faculties focus on educating their students and prepare them with skills so that they become desirable employees for engineering companies. Some of their students attend those universities because they enjoy working in those fields; they wonder how machines work, they want to know how the universe works. But the majority of them attend because they want to be employed, have regular salaries and live a comfortable life. In other words, they want to make money.

Aware of this motivation, especially those universities with high tuition fees, teach courses with an emphasis on how these courses will propel students to their desired jobs, how technologically cutting edge their education is. They do not stop for a second to consider what the philosophy behind engineering is. You would be surprised how few of the professors know what the reason behind why we spend so much time, money and energy on research and education. If you ask a professor why we do science, he or she will most probably say "to enlighten the human kind."

Enlightening the human race is definitely one of the reasons and a noble one. But it is nowhere near the main reason. The main reason we do science and engineering is **to enhance the quality of human life**. Consider the person who earns minimum wage. Governments all around the world collect taxes from these people and use it to fund scientific research and universities. Every penny, cent or kuruş of tax being collected from those people is less money that those people can spend on their kids**. In other words, the money that could have been used to feed the kids of the taxpayers is being used for the research and education at the universities**.

The lesson here is that science must be conducted to solve the humanities', or the taxpayers' problems. The duty to enlighten the public is a far less important job of a scientist. A professor must know where his/her research money is coming from and feel the responsibility.

Now that we know the philosophy behind why we conduct scientific research, let's focus on engineering. Engineering is the application of scientific knowledge in order to design things that enhance, simplify or speed up human

life. This part of the definition is common knowledge and well-known by both students and professors. What is missing in this definition, that is not being thought at most of the universities is that **engineering must be performed using minimum resources**. In other words, engineering is the application of our scientific knowledge to enhance human life by using minimum resources.

We are all aware of the devastating effects of pollution. The ecosystem of this planet cannot recover from the huge amount of toxic byproducts that our industry produces. Not only the global warming caused by excessive carbon dioxide, but also other chemicals are destroying natural life at a pace unseen in human history. On top of that, our resources are diminishing at an alarming rate. Prices of oil, copper, zinc and rare metals like antimony are rising because the reserves are decreasing while demand increases. Increasing human population and environmental decay has put stress on the amount of fresh water and agricultural production. That is why constructing designs that use less resources are important for an engineer more than ever before.

However, a less known fact among engineers is that using less resources is important for your own company and country. Imagine three car producers from Germany (yellow), Türkiye (turquoise) and China (red). Let's assume that the first two (German and Turkish) cars are being sold for 10K dollars. But the German car has superior quality to the Turkish car. It consumes less gas per kilometer (or mile) and it doesn't require parts change as often. It produces more power for the same size (liters) engine and it is safer and more comfortable. Can the turquoise car be sold? Would customers prefer the Turkish made car over the German one? The answer is a simple big NO. Customers would try to get maximum for their money. Therefore, they would buy the superior German car because they cost the same. What would happen to the Turkish manufacturer? They spent a lot of money to build a factory, they paid to the workers and purchased raw materials for the car. They need cash to turn the wheels. Obviously if the company has no income, it would go bankrupt.

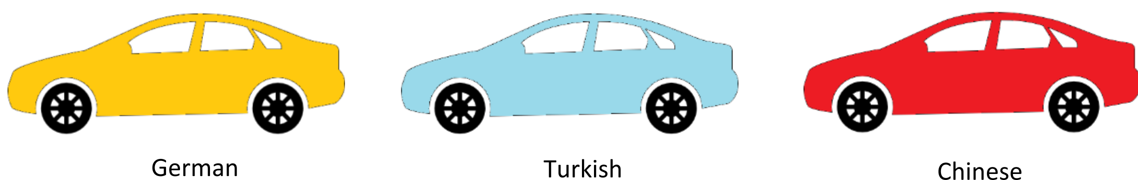


Figure 1.4. Three cars manufactured by three different car makers from Germany, Türkiye and China.

But wait. There is another solution. The Turkish government can (or be forced to) reduce the value of the Turkish Lira. If the Lira is devalued to %85 of what it was, then the Turkish car would cost 8500 dollars. Because it is cheaper, some customers would prefer the Turkish car over the German one and the Turkish car manufacturer can stay alive. Workers will get paid (less in dollars but same in Liras) and their families will not become (very) poor.

Let's consider another scenario. Let's say that the Turkish and the Chinese cars have almost the same quality but the Chinese car is being sold for 8000 dollars because it costs less. Would the customers prefer Turkish made car? Again, the answer is a big no (unless the Turkish car is called iPhone). Why would you pay more for the same product? You may ask how the Chinese can produce the same quality car for much less cost? They may be using cheap labor, or they may be using an efficient, automated production infostructure. In other words, they might be using **robots** to lower their costs.

Robots are increasingly being used in manufacturing, especially for the production of motor vehicles. Toyota is one of the leading companies in using robots, helping them become the worlds' biggest car maker. Robots do not get tired, sick or depressed. They don't strike when they aren't paid enough. They work 24 hours a day and 353 days (once a month maintenance day) a year.

1.3. The Design of the I-See-Bytes

I-See-Bytes is a Robot Vision (RV) library designed to lower the cost of not only research and development but also teaching. In order to be efficient, a Robot Vision library needs to be very fast because it processes huge amount of data in real-time. We all know that the fastest high-level programming language is C++. Unfortunately, C++ is difficult to use. That is why majority of the universities started teaching Python or Matlab as the introduction to programming courses in the engineering departments. As mentioned in the book titled "I-See-Bytes a Simplified C++ Library;" despite being a C++ library, it has many of the comforts of Python and Matlab. The main class of this library, which is called a fluid, can be used to hold any variable type and different containers / data structures such as matrixes (multidimensional C++ arrays) and lists (2D C++ STL vectors).

Unlike the OpenCV, there is a single data type (called ICBYTES) and all of the functions receive or return this same data type. This not only decreases the time spent for writing code, but also makes it easier for the instructors to teach this library to the new students. OpenCV has a Python version but because Python runs on virtual machine, a code written in Python works 10 times slower than the same code written in C++. Robot vision applications require huge amount of processing power and if using Python makes your product 10 times slower than your competitors, your company will lose its competitive edge from the start.

You may say that you will use hardware that is 10 times faster. Then your costs will go up because you will need to use 10 CPUs in your motherboard while your competitors use only one. Things get even worse if you are designing a mobile robot. A mobile robot runs on batteries and if you use 10 CPUs on your design then your batteries will run out 10 times faster. Who would want to buy a robot that is slow, expensive and runs out of battery faster than other robots?

1.3.1. High Quality GUI

Another important superiority of the I-See-Bytes library is its simplified interface to the Windows API. The Windows API (used to be called the Win32 API) is the programming interface to create and use graphical user interface (GUI) elements such as buttons, edit windows and image frames. Other libraries such as the OpenCV has little or no support for GUI making it impossible to create professional looking applications. On the other hand, because the I-See-Bytes library is designed based upon the Windows API, you can not only create professional looking GUI interfaces, but also use the Windows API (if you know how) to create your own custom interface without any compatibility issues.

Using the Windows API at the heart of the I-See-Bytes library has two other important advantages. First of all, it is very fast. The Windows API is the lowest level programming interface of the Windows operating system with most of its calls directly tied to the operating system kernel. That makes this interface considerably faster than other interfaces or GUI libraries such as the QT or GTK+. If you want to display several HD videos on the screen or 3D animations, such advantages will make your application faster than your competitors.

The third advantage of using the pure Windows API is cross platform compatibility. As we all know, when the Java language came out, they promised that the Java bytecode would be executable on any platform, Windows, Linux or MacOS. As time passed, Java failed to deliver on that promise. Fortunately, the Windows API has been around as long as the Windows has been and people had

a long time to reverse engineer it. Consequently, a famous (mostly for Linux users) compatibility layer has been written called the **WINE** (**W**ine **I**s **N**ot and **E**mulator). WINE is an opensource compatibility layer that converts Windows system calls (Windows API calls) to the host operating system. As a result, if you are using Intel family CPU and Linux or MacOS operating systems, you can run Windows applications once you install WINE on your computer. The executables you create using the I-See-Bytes library run smoothly on both Linux and MacOS (with intel CPU) and at the same speed. That is something Java cannot do.

1.3.2. Direct Hardware Access

One of the most important advantages of using the I-See-Bytes library is its ability to control the robot hardware directly. A single class called “**ICDEVICE**” (pronounced I-See-Device) can be used to access not only cameras but also relay cards, LIDARs, servo controllers, microphones and even network devices.

As we will see in the following chapters, the I-See-Bytes library can control relay cards that are used for turning on and off heavy-duty devices. You can control even a refrigerator using a relay card. If you connect the motors of a mobile robot to a relay card, then you can control the movements of that robot. Servo motors can also be controlled directly, making robot arms very easy to use. Combined with vision algorithms, ability to receive data from cameras and sensors such as LIDARs, I-See-Bytes library allows you to develop robot vision applications without the need to learn or integrate any other interface.

1.3.3. Fastest, Reliable and Newest Vision Algorithms

Every year thousands of new vision algorithms are published in scientific journals. Unfortunately, we seasoned vision scientist know that, even the algorithms published in the highest ranked journals do not work as reliably as advertised in their articles. If an algorithm works on 70 images out of 100, its inventors chose to publish those 70 images and only a few of the 30 that doesn’t work, out of the pressure to publish their articles. On top of that, article publishing business is not a moral one, that favor western universities, especially the ones from USA and Britain as these countries make a lot of money from foreign students and those students want to attend a high-ranking research university.

The other problem is that the academicians are not worried about getting their company bankrupt because they don’t work at a company. They are focused on creating algorithms that work more successfully rather than

increasing the processing efficiency of their algorithm. As a result, you see algorithms published in respectable journals that improve the accuracy of an existing algorithm only %10 while increasing the processing power consumption %300. As an engineer, if you design a system that works only %10 more accurately than your competitor but increase hardware costs %300 percent, your boss would fire you. Why? Because there are many different algorithms with completely different approaches that do the same thing. Instead of using an algorithm that increases the cost %300, it is much smarter to use two completely different, parallel algorithms as shown in the figure below and fuse both algorithms at almost half the cost.

Because those two algorithms have completely different approaches, one of them will most probably correctly process the image that the other did not. Therefore, if you have very different two algorithms with accuracies of %80 and %65, a smart decision-making algorithm based on their confidences may combine the results producing an overall accuracy of %90. The combined processing power needed would be %100 + %70 (plus some for the decision-making algorithm) equaling %180. That is why, in the I-See-Bytes library, only (mostly) the fastest and most accurate or in other words, the most efficient vision algorithms are included. **Crappy algorithms like the Hough Transform** will never be included in the I-See-Bytes library.

As we mentioned before, every year thousands of algorithms are being published. Unfortunately, it takes a long time for these algorithms to be included in opensource libraries such as the OpenCV because people work in those projects as volunteers and as a result, the time they spent on them is limited. The I-See-Bytes library is supported by both the private sector and the universities and hopefully that will help this library to include newest algorithms faster than the opensource alternatives.

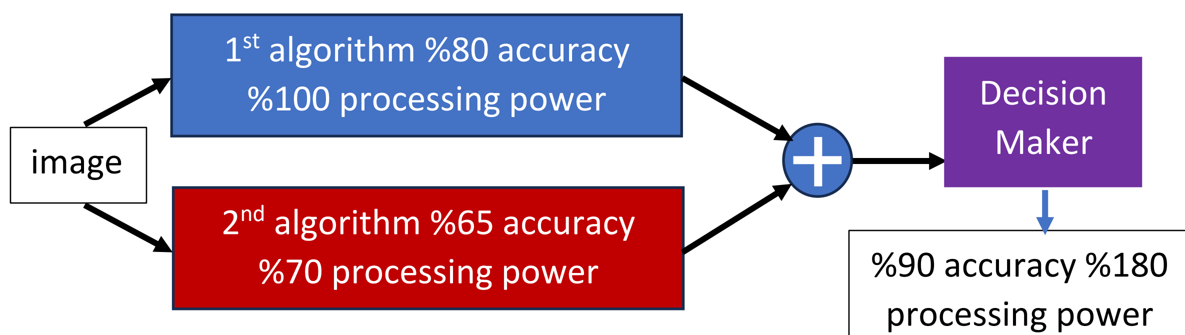


Figure 1.5. Two very different vision approaches can be fused together to result in a more successful vision algorithm.

2. Digital Signal Processing

Any digital image is considered a two-dimensional digital signal. Therefore, students need to understand the concepts and the techniques of Digital Signal Processing (DSP) in order to become good engineers. All signals are analog in nature. For example, when you speak to a microphone, the changing pressure caused by the vibration of air creates electric signals in the range of millivolts. If we plot these signal's magnitude with respect to time, we get graphs as shown in Figure 2.1. This signal is continuous in both value and time; meaning that between 345th and 346th milliseconds, there are infinite number of values.

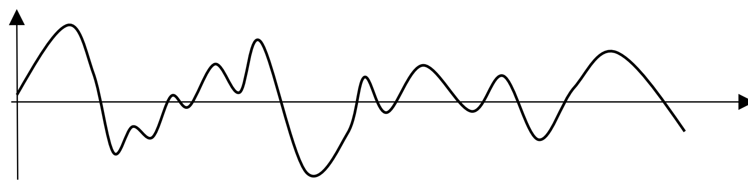


Figure 2.1. A continuous sound signal coming from a microphone.

What if we want to process this signal inside a computer? We know that everything inside a computer consists of numbers. Even the letter "A" is actually number 65 that is being interpreted as a character. This means that we need to convert the signal to numbers before processing it. The way we can do that is by measuring the voltage of the above signal with a voltmeter. A voltmeter tells us what the voltage of an energy source is. For example, if we connect a voltmeter to an everyday AA type battery, we will measure around 1500 millivolts.

The problem with a microphone signal is that it changes very quickly. Humans can hear sound frequencies between 20 Hz and 20 KHz. A 20 KHz signal alters its value **twenty thousand times a second**. Therefore, what we need is a voltmeter that can read values at twenty thousand times per second and record them. Fortunately, your computer is equipped with such a voltmeter. It is called

the sound card. When you talk into a microphone connected to your computer's sound card, the sound card measures those millivolt range signals and sends them to your computer's RAM. From there, you can either play those sounds or save them on your hard drive. You can change the base and treble of the sound. The base is low frequency signals such as drums, and treble is high frequency signals such as the sound of violin. So how does the music player on your computer increases base or treble? That is done by applying DSP techniques.

2.1. Removing Signals From Each Other

When we work with sounds, we actually work with sine signals. That is because any sound can be constructed as a sum of different sine signals. In this experiment we will write an application that generates two sine signals with different frequencies, then add those signals, and then try to separate them again. The first signal we generate is a sine signal with a period of 12 as shown below.

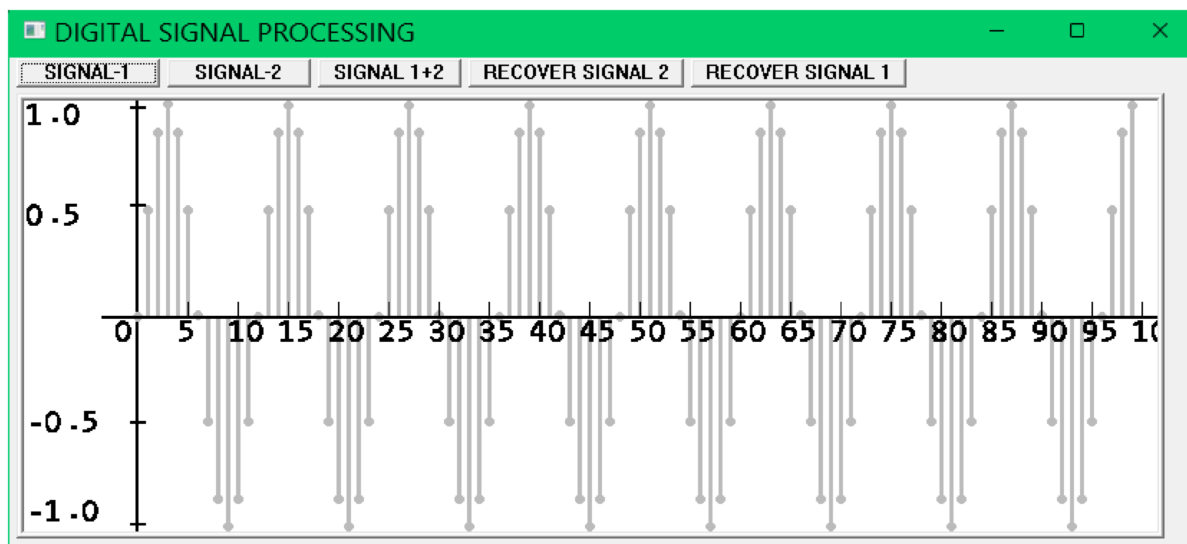


Figure 2.2. Screenshot of the program displaying a sine signal with period 12 after pressing the button "SIGNAL-1."

The code that generates this signal is shown below:

```
#include"icb_gui.h"
#include<math.h>
int FRM;
void ICGUI_Create()
{
      ICG_MWSize(850, 400);
      ICG_MWTitle("DIGITAL SIGNAL PROCESSING");
}
```

```
void Signal1()
{
	static ICBYTES signal1,graph;
	CreateMatrix(signal1, 100, 1, ICB_DOUBLE);
	for (int x = 0; x < 101; x++)
		signal1.D(x + 1) = sin(3.1415 * (double)x / (6.0));
	BarChart(graph, signal1, 300, 2, 0xffffff, 0xbbbbbb);
	DisplayImage(FRM, graph);
}
```

The above function, “Signal1(),” generates the sine function shown in Figure 2.2 and displays it. The second function, Signal2(), is called by the second button with the same name and draws another sine signal:

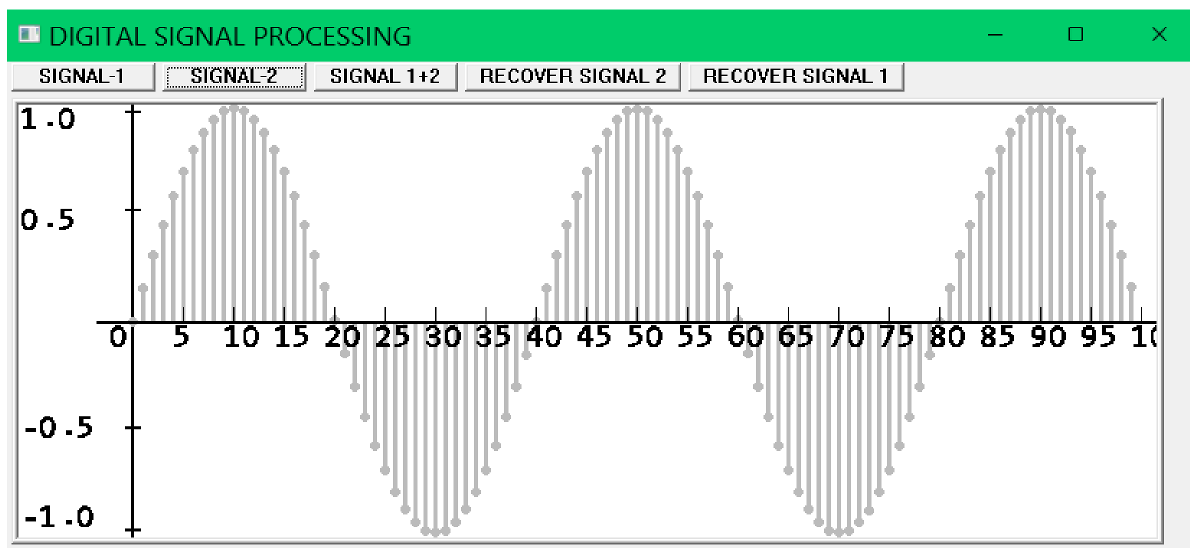


Figure 2.3. Screenshot of the program displaying a sine signal with period 40 after pressing the button “SIGNAL-2.”

The code that generates and displays the above graph is:

```
void Signal2()
{
	static ICBYTES signal2, graph;
	CreateMatrix(signal2, 100, 1, ICB_DOUBLE);
	for (int x = 0; x < 101; x++)
		signal2.D(x + 1) = sin(3.1415 * (double)x / (20.0));
	BarChart(graph, signal2, 300, 2, 0xffffff, 0xbbbbbb);
	DisplayImage(FRM, graph);
}
```

If we multiply the second one with 1.1 (increase its magnitude %10) and add these two signals, the resultant signal is shown in Figure 2.4.

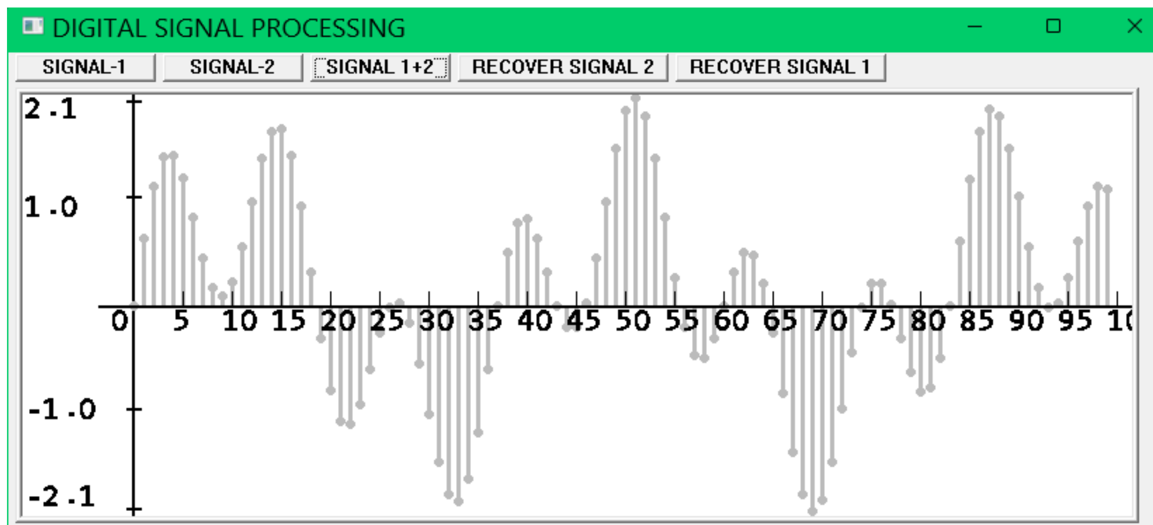


Figure 2.4. The screenshot of the program displaying a sine signal with period 12 plus period 40 sine waves; after pressing the button "SIGNAL 1+2."

The function that is called when the button labelled "SIGNAL 1+2" is shown below:

```
void Signal1_2()
{
 static ICBYTES data;
 static ICBYTES graph;
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 for (int x = 0; x < 101; x++)
   data.D(x+1)=sin(3.1415*(double)x/(6.0))+1.1*sin(3.1415*(double)x/(20.0));
 BarChart(graph, data, 300,2,0xffffff,0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

As you can see from the previous graphs, Fig 2.4 resembles neither the sine signal with period 12 nor the one with 40 samples. So how do we separate these two components if we were given only the combined signal?

EXERCISE: Before reading further, students should try to come up with a way (program) to separate these two signals.

HINT: When summed over an exact entire period (or integer multiples of it), the result is always zero. Otherwise, it is never zero.

2.2. FILTERING

Filtering is essentially shifting one short signal over a longer one and multiplying and adding sample values. The short signal is called "The Filter" and it can be something like this:

filter={ 0.1, 0.2, 0.3,0.2,0.1}

signal={0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

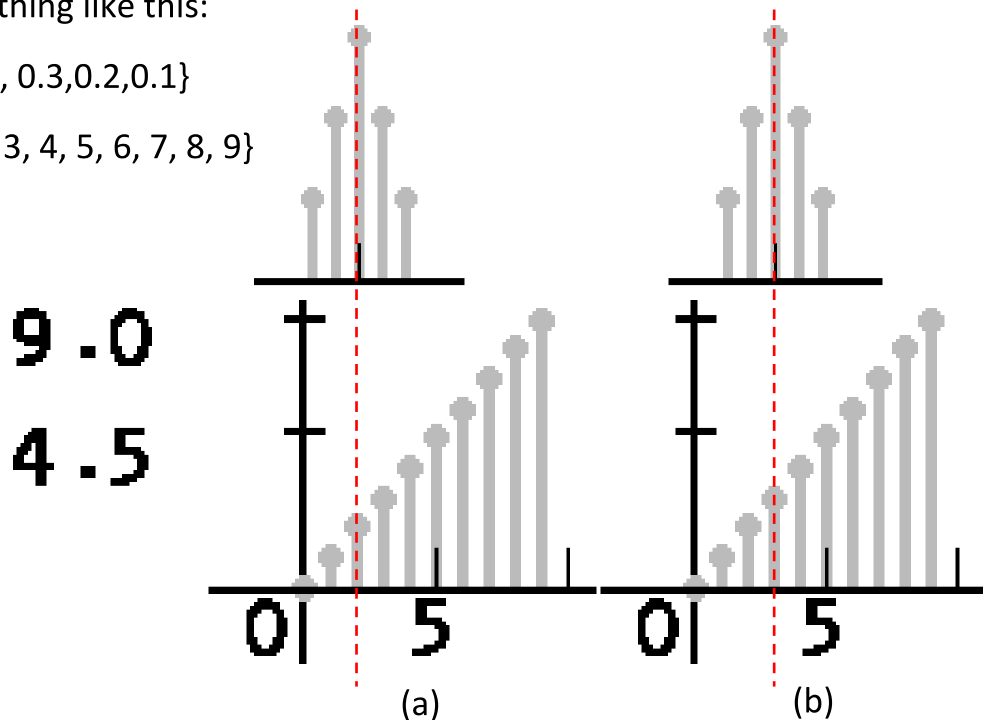


Figure 2.5. The process of filtering demonstrated with two simple signals. The filter is shifted towards the right and coinciding samples are multiplied and summed to calculate each output sample.

Let's assume that we wish to calculate the output of the filter at t=2. (Figure 2.5(a)) At this instance, the signal value is 2. We bring the middle of the filter, which is the third element (0.3) on top of t=2 and we start multiplying and adding the elements of each signal that coincide upon each other:

$$F(2)=0\times0.1 + 1\times0.2 + 2\times0.3 + 3\times0.2 + 4\times0.1$$

$$F(2)=1.8$$

Then, to calculate the output at t=3, we shift the filter signal one step to the right making the t=2 (0.3) sample of the filter with t=3 (3) sample of the signal. This time the output at t=3 is:

$$F(3)=1\times0.1 + 2\times0.2 + 3\times0.3 + 4\times0.2 + 5\times0.1$$

$$F(3)=2.7$$

2.3. Filtering the Second Signal

Since we know that the sum of a sine signal over its entire period is zero, we can use this knowledge to design a filter. The first signal has a period of 12 meaning that, if we average every 12 samples in the combined signal, the first signal's values will cancel each other out and all that remains will be the second signal. Below is the function that first generates the combined signal and then extracts the second one out of it:

```
void Recover2()
{
 static ICBYTES data, out;
 static ICBYTES graph;
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 CreateMatrix(out, 100, 1, ICB_DOUBLE);
 out = 0;
 for (int x = 0; x < 101; x++)
  data.D(x+1)=sin(3.1415*(double)x/(6.0))+1.1*sin(3.1415 * (double)x/(20.0));
  for (int x = 6; x < 95; x++)
   for (int z = -5; z <= 6; z++)
      out.D(x) += data.D(x - z) / 12.0;
 BarChart(graph, out, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

The output of this function is stored in the "out" vector and its graph is shown below in Figure 2.6. Since the middle of the moving average filter is at the 6th sample, and the entire signals must be on top of each other, the first signal that can be calculated is the 6th signal of the output which is at t=5. The slight distortion at the beginning is caused by the fact that 12 is not an odd number. If the filter's length is not odd, then such distortions might occur at either ends of the output signal. Also, you may notice that the location of the peak of the signals may shift as well.

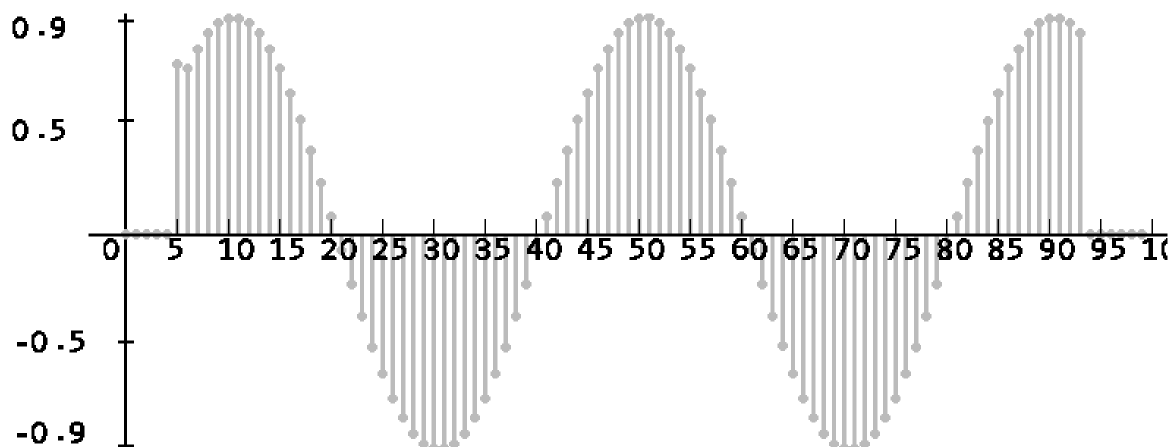


Figure 2.6. The recovered signal from the combination of two sine signals that was shown in Figure 2.4.

So how do we recover the first signal? We can do the same thing we did for signal-1 and create a longer moving average filter to suppress signal-2. Unfortunately, that method might not work well because the period of the second signal, 40, is slightly close to the quadruple the period of the first signal, 12. Another method we can apply is to boost signal-1. For this, we use a filter as shown below:

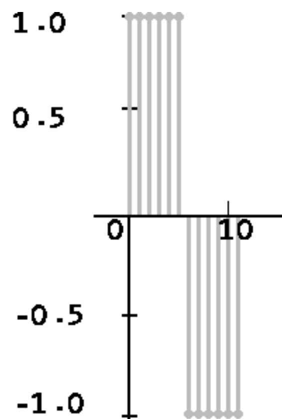


Figure 2.7. The high-pass filter (shifted) that boosts signal-1 and suppresses signal-2.

The code for this type of filtering is:

```
void Recover1()
{
 static ICBYTES data, out;
 static ICBYTES graph;
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 CreateMatrix(out, 100, 1, ICB_DOUBLE);
 out = 0;
 for (int x = 0; x < 101; x++)data.D(x + 1) = sin(3.1415 * (double)x /
(6.0)) + 1.1*sin(3.1415 * (double)x / (20.0));
 for (int x = 6; x < 95; x++)
 {
      for (int z = -5; z < 1; z++)
            out.D(x) += data.D(x - z);
      for (int z = 1; z < 7; z++)
            out.D(x) -= data.D(x + z);
 }
 BarChart(graph, out, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

When we apply this filter to the combined signal, the output of the process is shown below in Figure 2.8.

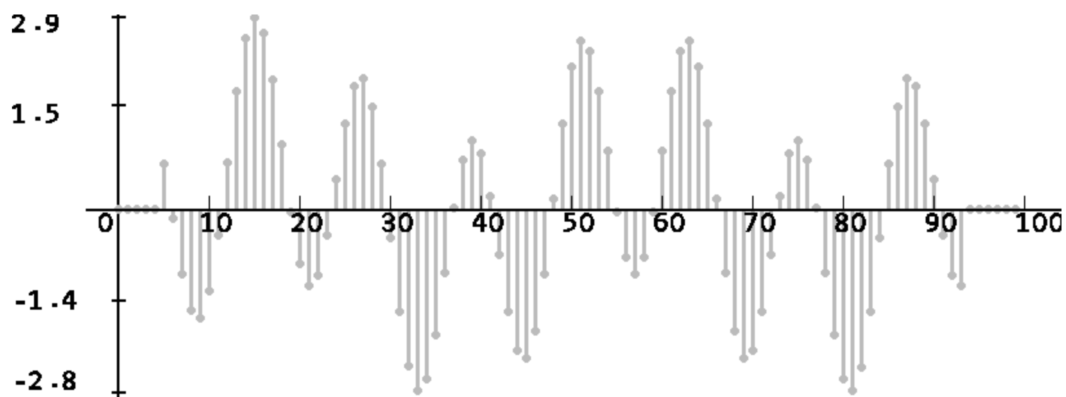


Figure 2.8. Output of the high-pass filter on the combined signal.

The rest of the code is shown below:

```
void ICGUI_main()
{
	ICG_Button(5, 1, 100, 20, "SIGNAL-1", Signal1);
	ICG_Button(110, 1, 100, 20, "SIGNAL-2", Signal2);
	ICG_Button(215, 1, 100, 20, "SIGNAL 1+2", Signal1_2);
	ICG_Button(320, 1, 150, 20, "RECOVER SIGNAL 2", Recover2);
	ICG_Button(475, 1, 150, 20, "RECOVER SIGNAL 1", Recover1);
	FRM = ICG_FrameMedium(5, 25, 160, 110);
}
```

2.4. What Does That All Mean?

In the previous sections we learned about one dimensional digital signals. We created two sine signals with different frequencies summed them up together and then try to separate that sum into its components by applying filters. What does that have anything to do with images? Images are digital signals too. The main difference is that it is a two dimensional signal. One other less important difference is that image signal samples, which are called pixels, are usually positive integers. Image pixel values usually never go below zero. In C/C++, “unsigned char” or “byte” variable type is used for holding image signals. An unsigned char is an 8 bit unsigned integer that can hold values between 0 and 255. Therefore, if we modify our code to create a sine wave that encompasses this entire range, then we get the code below.

```
void Signal1()
{
 static ICBYTES signal1,graph;
 CreateMatrix(signal1, 100, 1, ICB_UCHAR);
 for (int x = 0; x < 101; x++)
  signal1.B(x + 1) = 128.0*(sin(3.1415 * (double)x / (6.0))+1.0);
 BarChart(graph, signal1, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

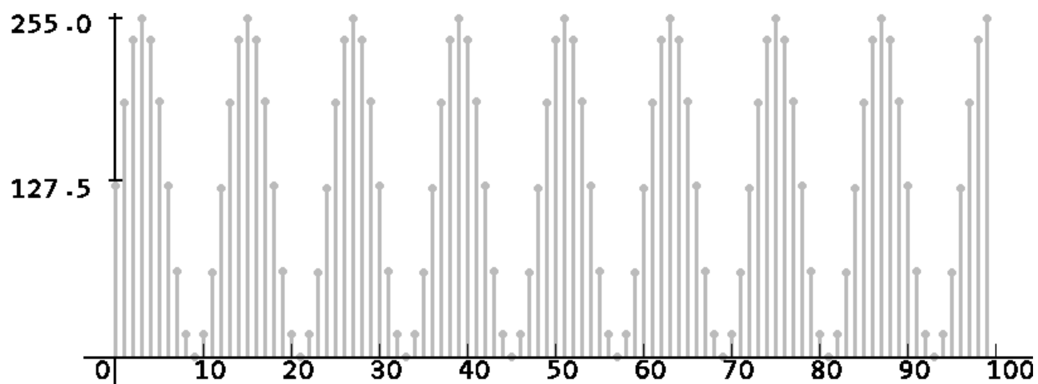


Figure 2.9. The graph drawn by the code in the previous page.

Now, what will happen if we use this signal as pixels? All we need to do is copy this signal as rows of a matrix and display that matrix as an image. This signal consists of 100 samples. If we replicate this signal 100 times for each row of a matrix then we will get a 100×100 element matrix which is a grayscale image of same size. Let's modify our code:

```
void Signal1()
{
 static ICBYTES signal1,graph;
 CreateMatrix(signal1, 100, 100, ICB_UCHAR);
 for (int y = 1; y < 101; y++)
 for (int x = 0; x < 101; x++)
  signal1.B(x + 1,y) = 128.0*(sin(3.1415 * (double)x / (6.0))+1.0);
 //BarChart(graph, signal1, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, signal1);
}
```

For a sine wave with a period of 12 pixels, we would get the image shown in Figure 2.10.(a) and for 40 pixel period (signal-2), we would get the one on the right, Figure 2.10.(b).

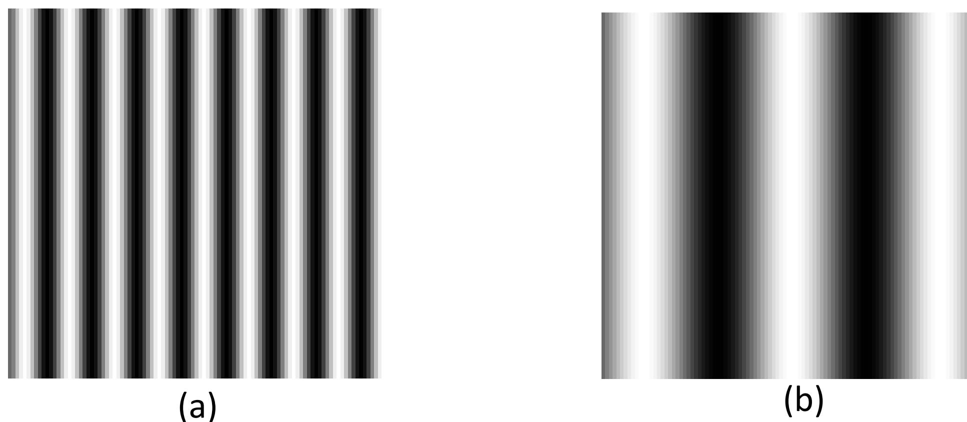


Figure 2.10. An image whose horizontal pixels are defined by a sine wave with period 12 (a). Image produced same way with a sine wave of period 40 (b).

When we look at the images shown in Figure 2.10, we see that the sine wave with the shorter period (12) creates more rapidly changing gray tones (a). A sine wave with a smaller period means a signal with higher frequency. A higher frequency in an image signal means more rapidly changing contrast levels.

Now let's remember what the moving average filtered did to our combined signal1+signal2. It suppressed the high frequency signal1 and it had less effect on the signal2. Why? Because when we average the neighboring signal values, the negative ones cancel the positives and the signal cancels itself out. A high frequency sine wave has positive and negative values closer to each other but a low frequency signal has them so far apart that the positive ones are too far away to be canceled out by the negative ones and vice versa. On the other hand, a high pass filter subtracts the negative ones from the positive ones creating even higher values (1 - (-1) = 2), boosting the high frequency signals. Because the low frequency signals have similar values around a neighborhood, this time they start canceling each other and the low frequency signal gets suppressed.

2.4.1 Notation

The moving average filter we used in the previous section can be simply expressed as a function as shown below:

$$y[n] = \frac{1}{12}\sum_{i=-5}^{6} x[n+i] \qquad (2.1)$$

Since we are dealing with discrete signals, we used the ∑ sign to express that samples are going be summed. If this were a continuous analog signal, we would have used the following notation:

$$y(t) = \frac{1}{12}\int_{k=-5}^{6} x(t+k)dx \qquad (2.2)$$

This tells us that a moving average filter is an integrator. As a matter of fact, all (pure) low-pass filters are integrators and high-pass filters are derivative operators or difference operators for the discrete case. Based on this knowledge, we can design the simplest low pass and high pass filters:

y[n]= (x[n]+x[n+1])/2 is the simplest low-pass and,

y[n]= x[n]-x[n+1] is the simplest high-pass filter.

Note that we prefer to keep the sum of the coefficients 1 for of a low-pass filter and 0 for an high-pass.

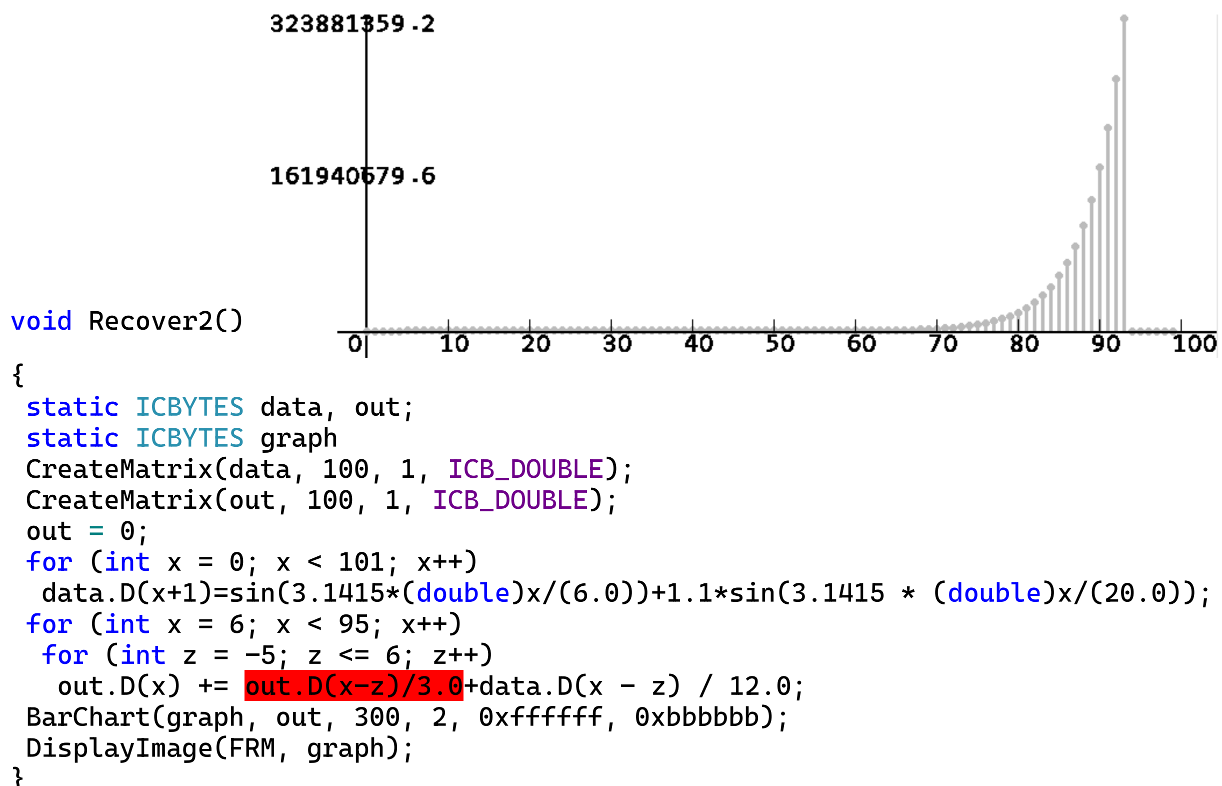
The general form of a Finite Impulse Response (FIR) Filter is:

$$y[n] = \sum_{i=0}^{m} a_n x[n-i]$$

This is a finite response filter meaning that if the input x[n] vanishes, or in other words becomes zero, then, eventually y[n] becomes zero as well. Opposite to this is an Infinite Impulse Response (IIR) filter:

$$y[n] = \sum_{j=1}^{p} b_n y[n-j] + \sum_{i=0}^{m} a_n x[n-i]$$

As you can see, this filter uses the previous outputs of the filter. That is why it is also called a recursive filter. In practice, just like an FIR filter, when the input vanishes, we would want the output of an IIR filter get so small that it can be practically considered zero. IIR filters are more difficult to design and they can easily become unstable, meaning that their output can fluctuate forever or approach infinity**. If you are not an expert, avoid designing them.** For example, do the following tiny change to the Recover2() function (highlighted with red color) and get the output shown below. Note that the if the input signal were in volts, the output would have reached almost 324 billion volts!




```
void Recover2()
{
 static ICBYTES data, out;
 static ICBYTES graph
 CreateMatrix(data, 100, 1, ICB_DOUBLE);
 CreateMatrix(out, 100, 1, ICB_DOUBLE);
 out = 0;
 for (int x = 0; x < 101; x++)
  data.D(x+1)=sin(3.1415*(double)x/(6.0))+1.1*sin(3.1415 * (double)x/(20.0));
 for (int x = 6; x < 95; x++)
  for (int z = -5; z <= 6; z++)
   out.D(x) += out.D(x-z)/3.0+data.D(x - z) / 12.0;
 BarChart(graph, out, 300, 2, 0xffffff, 0xbbbbbb);
 DisplayImage(FRM, graph);
}
```

2.5. Filters: What are They Good for?

In Figure 2.12, we see the famous mandrill image tat has been used countless times for image processing demonstrations. We first convert the RGB (**R**ed, **G**reen, **B**lue) color image to luminance image. This image has 8-bit unsigned bytes to indicate the brightness of each pixel. Then we extract the first 200 pixel values from the 100th line (row) of this image and put them in a vector called "signal". If we display that vector as a signal we get the waveform shown at the bottom of Figure 2.12.

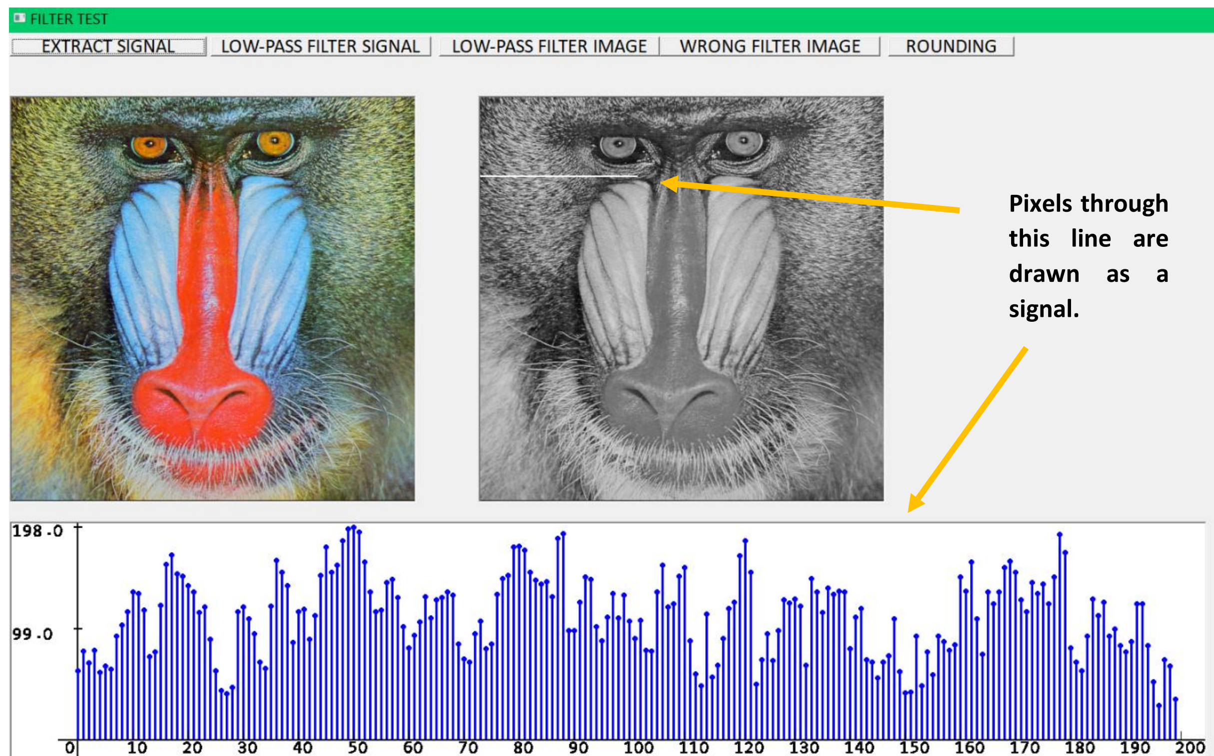


Figure 2.12. The screenshot of the program used for demonstrating low-pass filtering effects.

Below is the first part of the code for this program:

```
#include"icb_gui.h"

int MLE, SLE, FRM1, FRM2,FRM3;
ICBYTES signal;
void Extract();
void LowPassSignal();
void LowPassImage();
void WrongFilter();
void Rounding();
```

```
void ICGUI_Create()
{
      ICG_MWTitle("FILTER TEST");
      ICG_MWSize(1800, 1000);
}

void ICGUI_main()
{
  ICG_SetFont(30, 0, "Calibri");
  ICG_Button(5, 5, 250, 25, "EXTRACT SIGNAL", Extract);
  ICG_Button(260, 5, 280, 25, "LOW-PASS FILTER SIGNAL", LowPassSignal);
  ICG_Button(550, 5, 280, 25, "LOW-PASS FILTER IMAGE", LowPassImage);
  ICG_Button(830, 5, 280, 25, "WRONG FILTER IMAGE", WrongFilter);
  ICG_Button(1120, 5, 160, 25, "ROUNDING", Rounding);
  FRM1 = ICG_FrameSunken(5, 80, 512, 512);
  FRM2 = ICG_FrameSunken(600, 80, 300, 300);
  FRM3 = ICG_FrameSunken(5, 620, 1700, 300);
}
```

Now let's see the "Extract()" function that extracts the 200 pixel values from the 100th line of the grayscale image:

```
void Extract()
{
      ICBYTES RGB, grey, graph;
      ReadImage("mandrill.bmp", RGB);
      Luma(RGB, grey);
      DisplayImage(FRM1, RGB);
      CreateMatrix(signal, 200, ICB_UCHAR);
      for (int x = 1; x <= 200; x++)
      {
            signal.B_[1][x] = grey.B_[100][x];
            grey.B_[100][x] = 255;
            grey.B_[101][x] = 255;
      }
      DisplayImage(FRM2, grey);
      BarChart(graph, signal, 300, 2, 0xffffff, 0xff);
      DisplayImage(FRM3, graph);
}
```

This function creates three fluids: RGB, grey, and graph. The RGB matrix contains the color mandrill image. Note that the "mandrill.bmp" must be inside the Visual Studio project directory. After the image is load, it is converted to grayscale image using the Luma(.) function and copied into the "grey" matrix. Then a vector of size 200 is created inside the global "signal" fluid. The following line inside the for loop:

```
signal.B_[1][x] = grey.B_[100][x]; //using fast access
```

(or you could use: `signal.B(x)=grey.B(x,100);` for slow but secure access.)

tells us that the pixel values of the 100th line in the grayscale image is being copied into the "signal" vector. The next two lines are making the pixels of the 100th and 101th lines (in order to make the white line double thickness so that it can be seen better) pure white so that we can see where we are extracting the signal. Then the signal is drawn on the "graph" matrix using the BarChart(.) function.

This bar chart graph tells us that the maximum value of the pixels in the "signal" is 198. We also see that the pixel values rapidly change. That is because the signal is extracted from an area where the monkey has a lot of hair. The rapid color change caused by the monkey's hair causes the signal to go up and down quickly. From the previous section, we know that, if a signal's value changes rapidly, then it has high frequency sine signals (components) in it. Now let's see what happens when we press the "LOW-PASS FILTER SIGNAL" button:

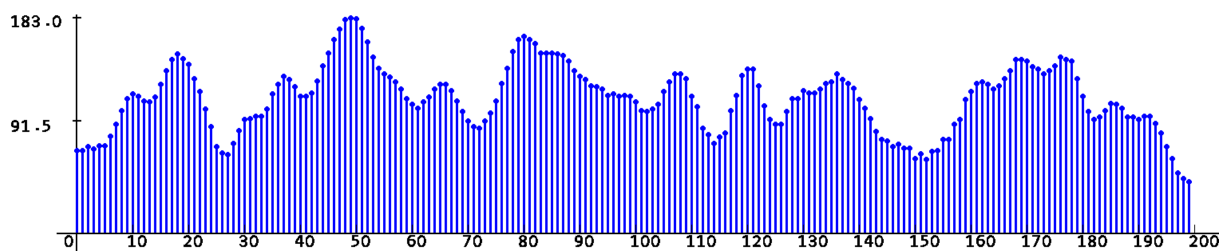


Figure 2.13. The "signal" shown in Figure 2.12 after low-pass filtering.

As we can see from this result, after the low-pass filtering is applied, the signal's values do not fluctuate as rapidly as before. The higher frequency components are suppressed. Now let's take a look at the function that accomplished this:

```
void LowPassSignal()
{
      ICBYTES filter_output, graph;
      ICBYTES filter = { 0.1,0.1,0.2,0.2,0.2,0.1,0.1 };
      Filter_H(signal, filter_output, filter);
      BarChart(graph, filter_output, 300, 2, 0xffffff, 0xff);
      DisplayImage(FRM3, graph);
}
```

In this function, we see that a vector called "filter" is defined with 7 coefficients. Then, the "signal" vector is horizontally filtered by the "Filter_H(.)" function using those coefficients.

Now let's see (Figure 2.14) what happens to the mandrill image when we press the "LOW-PASS FILTER IMAGE" button:

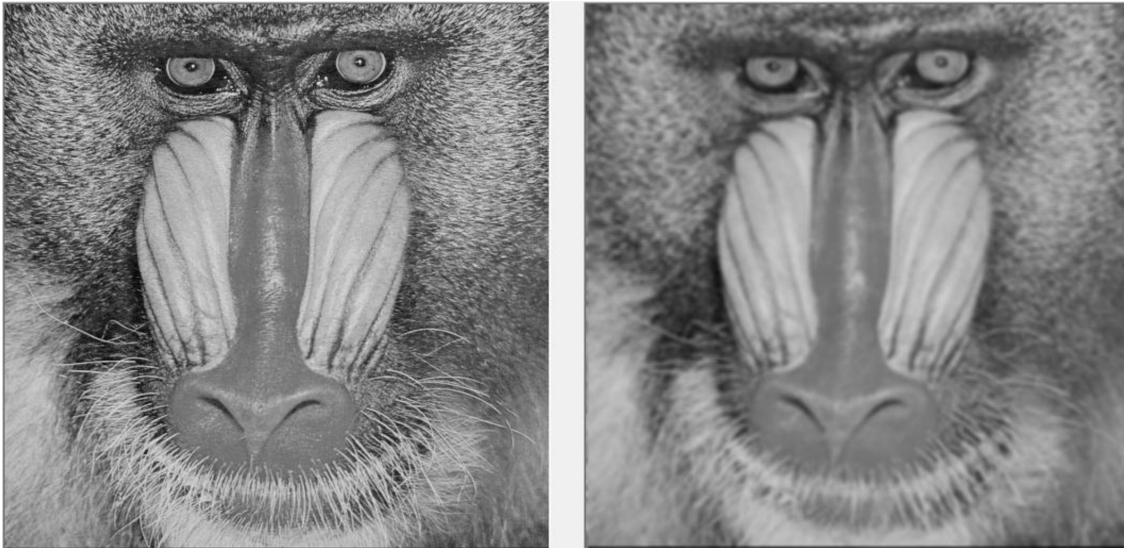


Figure 2.14. Grayscale "Mandrill" image on the left and low-pass filtered version on the right. (Original image has black pixels at the bottom.)

If you look carefully at the mustache of the monkey, you will notice that Figure 2.14 clearly demonstrates how the low-pass filtering makes an image blurry. The effect of suppressing high frequency components in an image is blurriness. You might wonder why anyone would want to make a good image blurry. We wouldn't. However, if you look at the hair of the monkey where we extracted our "signal," you will see that the rapid change of colors between black and white has been reduced as well. Imagine that those rapidly altering colors in the monkey's hair are not from the actual object (in this case the monkey) but are caused by noise. Then, we can say that:

LOW-PASS FILTERING REDUCES NOISE.

The proof of this is that larger objects with less color fluctuations, such as the nose and the eyes are less effected by the filtering. On the other hand, the hair and the mustache got intensely blurry.

So, what is the cause of noise in an image? The main cause is the camera sensor. In a camera, each pixel value is generated by a tiny light-to-voltage converter sensor. In an HD camera there are 1920×1080 (×3 for RGB) tiny sensors that convert to electricity. An approximate function of this conversion is given by this equation: $V_{out} = Light \times sensitivity + bias$.

Unfortunately, the fabrication process of a camera sensor is not perfect. As a result, each pixel has different sensitivity and bias values. This means that, even when you take a picture of a perfectly uniform color object under perfectly uniform sun light, each pixel value coming from the camera will be different from each other. That is why industrial machine vision cameras have calibration procedures that can correct this problem through software.

On top of these imperfections, thermal noise, electromagnetic signals coming from objects such as your mobile phone, and discretization effects can add on to that noise. The discretization is a well-known problem for programmers as well. Converting an analog signal to digital is like storing a float value in int. If you type:

```
int x = 155.23897712;
```

then x will contain 155. If your code were like this:

```
int x = 155.93897712;
```

x would still contain 155. The correct way of handling float to integer conversion is:

```
int x;
float f=155.500001;
x = (f + 0.5);
```

In this example, f is correctly rounded to 156 if it is equal or higher than 155.5. Otherwise, x will contain 155. Either way, if this float number were a voltage value coming from camera sensor, it would have been altered from its original value while digitization. These all are reasons why digital images, especially when recorded under dark conditions can contain a lot of noise and this noise may cause miscalculations in your algorithms. The code that low-pass filters the image is shown below:

```
void LowPassImage()
{
	ICBYTES RGB, grey, outp;
	ICBYTES filter = { 0.1,0.1,0.2,0.2,0.2,0.1,0.1 };
	ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
	DisplayImage(FRM1, grey);
	Filter_H(grey, outp, filter, ICB_UCHAR);
	Filter_V(outp, grey, filter, ICB_UCHAR, true);
	DisplayImage(FRM2, grey);
}
```

As you can see from the code, the image is first horizontally, then vertically filtered using the same filter coefficients:

```
Filter_H(grey, outp, filter, ICB_UCHAR);
```

means:

Horizontally filter the "grey" image with the coefficients in "filter" and put the result in the "output" matrix. I want the results in the "output" matrix to be of type **unsigned char**. On the other hand,

```
Filter_V(outp, grey, filter, ICB_UCHAR, true);
```

means:

Vertically filter the image inside the "outp" matrix with the coefficients in "filter" and put the result in the "grey" **image** matrix (That is what the "**true**" at the end is for. True for image matrix and false or no entry for ordinary matrix.) I want the results in the "grey" matrix to be of type **unsigned char**.

Filtering the image in both directions using the same coefficients means actually two-dimensional (2-D) filtering with this matrix:

$$M = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.2 \\ 0.2 \\ 0.2 \\ 0.1 \\ 0.1 \end{bmatrix} \times [0.1 \quad 0.1 \quad 0.2 \quad 0.2 \quad 0.2 \quad 0.1 \quad 0.1]$$

$$M = \begin{bmatrix} 0.01 & 0.01 & 0.02 & 0.02 & 0.02 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.02 & 0.02 & 0.02 & 0.01 & 0.01 \\ 0.02 & 0.02 & 0.04 & 0.04 & 0.04 & 0.02 & 0.02 \\ 0.02 & 0.02 & 0.04 & 0.04 & 0.04 & 0.02 & 0.02 \\ 0.02 & 0.02 & 0.04 & 0.04 & 0.04 & 0.02 & 0.02 \\ 0.01 & 0.01 & 0.02 & 0.02 & 0.04 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.02 & 0.02 & 0.02 & 0.01 & 0.01 \end{bmatrix}$$

We must avoid using 2-D filtering as much as we can because when we do horizontal or vertical (1-D) filtering we perform 7 multiplications and additions per pixel. We do it twice so it is 14 multiply-add instructions per pixel. Bu if we perform 2-D filtering using the matrix M shown above, then we have to perform 49 multiply-add instructions per pixel. Therefore we, must try to decompose our 2-D filter into two matrices, one horizontal and one vertical if we can.

2.6. Wrong Coefficients

In the previous example, the sum of the coefficients of the filters are always one. You must make sure that the coefficients of a matrix are between 0.0 and 1.0 because all of the C++ variables have a limit and some of them have very small limits. Most of the time, the pixels or digital signals such as sound and music will be stored in either unsigned char (byte) or 16-bit short integer formats. The maximum value that an unsigned char can hold is 255. Therefore, if your filter generates values larger then 255 and you try to store them in an 8-bit unsigned char, C++ will store only the 8 least significant bits of that value. For example, if you write this line:

```
unsigned char B = 300;
```

you will be actually storing 44 inside the variable “B” which is a far darker pixel value then 255. This is called “overflow” in programming terminology. Now let’s alter the coefficients of our filter slightly, and perform the filtering again:

```
void WrongFilter()
{
	ICBYTES RGB, grey, outp;
	ICBYTES filter = { 0.1,0.1,0.3,0.3,0.3,0.1,0.1 };
	ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
	DisplayImage(FRM1, grey);
	Filter_H(grey, outp, filter, ICB_UCHAR);
	Filter_V(outp, grey, filter, ICB_UCHAR, true);
	DisplayImage(FRM2, grey);
}
```

As you can see, we changed all of the 0.2 coefficients to 0.3. The sum of the coefficients is now 1.3. The output of this filter is shown in Figure 2.15.

Figure 2.15. Grayscale “Mandrill” image after being low-pass filtered with wrong coefficients. The dark areas are due to the overflow of unsigned char.

How can we correct this problem? Obvious choice would be to reduce the sum of filter coefficients to one but sometimes you may need a special type of filter to have a sum of coefficients that are more than one or less than zero. For those cases you can use the function “RoundFloat2Byte(.)” as shown below.

```
void Rounding()
{
      ICBYTES RGB, grey, outp;
      ICBYTES filter = { 0.1,0.1,0.3,0.3,0.3,0.1,0.1 };
      ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
      DisplayImage(FRM1, grey);
      Filter_H(grey, outp, filter, ICB_FLOAT);
      RoundFloat2Byte(outp, grey);
      Filter_V(grey, outp, filter, ICB_FLOAT);
      RoundFloat2Byte(outp, grey);
      DisplayImage(FRM2, grey);
}
```

In this function, we still use the wrong filter coefficients but in order to avoid the overflowing, we ask the filter functions to return the output in a **float** matrix:
`Filter_H(grey, outp, filter, ICB_FLOAT);`
Float can hold much larger values than 255 therefore, there is no danger of overflow. Then we use the RoundFloat2Byte(.) function to round the **float** values to **unsigned char** (byte). This function clamps the float values so that they are neither less than zero nor larger than 255. On top of that, while converting float values to integer, it does not simply take the integer part like the C++ compiler does, instead it rounds them by adding 0.5. In other words, if you have a 155.93 in the float matrix, the corresponding value in the unsigned char (byte) matrix is 156, not 155. This makes those filter calculations **scientifically more accurate**. The output of this function is shown in Figure 2.16.

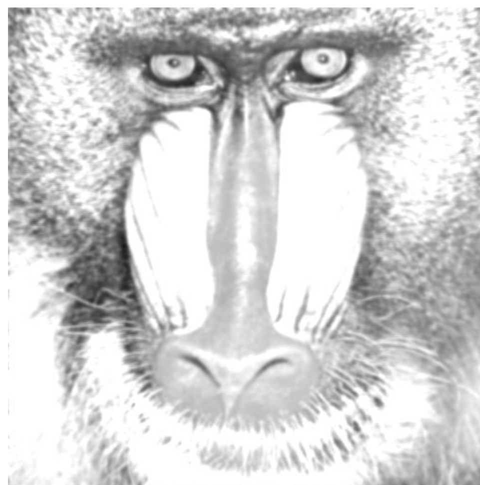


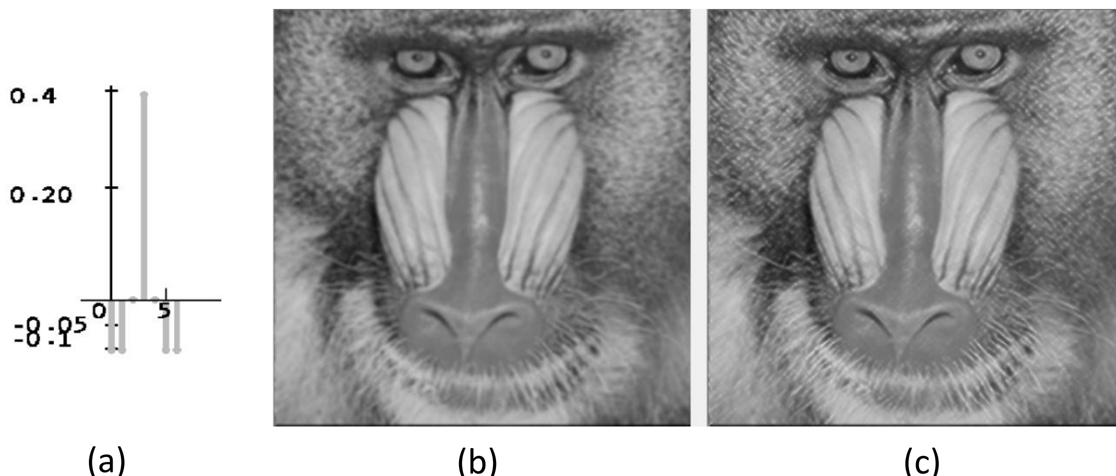
Figure 2.16. The result of filtering with wrong coefficients using byte clamping.

2.7. High-Pass Filtering

In this experiment, we shall demonstrate the effects of high-pass filtering. In the previous section we discovered that low-pass (LP) filtering makes an image blurry. Using this logic, we can infer that the high-pass (HP) filter works the opposite way and makes a blurry image sharper. In this example we will first make the mandrill image blurry and then try to recover the original image. We made the following changes to the “Rounding(.)” function:

```
void Rounding()
{
	ICBYTES RGB, grey, outp,lowpass_out;
	ICBYTES filter_low = {0.1,0.1,0.2,0.2,0.2,0.1,0.1};
	ICBYTES filter_high = {-0.1,-0.1,0.0,0.4,0.0,-0.1,-0.1};
	ReadImage("mandrill.bmp", RGB); Luma(RGB, grey);
	Filter_H(grey, outp, filter_low, ICB_UCHAR);
	Filter_V(outp, grey, filter_low, ICB_UCHAR, true);
	lowpass_out = grey;
	DisplayImage(FRM1, grey);
	Filter_H(grey, outp, filter_high, ICB_FLOAT);
	Filter_V(outp, grey, filter_high, ICB_FLOAT);
	grey *= 25.0;
	RoundFloat2Byte(grey, outp);
	outp += lowpass_out;
	DisplayImage(FRM2, outp);
}
```

In this function, we used the same “correct” filter coefficients to make the image blurry, and kept a copy of it in the “lowpass_out” matrix. Then we filtered the image with a high-pass filter whose coefficients are stored in the “filter_high” vector. Note that the sum of the coefficients of this filter is zero. It has both negative and positive coefficients as shown in Figure 2.17-a.



(a) (b) (c)

Figure 2.17. (a) HP filter coefficients. (b) Blurred image. (c) Recovered image.

High-pass filter coefficients contain both negative and positive numbers and usually their sum is zero. This means that the output of an HP filter contains much lower values then the output of an LP filter. That is why we multiply the output of the HP filter (grey*=25.0) with 25 and then add this result to the input image, which actually is the LP filtered image. Note also that the input image to the Filter_V(.) function is of type float. The filter functions can input and output any kind of matrix, however, the filter coefficient vector must always be of type **double**.

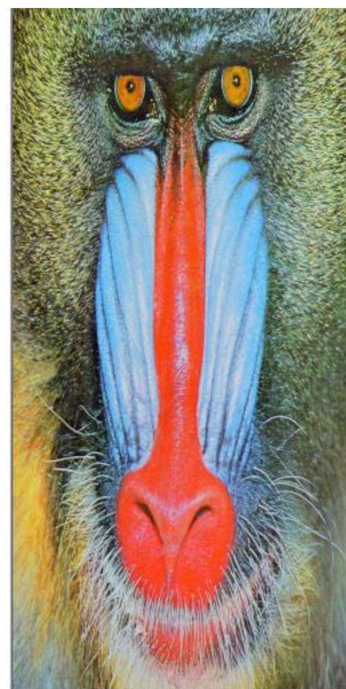
2.8. Down Sampling

A signal can be 2:1 down sampled by selecting every even or odd sample. That way the amount of data to be processed can be reduced at the expense of information loss. Sometimes, robot vision algorithms may have enough information even in a down sampled image to reach the correct results. Under such circumstances, down sampling speeds up the process by reducing the amount of data to be processed. For images however, it is better to take the average of two neighboring samples.

I-See-Bytes library allows the user to down sample signals and images 2:1 both in x and y directions using DownSampleImg_X(.) and DownSampleImg_Y(.).

The following functions' output is shown below:

```
void DownSample()
{
  ICBYTES RGB, RGBhalf,grey, greyhalf;
  ReadImage("mandrill.bmp", RGB);
  DownSampleImg_X(RGB, RGBhalf);
  Luma(RGB, grey);
  DownSampleImg_Y(grey, greyhalf);
  DisplayImage(FRM1, greyhalf);
  DisplayImage(FRM2, RGBhalf);
}
```



3. Thresholding & Segmentation

3.1. Raw Image Types

In the I-See-Bytes library there are many types of raw image types and more of them will be added in the future versions. However, the most important types are hardware compatible ones. There are two types of hardware: input and output. The input hardware are cameras, scanners etc. that generate those images. The output hardware are the ones that you can display those images. Those are mainly computer screens and printers.

Most of the time, a computer vision engineer spends his/her time in front of a computer displaying the results of algorithms on different type of images that has been saved on the hard disk. Therefore, it is important to know the types of raw image formats that a video card can display. The native image format of a video card is also the type of image that it can display fastest and that is another important reason for us to learn these formats because we wish to design the fastest software.

If you magnify a computer monitor (screen) or a TV as shown in figure 3.1, you will see that the surface of the screen consists of tiny colorful dots. Dots that are red, blue and green:

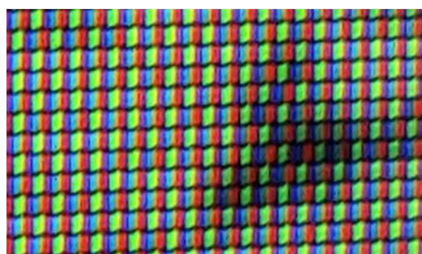


Figure 3.1. Magnified view of a computer screen.

The reason for that is because the different combinations of those three colors can produce any color that we can see (or we think that we see). The human eye has exactly the same type of receptors (neurons) that sends signals to our brain when they see those colors. This has a profound meaning: When we think that we see the color yellow, we might not be seeing a yellow light whose wavelength should be between 575 and 585 nano meters (nm) at all. We might be seeing a combination of red (620 nm) and green (550 nm). Our brain thinks that it sees a yellow object when the light coming from that object has both red and green light in it. Or it may really have yellow light coming out. It doesn't make no difference to us. Consequently, that is how we designed our electronic devices. If images are being recorded or displayed, they are always expressed as combination of these three colors: Red, Green and Blue. RGB for short.

If you have two small lights, red and green, and if you put them next to each other far away enough from yourself, you will see a single yellow light. If you repeat this experiment using combinations of red, green and blue lights, you will see the colors as shown in figure 3.2.

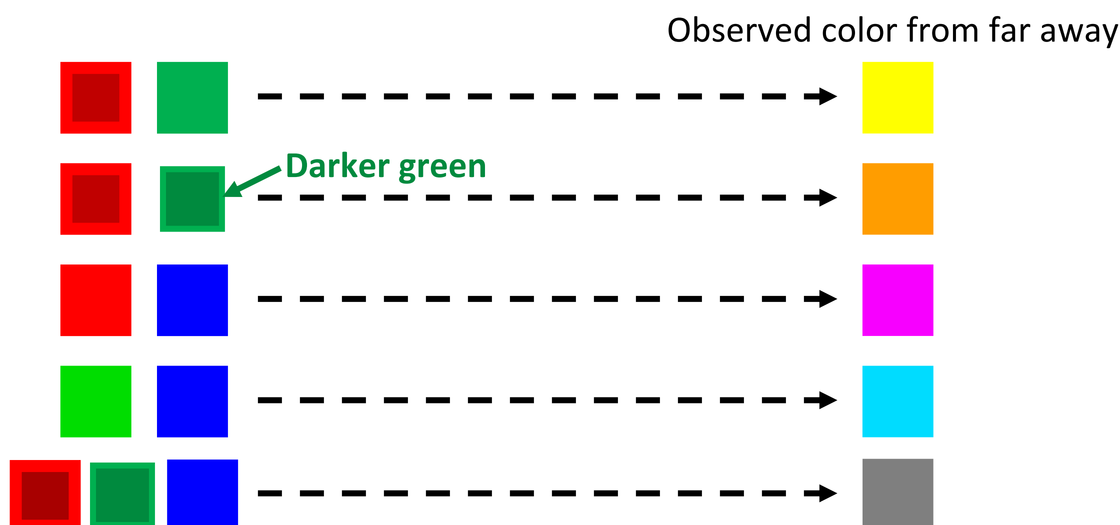


Figure 3.2. Different combinations of lights when observed from far away are seen as different colors.

If all three colors, RGB, have the same amount of brightness, then the observed light color will be shades of gray. For pixels, the amount of brightness is set by an unsigned integer which usually has a value between 0 and 255. An unsigned byte is enough to hold these values, therefore, 3 bytes are enough to create all colors that a human eye can distinguish.

Figure 3.3. Three bytes are required to set (select) the color of a pixel.

Every pixel in a color image must have 3-byte values assigned to it in order to create life like pictures. We mentioned that digital pictures are represented as matrices. So how do we represent different colors? Each color is called a channel and the matrix can have channels as in OpenCV for each color which is a bit confusing and hard to deal with. Or, as in I-See-Bytes, we can represent the color as the third dimension as show in Figure 3.4.

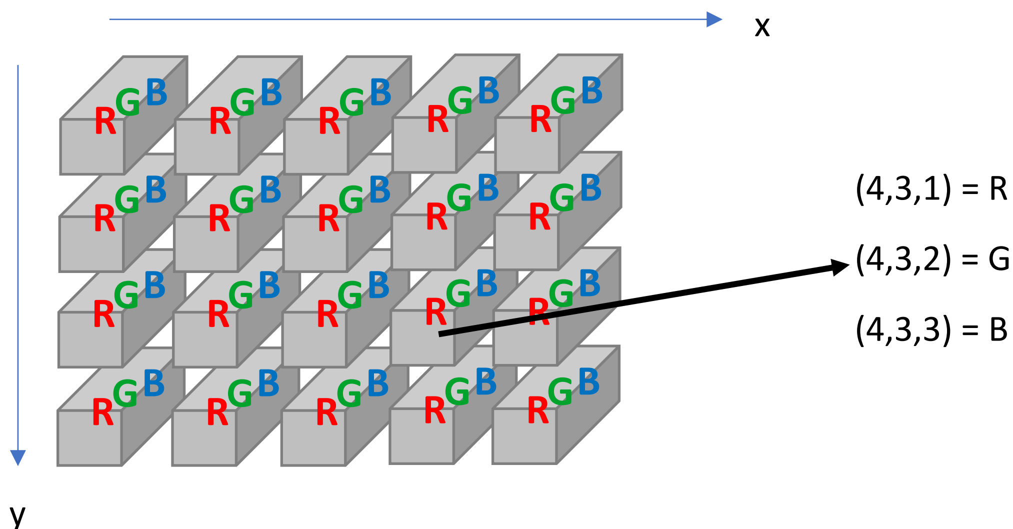


Figure 3.4. In the I-See-Bytes library, color is the third dimension of a digital image.

In order to create such an image (which is also called a 24-bit image), we can use the following commands:

```
ICBYTES RGB;
CreateImage(RGB, 200, 200, 3, ICB_UCHAR);
```

These commands will create a 200×200 image with 3 bytes (or 24 bits) allocated for color representation (depth). If we wanted to set the brightness of the green component of the pixel on the 4th column and 3rd row to 155, we can simply write: RGB.B(4,3,2)=155. (.B for "Byte" or uchar).

Another popular raw image format is 32-bit image. Addition to the three bytes of red green and blue channels, there is an alpha channel which may be used for different purposes such as transparency or extra brightness. Such formats can be abbreviated as ARGB (A for alpha). To create such an image in I-See-Bytes:

```
ICBYTES ARGB;
CreateImage(ARGB, 200, 200, 4, ICB_UCHAR);
```

you can either define the matrix as 3-dimensional with color dimension having 4 bytes (4×8 bit = 32 bit) or,

```
CreateImage(ARGB, 200, 200, ICB_UINT);
```

Since unsigned integer (UINT) is 32 bits long, this 2-dimensional matrix still provides 32-bit color depth for your pixels. For this case, you need to access the elements of the matrix using "U" (short for UINT) as shown below:

```
ARGB.U(4,3) = 0x80A5CC;
```

R=0x80=64 G=0xA5=165 B=0xCC=204

This expression sets the red component of the 4th column and 3rd row pixel to 64, the green component to 165 and the blue one to 204 in one single assignment. In order to achieve that, we use the hexadecimal number system because we can identify the values of each channel by looking at the digits of an hexadecimal number. In C/C++, any expression starting with 0x is treated as hexadecimal. If we don't want to use hexadecimal system we can do the following:

```
ARGB.U(4,3) = 8431052;
```

That expression will set the RGB components to 64, 165 and 204 also. Unfortunately, we cannot see that by simply looking at the number 8431052.

How about the 4th component, alpha? For the moment we shall not use that component. If we are not going to use the alpha component, why do we use 32-bit images? Because we can set the RGB components with a single instruction instead of three and that is %300 faster. Note that the sequence of RGB can be different for different hardware, it can be BGR or ABGR or some other combination. Do not assume that every video card, camera or image format use the same sequence of colors in memory.

3.1.1. Grey Level Images

What if we want to create black and white images that has only shades of grey? One way to do that is obviously to use RGB-24 or ARGB-32 images and set all colors of a pixel to the same level. For example, if you set R=128, G=128 and B=128, then you will obtain the grey shade that is in the middle of the grey spectrum. But that would be wasting memory. If every component of each pixel has the same value then there has to be a way to create a matrix that holds only one value instead of keeping three copies of the same value. That method is:

```
ICBYTES GRAY;
CreateImage(GRAY, 200, 200, ICB_UCHAR);
```

Simply, a 2-dimensional matrix with a depth of 8-bits will give you a gray level image. You can set the gray level of a pixel by writing:

```
GRAY.B(4,3) = 128;
```

"B" is for byte, which is the same as unsigned char in C/C++. To convert a color image to grey level, use the Luma(.) function:

```
ICBYTES RGB,GRAY;
CreateImage(RGB, 200, 200, 3, ICB_UCHAR);
//..
//do something with the RGB then convert it to grey level:
Luma(RGB,GRAY);
```

You can use both 24-bit and 32-bit images as an input to the Luma(.) function. If you send an already gray level image by accident you will receive a copy of it.

3.2. Color Thresholding

Thresholding is the process of labeling pixels belonging to target objects or regions. For example, you might need to label pixels that belong to a product or the background. In a factory where machine parts are manufactured, you should inspect these parts before selling to your customers. If you sell defective machine parts to your customers, they will complain, return the parts back to you, or abandon purchasing your products all together. That is why every manufacturer perform quality control on their products before selling them to customers. But those customers that buy those machine parts are usually companies themselves and if they don't trust the seller's quality, they can set up a quality inspection department themselves to ensure the quality of the machine parts they purchase. Otherwise, the products they manufacture using those machine parts they bought would be of low quality.

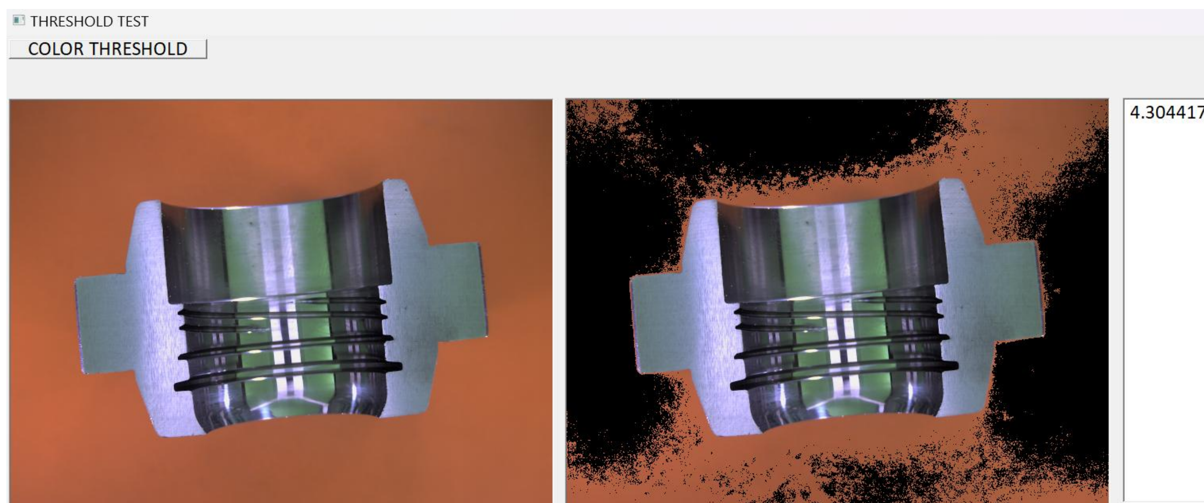


Figure 3.5. A machine part photographed on top of an orange background and the first attempt to threshold the pixels belonging to it.

Figure 3.5 shows a machine part that we wish to inspect. The metal object is located on top of an orange background and photographed. The actual digital image has much higher resolution but for our purpose this resolution is adequate. If we know that the object is located around the middle of the image frame, then we can use this trick to learn the background: We can learn the colors at small sections of the corners because we know that they always belong to the background.

If you look at this picture carefully, you will notice that the metal part has bluish-grey color. Steel is silvery shiny, but it also has tendency to be slightly blue. That is why we hear the “blue steel” phrase very often. Who ever chose this background, wanted a color opposite to blue so that the thresholding process can be performed more easily. We also see green reflections on the machine part meaning that an opaque, green light source has been used to illuminate the object. This means that we can use the color difference between the background and the object to label the pixels. In order to do that, we write a program that calculates the average yellow (red+green) and blue components of the 50×50 pixel portion of the top left corner of this image:

```
ICBYTES RGB,masked;
ReadImage("machine_part.bmp", RGB);
CreateImage(masked, RGB.X(), RGB.Y(), RGB.Z(), ICB_UINT);
float blue=0, yellow=0, dark_thresh, color_thresh;
int x, y;
unsigned char* pix;
for (y = 1; y <= 50; y++)
{
	pix = (unsigned char*)RGB.U_[y];
```

```
        for (x = 1; x <= 50; x++)
        {
                //          green               red
                yellow += pix[x * 4 + 1] + pix[x * 4 + 2];
                blue += pix[x * 4];
        }
}
```

In this code, we first load the "machine_part.bmp" as a 32-bit color image matrix called "RGB." Then we create another 32-bit blank (empty) image exactly the same size as that one. Using the "for" statements, we sum the green and red components of the pixel values and store the sum inside the variable called "yellow." In order to do that, we type cast the unsigned int values of 32 bits to unsigned char using the following code line:

```
pix = (unsigned char*)RGB.U_[y];//the first pixel of every row
```

We also sum the blue components and store them in another variable called "blue." That makes a total of 50×50=2500 pixels. You would think that such an amount would be enough to give us the average strength of yellow (or orange) and blue components of the background. Therefore, we set our threshold for pixel color as follows:

```
color_thresh = yellow / blue;
ICG_printf(MLE,"%f\n", color_thresh);
```

This gives us 4.304417. Now that we know the ratio of the orange to blue components of background pixels, we can use this knowledge to label the entire pixels of the image as background and foreground (metal object). We know that the orange component of the metal is much lower than the background but the blue component is much higher. Therefore, if we scan the entire image and calculate orange/blue ratio for each pixel, we can decide if that pixel belongs to the orange background or the metal foreground. The following code does exactly that:

```
for (y = 1; y <= RGB.Y(); y++)
{
        pix = (unsigned char*)RGB.U_[y];
        for (x = 1; x <= RGB.X(); x++)
        {
                //         green               red
                yellow = pix[x * 4 + 1] + pix[x * 4 + 2];
                blue = pix[x * 4];
                if ((yellow / blue) > color_thresh) masked.U_[y][x] = 0;
                else masked.U_[y][x] = RGB.U_[y][x];
        }
}
```

```
DisplayImage(FRM1, RGB);
DisplayImage(FRM2, masked);
```

This code calculates the yellow/blue ratio for each pixel and if the ratio is larger than the threshold calculated in the previous step than that must be a background pixel and the corresponding pixel (the pixel with the same coordinates) in the "masked" image is set to color black. On the other hand, if the threshold is not higher, then it is considered a foreground pixel and the corresponding pixel's color in the masked image is set to the color of RGB's image. This way we hope that all the orange background turns black (0) in the "masked" image and only the metal foreground remains.

Unfortunately, looking at the resultant "masked" image in Figure 3.4, we see that nearly half of the orange foreground pixels are still there. Where did we go wrong? If we look at the original image carefully, we will notice that the background color is not exactly uniform. There are slightly darker and lighter areas caused by nonuniform illumination. As a result, our threshold is too high for some regions. In that case, we can try to lower the threshold just a little to see what happens. The original threshold value is 4.304417. If we subtract 0.8 from this threshold value:

```
color_thresh -= 0.8;
```

And run the last part of the code again. We will get the image in Figure 3.6. some people might think that using a threshold is wrong and the threshold value should be adaptive or pattern classifier should be used. Those are ignorant people who do not know or understand the math behind these algorithms. First of all, when we calculated the average yellow and blue components of the 50×50 pixel portion of the top left corner, that was adaptively learning the threshold.

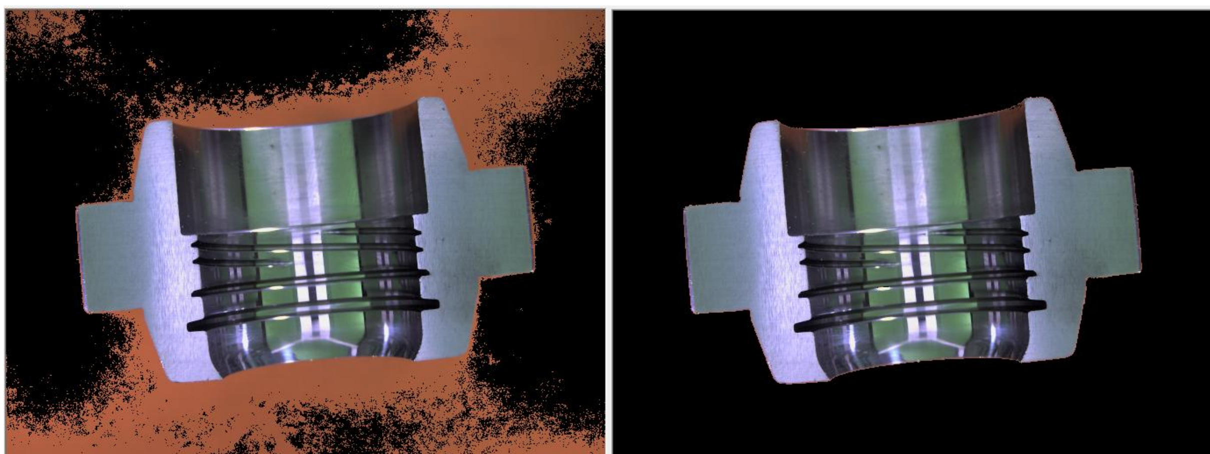


Figure 3.6. The first attempt to threshold the pixels belonging to the metal object (left), and the second one with slightly lowered threshold (on the right).

Secondly, a pattern classifier is IN FACT a threshold machine. A pattern classifier receives many features as input and it determines the optimum thresholds based on those features. If there were only one feature available, then any pattern classifier would be reduced to a single threshold. But people who do not understand the math behind the pattern classifiers are unaware of that.

The important thing is to be efficient. And the best way to be efficient is to find the one perfect feature that solves your problem. That is the philosophy behind good engineering. Simple, fast and inexpensive. Unfortunately, perfect features do not come by easy and the reader should not think that the feature we used here, yellow/blue ratio, is a perfect feature. There are many ways to improve up on it. Our goal here is to demonstrate the basic concepts of color thresholding.

3.3. Grey Level Thresholding

Figure 3.7 shows a poor-quality photograph of a book page. If we wanted to recover the letters from the background, what type of pixel threshold should we use? The problem with grey level images is that there are no colors such as yellow and blue whose ratios we can use to create a threshold. All colors are grey. In that case, we have no other option but to use the pixel values as threshold. In a grey level image pixels values may vary between 0 (black) and 255 (white). If we create a histogram of pixel values perhaps the histogram might give us an idea.

Figure 3.7 also shows the histogram of this image. A histogram is a count of occurrence of a pixel value in that image. For example, according to the histogram, neither full black (pixel value 0) nor full white (pixel value 255) occurred in this image. On the other hand there are 967 pixels with a brightness of 146.

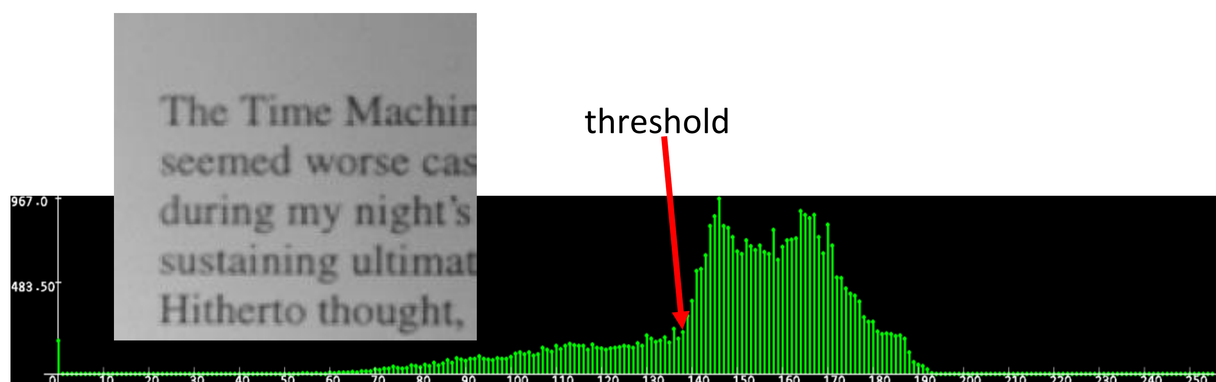


Figure 3.7. A poor-quality photograph of a book page and its histogram.

The code that generates and displays the histogram is shown below:

```
ICBYTES RGB, grey, histo, histograph;
ReadImage("textsample.jpg", RGB);
Luma(RGB, grey);
GrayHisto(grey, histo);
BarChart(histograph, histo, 300, 3);
DisplayImage(FRM1, RGB);
DisplayImage(FRM3, histograph);
```

This code loads the photograph of the book page and then converts it to a grey level image matrix using the function Luma(.). Then we call the GrayHisto(.) function to create a vector that contains the grey level histogram of the image. We are already familiar with the BarChart(.) function (Chapter 2).

Ideally, when we look at a histogram, we would want to see two hills as far away from each other as possible. These two hills are created by the foreground and background pixels. If that were the case, we could select a point equally distant from both peaks and call it the threshold value. Unfortunately, in our histogram, there is only one hill with a rugged peak. The location of the hill is close to the right (bright) side therefore, we can assume that the hill belongs to the background pixels which are relatively bright. In that case we can select the left foot of the hill as the threshold point which is around 138. That point is shown with a red arrow in Figure 3.7. In order to see what happens when we threshold this image using 138 as the threshold, we use the following program:

```
ICBYTES RGB, grey, masked;
ReadImage("textsample.jpg", RGB);
Luma(RGB, grey);
GrayThresh(grey, masked, 138);
DisplayImage(FRM2, masked);
```

The above code uses the function GrayThresh(.) to threshold the gray level image. In the previous example for color thresholding, we had to write a nested for-loop to scan all of the pixels one by one. The GrayThresh(.) function does exactly that for us but works only on grayscale images. The output of this code is shown in Figure 3.8-b.

You may have noticed the difficulty of choosing the correct threshold value. In this image, despite the fact that the foreground (letters) and the background (paper) pixels are clearly distinguishable by the human eye, we still had some difficulty choosing a threshold value. What if the image were more complicated and it looked like the foreground and the background colors were so close to each other that they looked like they merged together? Is there a way to automatically select the threshold value? Fortunately, many scientists worked

on this problem and they developed different methods. One of the most popular methods is Noboyuki Otsu's method. This method is implemented in I-See-Bytes library and should be used as follows:

```
ICBYTES RGB, grey, masked;
ReadImage("textsample.jpg", RGB);
Luma(RGB, grey);
int thresh = Otsu(grey);
ICG_printf(MLE, "%d\n", thresh);
GrayThresh(grey, masked, thresh);
DisplayImage(FRM1, grey);
DisplayImage(FRM2, masked);
```

In the above code, the grey level image matrix (8-bit color depth) is sent as an argument to the Otsu(.) function and the function returns an integer threshold value. Then we use this value as an input argument to the GrayThresh(.) function along with the greyscale image to obtain the binary image shown in Figure 3.8-c. The threshold value returned by Otsu(.) is 132 which is very close to the one we selected (138). Despite the fact that our threshold was only 6 levels higher than the one Otsu(.) returned, the improved result can be clearly seen between Figure 3.8-b and 3.8-c.

After thresholding, the images have only two pixel levels: 0 and 255. Those types of images are called binary images. When we look at those binary images, we see that the letters on the lefts side are missing pixels and the letters on the right side have extra pixels that make them as if they are bold style. The reason for that is because the illumination of the page is nonuniform. While the left side of the image has more illumination, the right side has less. Because we apply the same threshold to the entire image, we have such issues. The smart thing to do would have been to divide the image to sections and apply Otsu's threshold algorithm to those sections as if they are separate images. The result would have been much better.

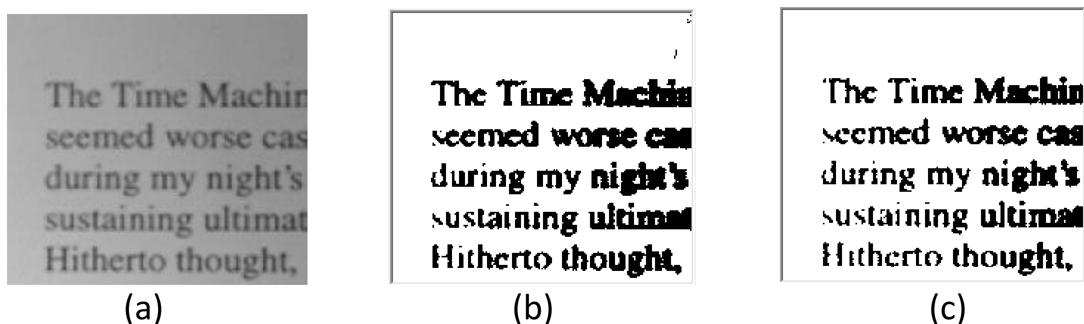
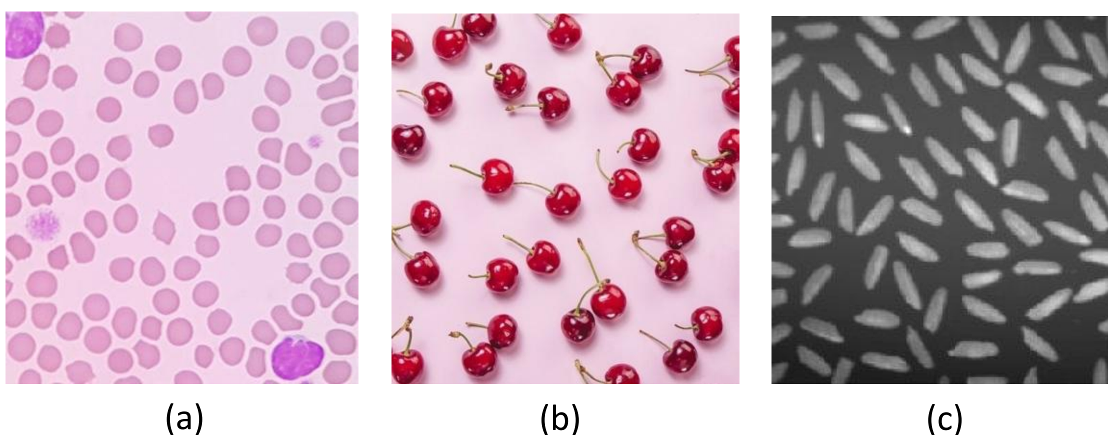


Figure 3.8. (a) A poor-quality photograph of a book page. (b) Result of thresholding with our choice (138). (c) Result of Otsu's method (132).

3.4. Pixel Segmentation

Consider the images shown in Figure 3.9. The first one shows the blood cells under a microscope. The second one shows cherries on a table and the third one is enlarged (zoomed) view of rice grains. Imagine that you need to write a program that counts these objects. One simple method may be to threshold the images, count the foreground pixels and then divide the number of foreground pixels to the average size of the objects. Although that would give us a very good estimate, unfortunately, both the "Blood cells" and the "Rice Grains" have very different sizes in terms of pixels and that approach would lead to wrong answers.



(a) (b) (c)

Figure 3.9. Three different images that we wish to count the objects in. (a) Blood Cells. (b) Cherries. (c) Rice Grains.

A better way to solve this problem is to group the pixels belonging to objects. If objects are not touching each other, then all the pixels that touch each other must belong to the same object. In that case, if we can count how many groups of connected pixels that are, then we will know exactly how many objects there are in the image. Figure 3.10 shows magnified pixels of a binary image with two separate objects on it. If we could group (label) these pixels based on which object they belong to, then we would know exactly how many groups of pixels, or in other words how many distinct objects there are in the image.

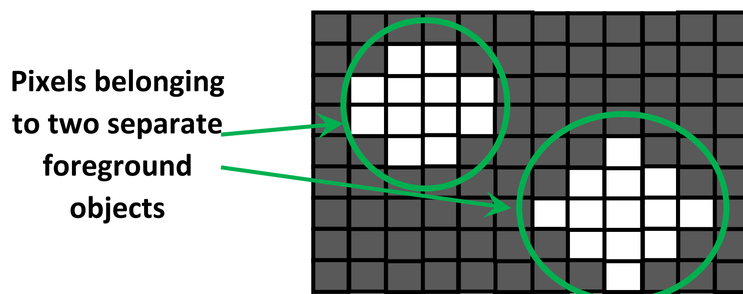


Figure 3.10. Magnified binary image with dark pixels as background and white pixels as foreground.

Segmentation process does exactly that. It either labels each pixel based on which object they belong to (Figure 3.11-a) or, it creates a list of pixel coordinates (x,y) for each object (Figure 3.11-b).

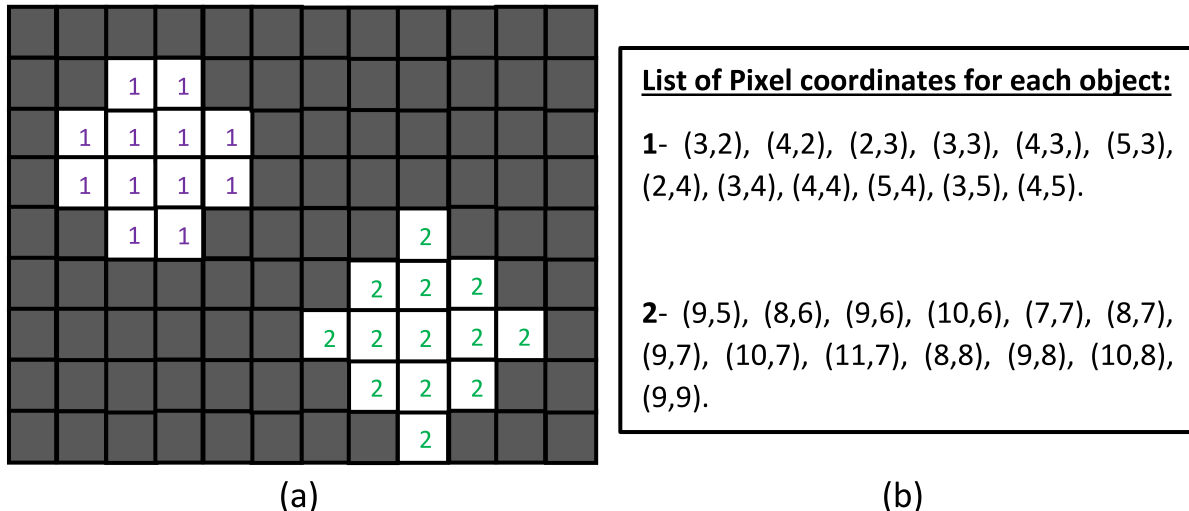


Figure 3.11. Results of binary image segmentation. (a) Pixel labeling. (b) Pixel listing.

Both of these formats are useful for different purposes. For example, pixel labeling allows you to trace the peripheral pixels more easily. On the other hand, listing coordinates allows you to find the center of gravity of the object faster. Therefore, the I-See-Bytes library prepares both of these lists. The function that segments binary images is called SegmentPixels(.). After the segmentation process, this function returns two ICBYTES objects. The first one contains the labels as an unsigned short matrix that is the same size as the image. The second one contains a vector list which contains the coordinates of the pixels of each object. There is also a function that paints these segments using different random colors and that one is called PaintSegments(.). It accepts either the labels or the pixel lists the SegmentPixels(.) prepares.

WARNING: SegmentPixels(.) function always assumes that foreground objects have white pixels and background pixels are dark. When you simply threshold an image, you need to know if your foreground objects are darker or brighter than the background. For example, in Figure 3.9, “Blood Cells” and “Cherries” have bright background. On the other hand, the “Rice Grains” picture has a dark. You need to use the hidden fourth parameter (true/false) of GrayThresh(.) function if you want its result to be color inverted. Default is false so it doesn’t invert. Make it true and it will.

Figure 3.12 shows the results of thresholding and segmentation for the images shown in Figure 3.9. The code that generates cherries is given below:

```
ICBYTES RGB, grey, masked, seg, map, coloredsegments;
ReadImage("cherries.jpg", RGB);
Luma(RGB, grey);
int thresh = Otsu(grey);
GrayThresh(grey, masked, thresh, true);
SegmentPixels1(masked, seg, map);
PaintSegments(map, coloredsegments);
DisplayImage(FRM1, RGB);
DisplayImage(FRM2, masked);
DisplayImage(FRM3, coloredsegments);
```

Extra “true” here inverts the binary image colors

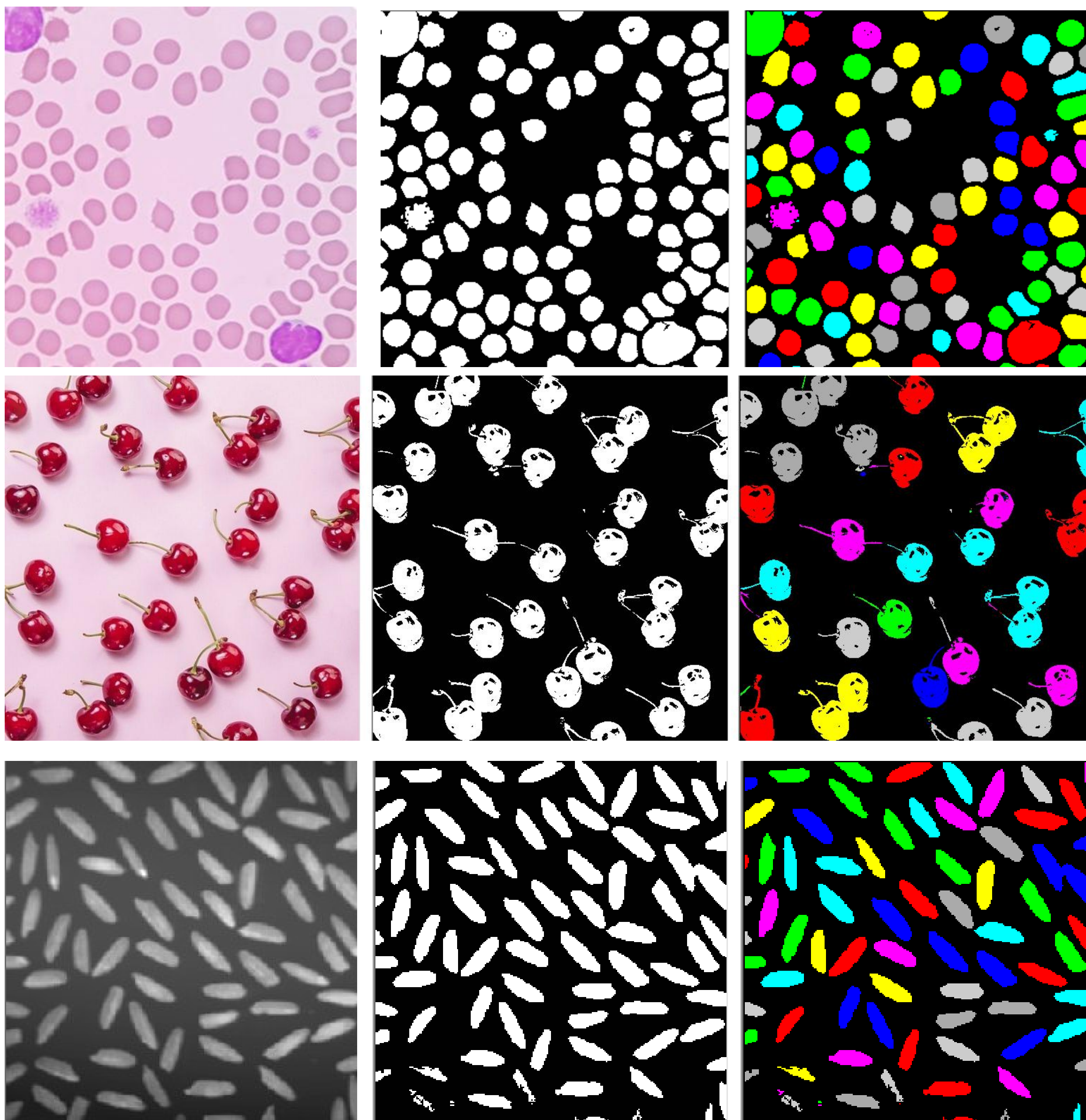


Figure 3.12. Results of grey level thresholding and segmentation. Images of Figure 3.9 (left). Binary images after thresholding with Otsu’s method (middle). Segmented and colored versions of binary images (right).