ver. 1.0

Introduction to image processing in Scilab

SCILAB 5.4.1 / IMAGE PROCESSING DESIGN TOOLBOX 8.3

SZYMON MOLIŃSKI

Table of Contents

Introduction

Scilab is a high level numerical programming language. It is used for the complex mathematical calculations which are everywhere in the engineering. Scilab may be used for rapid prototyping of new algorithms from many fields: also in remote sensing and image processing.

This tutorial is created to show you functionalities of Scilab. We will focus especially on the functions for image processing and remote sensing. This document is only the first step which you may do to become a professional. When you end this tutorial - practice. Build your own algorithms and programs with Scilab. This is the way to be an expert.

This document shortly will be out-of-date, because new version of Scilab was released (Scilab 6.0..0) which is much better than version 5.4.1 used in this tutorial. The reason why we choose older release is it compatibility with the Image Processing Design Toolbox.

When a new version of Image Processing Toolbox is available I will modify this tutorial.

But don't worry – Scilab syntax will be the same as presented here so this is not out-of-date knowledge.

Licensing and copyrights

Scilab is governed by the CeCILL license (GPL compatible) abiding by the rules of distribution of free software since Scilab 5 family. More information about the license is provided in the external link given in the references [1]. Generally CeCILL is very similar to GNU license. And Scilab 6.0.0 is licensed by GNU.

This tutorial is publicized by Creative Commons Share-Alike 4.0 [2].

Materials from this document may also been licensed. Information about this fact will be provided with them.

Installation

Installation steps provided here are for Windows operating system and Scilab version 5.4.1. For Linux or Mac please follow the instructions from the site: https://wiki.scilab.org/howto/install/linux or http://www.scilab.org/download/latest. Scilab may be not compatible with the Mac OS. Image Processing Design Toolbox may be not compatible with the Linux systems. That's why the safest installation for the beginners is on the Windows.

Installation process step by step:

1) Go to the website: **http://www.scilab.org/download/5.4.1**
2) Choose the installation files proper for your system properties (32-bit or 64-bit) and download it.
3) Run execution file.
4) Choose default installation type (Full).
5) Launch Scilab.
6) Go to the: Applications -> Module Manager ATOMS
7) Choose Image Processing folder from the list. Then choose the "Image Processing Design Toolbox" and click to 'install' button.
8) Restart Scilab. Everything is ok if you see this in the Command Window:

```
>>Start IPD - Image Processing Design

     Load macros

     Load dependencies

     Load gateways

     Load help

     Load demos
```

9) If you didn't see IPD toolbox on the list from Image Processing folder check All Modules folder. If there is still nothing you must download IPD Toolbox from the website: https://atoms.scilab.org/toolboxes/IPD
10) Then type in the Command Window: atomsInstall("*path_to_your_zip_file*").
11) Restart Scilab.

The basic operations in Scilab

In this tutorial I will show you basic **datatypes and operators** in Scilab. There are much more of them. If you are interested in check Scilab's help or demonstrations. Access to demos is provided by toolbar: ? ->Scilab demonstrations -> Introduction: Getting started with Scilab.

The code which you should provide to the *Command Window* is started with the "-->" symbol and output in the *Console Window* has symbol ">>".

The code written in the *editor* is predicted by ">" symbol and it is always written in the frame.

a) SCALAR

```
--> s = 4
--> s

>> s = 4
```

b) BOOLEAN

```
--> (1==2) == %t

>> F
```

c) STRING

```
--> s = "string"

>> s = string
```

d) MATRIX

```
--> M = [1 1 1; 1 1 1; 1 1 1]

>> M  =

   1.   1.   1.
   1.   1.   1.
   1.   1.   1.
```

e) ROW VECTOR (one row, many columns matrix)

```
--> V = sin(0:0.1:1)

>> V  =

      column 1 to 6
   0.   0.0998334   0.1986693   0.2955202   0.3894183   0.4794255
      column  7 to 11
   0.5646425   0.6442177   0.7173561   0.7833269   0.8414710
```

f) COLUMN VECTOR (one column, many rows)

```
--> V = V'

>> V  =

   0.
   0.0998334
   0.1986693
   0.2955202
   0.3894183
   0.4794255
   0.5646425
   0.6442177
   0.7173561
   0.7833269
   0.8414710
```

g) MATRIX MULTIPLICATION BY SCALAR

```
--> MAT = ones(3,3); SCAL = 0.2e-3;
--> M_S = MAT * SCAL

>> M_S  =

   0.0002    0.0002    0.0002
   0.0002    0.0002    0.0002
   0.0002    0.0002    0.0002
```

h) MATRIX MULTIPLACATION BY MATRIX (i-th and j-th ELEMENT from matrix (A) BY i-th and j-th ELEMENT from matrix (B)):

```
--> M1 = rand(2,3);
--> M2 = round(rand(2,3)*10);
--> M = M1.*M2

>> M  =

   1.969348     9.7023218    3.3933045
   6.1997254    5.1094586    7.0180254
```

i) MATRIX PRODUCT / ROW ADRESSING

```
--> M1 = ones(2,3);
--> M2 = zeros(3,4);
--> M2(1,:) = exp(1:1:4); M2(2,:) = log(2:200:2+200*3);
--> M1 * M2

>>ans  =

   3.411429    12.697324    26.081989    60.998407
   3.411429    12.697324    26.081989    60.998407
```

j) LAST COLUMN / COLUMN SUBTRACTION

```
--> VEC = [1:1:6];

--> VEC(1,$) = 40;

--> VEC

>> VEC  =

   1.   2.   3.   4.   5.   40.

--> VEC(1,3:$) = []

>> VEC  =

   1.   2.
```

k) ADDING NEW ROW TO THE MATRIX / MULTIDIMENSIONAL MATRICES

```
--> MAT = [1 1 1; 1 1 1];

--> ROW = [2 2 2];

--> MAT($+1, :) = ROW

>> MAT  =

   1.   1.   1.
   1.   1.   1.
   2.   2.   2.

--> HYPERMAT  = zeros(3,3,3);

--> HYPERMAT(:,:,2) = MAT

>> HYPERMAT  =

(:,:,1)

{3x3 : 0}

(:,:,2)

   1.   1.   1.
   1.   1.   1.
   2.   2.   2.

(:,:,3)

{3x3 : 0}
```

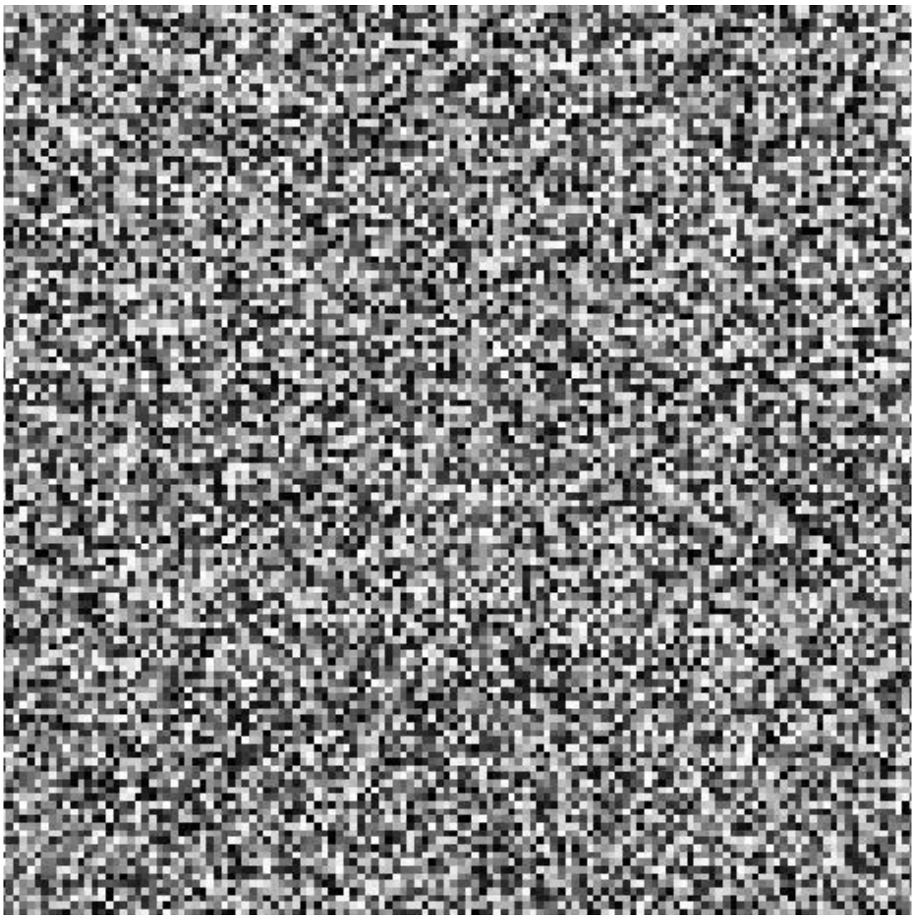
Exercise 1: Basic operations in Scilab for image processing

You should use Scilab's **SciNote** editor for exercise and try to avoid Command Window. We will write many lines of code and it is easy to do some mistakes if you process everything "on the fly". SciNote is hidden behind 'Applications' on the toolbar.

Scilab functions needed for exercise: round(), rand(), figure(), length(), uint8(), double(), for loop (syntax example in exercise), if statement (syntax example in exercise)

IPD functions needed for exercise: ShowImage(), ShowColorImage()

1) Create variable „m". „m" should be a random numbers matrix of size (128 x 128) and maximum possible value of 255 (limit of the 8-bit digital images).
2) Create new figure. Show variable „m" as the grayscale image. You should obtain image like this:



3) Create 3-dimensional matrix of zeros „M". „M" size should be equal to (128 x 128 x 3).
4) Change the first layer of the matrix „M" and put there small matrix „m".
5) Create new figure. Show variable „M" as the color image. What do yousee?
6) Create vector „v" of length 256 (from 0 to 255).
7) Create matrix of zeros „W" of size (150 x vector length x 3).
   a) Using a for loop and if statement for filling rows from 1 to 50 of the first layer with values provided by vector „v".

b) Then inside for loop create elseif statement for filling rows from 51 to 100 of the second layer with values provided by vector „v”.

c) At the end inside for loop create else statement for filling rows from 101 to 150 of the third layer.

You should obtain a 3D matrix similar to this:

0 1 2 3 0 0 0 0 0 0 0 0
0 1 2 3 0 0 0 0 0 0 0 0
0 1 2 3 0 0 0 0 0 0 0 0
0 0 0 0 **0 1 2 3** 0 0 0 0
0 0 0 0 **0 1 2 3** 0 0 0 0
0 0 0 0 **0 1 2 3** 0 0 0 0
0 0 0 0 0 0 0 0 **0 1 2 3**
0 0 0 0 0 0 0 0 **0 1 2 3**
0 0 0 0 0 0 0 0 **0 1 2 3**

Example with for loop and if statement:

```
>example_row = [1 2 3 4 5 1 2 3 9];
>
> L = length(example_row);
>
>fori = 1:1:L
>    ifexample_row(i) == 1 then
>    disp(„first”)
>    elseifexample_row(i) == 2 then
>    disp(„second”)
>    else
>    disp(„else”)
>    end
>end
```

8) Create new figure. Show matrix „W” as a color image. What is wrong in your opinion?
9) Create new matrix „W8” the same as the „W” with the exception of data type: change the type of matrix „W” to 8-bit unsigned integer. Create new figure. Show new matrix derived from „W” as a color image. Compare this figure to the previous figure.
10) Create new matrix „WD” as the matrix „W” divided by 255. Create new figure. Show new matrix derived from „W” as a color image. Compare this figure with the previous figures.

Exercise 2: Basic image operations and noise

In this chapter we will do some basic image processing operations. Processed image may be downloaded from here: https://github.com/szymonMolinski/Scilab-workshops.

Landsat 8 view, raw multispectral image (composite of bands 4-3-2)

This image is very good for the first steps in image processing. It has smooth areas (fields) and more detailed urbanized region. It is very dark so some preprocessing steps are needed.

The good practice is to start each script with clearing the console then clearing all variables and then to make stacksize of max possible value. Stack size is limited in Scilab 5.4.1 which is really annoying because all images consumes many bytes of stack memory (the upper limit is not depended on the user's hardware).

(Important fact is that Scilab 6.0.0 has much better memory management and I recommend you to start using it after tutorial when Image Processing Toolbox will be released).

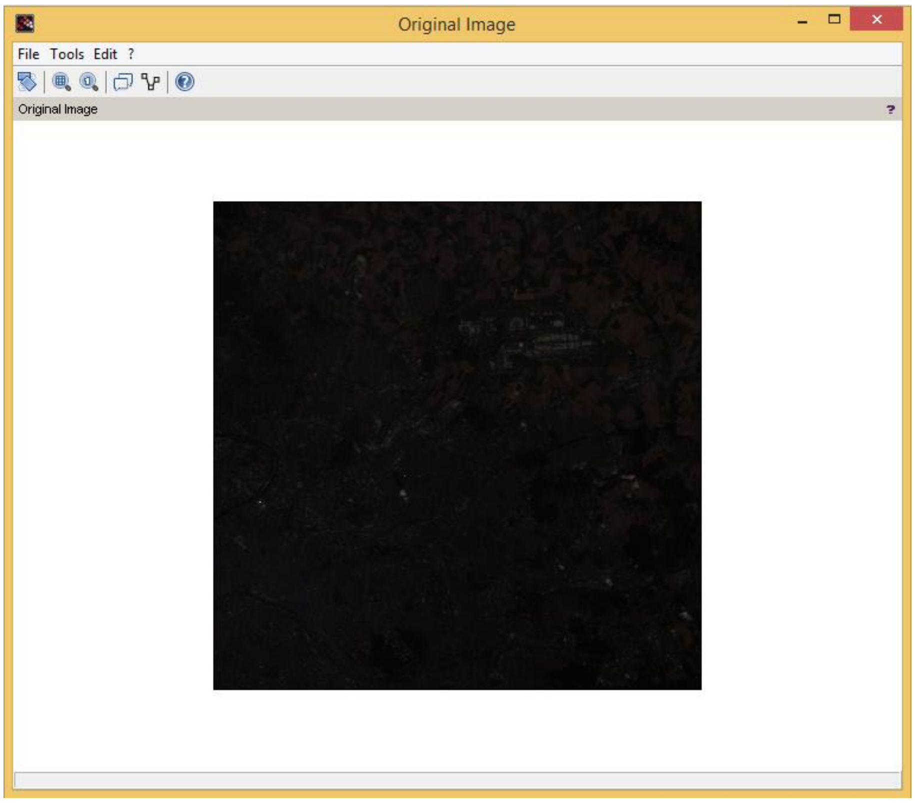
The first three lines of code:

```
>clc// Clears console window
>clear // clears variables from memory
>stacksize('max') // Max stacksize for operations
```

Then we can read our image as the multidimensional matrix and show it in the new window:

```
>I = ReadImage("path_to_your_image");

>// Show image

>figure(1)
>ShowColorImage(I, "Original Image");
```

Displayed image will be look like:



Usually preprocessing steps are done only on the 2D images (or spectral channels) as the layers of the more dimensional objects. That’s why in the next step we are using **RGB2Gray**and convert our data from 3D color image to 2D grayscale image. Important thing is to convert gray image to the double type, because we will perform some operations with real numbers. If we will use integers then the division by zero error will appear…

```
>// Make image gray and show it in the new window
>gray_I = RGB2Gray(I);
>gray_I = double(gray_I);

>figure(2)
>ShowImage(gray_I, "Gray Image");
```

At this point we have grayscale image and nothing else… Usually digital images are noisy. And this is the reason of this exercise: to measure the difference between corrupted and original images.
We created Gaussian white noise in the first exercise. Here process is similar. Then we add this noise to the gray image (as the weighted sum) and show our noisy image.

```
>size_image = size(gray_I);
>noisy_matrix = (round(rand(size_image(1),
size_image(2))*255));
>noisy_image = round((0.98*double(gray_I) +
0.02*double(noisy_matrix)));

>figure(3)
>ShowImage(noisy_image,  "noisy image")
```

Now our base image is very dark and noisy… but because of the darkness it is hard to see any differences between original image and noisy one with the naked eye.

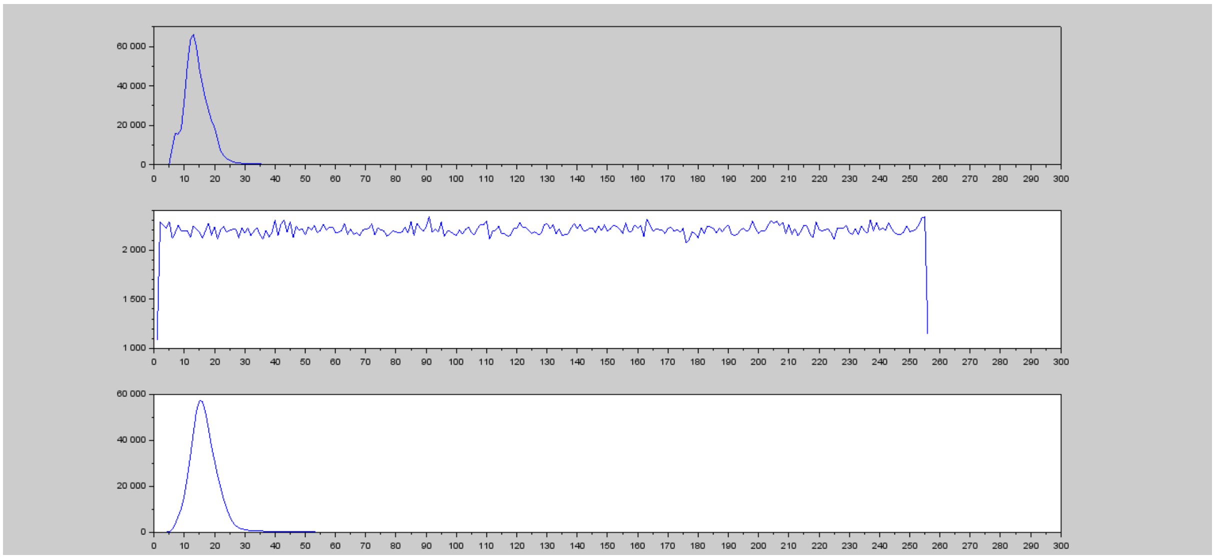
On the left: orginal image. On the right: image corrupted by Gaussian white noise.

The better comparison may be done by using a histogram for visualization of distribution of the brightness values. The horizontal axis of histogram shows all possible values of pixels on image. Vertical axis shows number of pixels of given value.

Histograms are powerful tools and gave us many information about image content. Distribution of values may tell us about contrast of the image or local properties of some image regions. In our case we take a look to histograms for better understanding how white noise distribution look like and how it affects "normal" or uncorrupted image.

We will use **CreateHistogram()** function from IPD toolbox for histogram counting and **subplot()** method for creation of many plots in the same graphical window.

```
>hist_base = CreateHistogram(uint8(gray_I)); // Important!
Change the type of the image to the uint8 to get the proper
values

>hist_noise = CreateHistogram(uint8(noisy_matrix));
>hist_output = CreateHistogram(uint8(noisy_image));

>figure(4)
>subplot(311) // subplot(3: numer of rows, 1: numer of
columns, 1: subwindownumer

>plot(hist_base)
>subplot(312)
>plot(hist_noise)
>subplot(313)
>plot(hist_output)
```



Up: Histogram of original image. Middle: Histogram of Gaussian white noise. Down: Histogram of original image with additional noise. Distribution is more smooth. As you can see image is dark because values are distributed near the left side of histogram.

There is also more analytical way to compare two signals (images). You may use Mean Squared Error and/or Peak Signal to Noise Ratio (MSE / PSNR).

MSE is defined as:

$$MSE = \frac{1}{M * N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - K(i,j)]^2,$$

where:

MSE: Mean squared error,
M:Image width,
N:Image height,
(i,j): Pixel coordinates,
I:Base image,
K:Noisy image.

PSNR (in dB) is defined as:

$$PSNR = 10 * log_{10}\left(\frac{MAX_I^2}{MSE}\right),$$

where:

PSNR: Peak Signal to Noise Ratio,
MAX$_I$: Maximum value of pixel in given range (usually 255, may be also 1 if scale is 0 to 1 double),
MSE: Mean Squared Error.

The next steps of exercise are related to the equations above. Try to do it alone.

Scilab functions needed for exercise: round(), sum(), log10(), disp()

Additional Scilab functions which may be used: double(), for loop

1) Create two more corrupted images from matrices which you already have. The first image should be created as the weighted sum of noise matrix and base image in the ratio 5% of noise and 95% of original image. The second image ratio should be 10% for the noise and 90% of the base content. At the end of this process you have 3 corrupted images.
2) Calculate MSE and PSNR between original image and noisy images. Display MSE and PSNR values.

Expected output:

```
--> MSE

>> MSE  =

         7.5659218    46.57874    185.9844

--> PSNR

>> PSNR  =

          39.342185    31.448926    25.436038
```

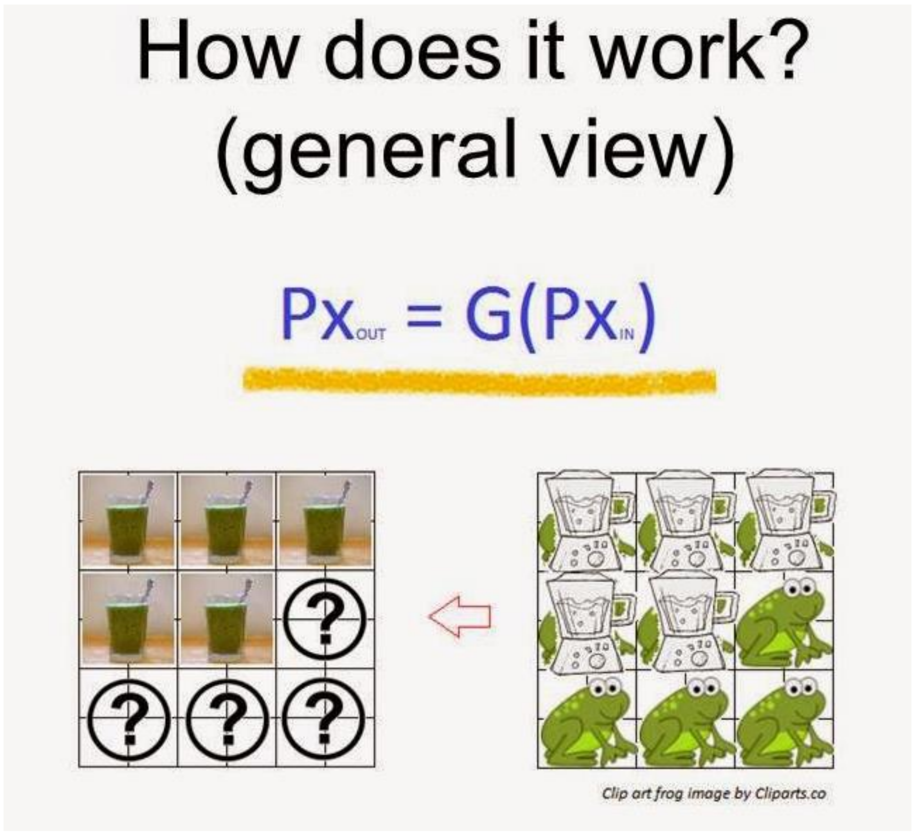
Exercise 3: Point operators and filtering

This exercise is up to you. You can do it with the knowledge from the previous examples. Exercise has two parts: enhancement (brightening) by point operations and filtering by local filters.

In image processing you may treat image as a set of individual pixels independent of each other or as a set of "patches" of pixels and their neighborhood or as a big uniform canvas. In this exercise you will learn something about:

- point operators (processing each pixel independently),
- local operators (convolving image content by special mask).

Point operators are really simple. The general idea of their work is explained below:

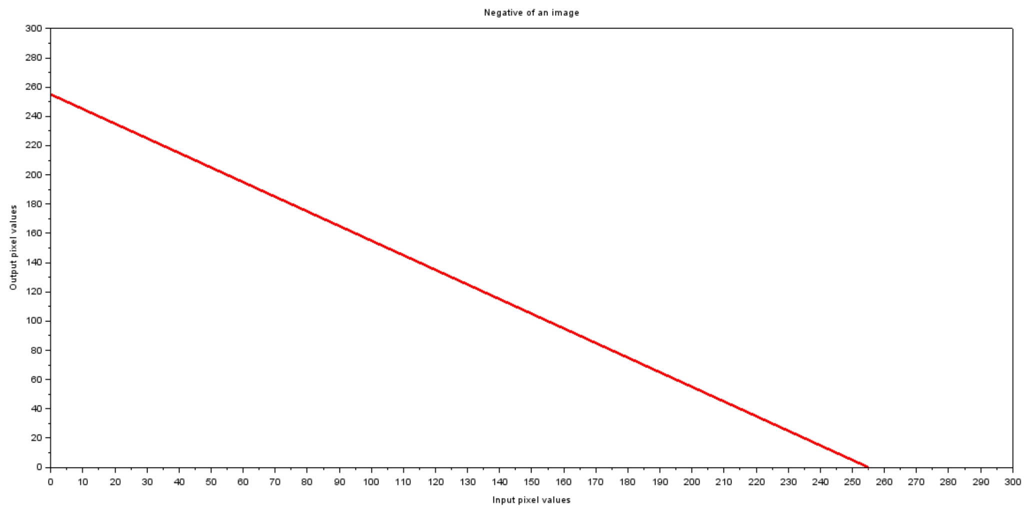


You take each frog alone (pixel from base image), put it in the mixer (some function) and as the output you obtain a green smoothie (output pixel).

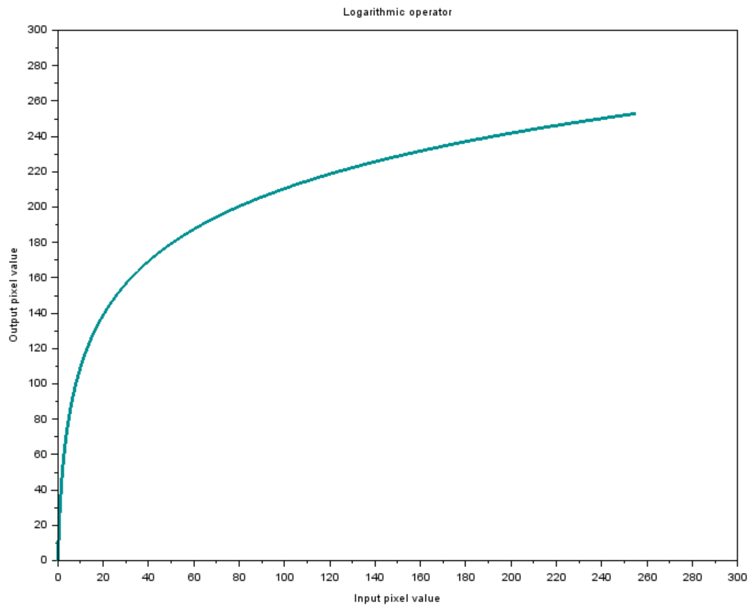
The easiest operation is a negative of an image and it can be done in Scilab very quickly, by one line of code:

-->negativImage = max_value_of_pixel_space – base image;

Plot of the negative is also very simple, for the 0 input pixel we obtain 255 on the output and for the 255 input pixel we obtain 0 on the output (for uint8 type):



In this exercise we must do something with many low-level values on the image because it is too dark. For this we will use logarithmic function which response look like:



At the beginning values raise quickly and brighten dark pixels:

INPUT	0	1	2	3	4	5	6	7	8	9
OUTPUT	0	31	50	63	74	82	89	95	101	105

And at the end they slow down:

INPUT	246	247	248	249	250	251	252	253	254	255
OUTPUT	253	253	253	253	254	254	254	254	254	255

With this knowledge we may go through the exercise.

Scilab functions needed for exercise: floor(), log10(), size(), double(), uint8(), round(), rand(), ones(), conv2(), sum(), for loops, clc, clear, stacksize('max'), figure()

IPD functions needed for exercise: ReadImage(), ShowImage(), ShowColorImage(), WriteImage(), MedianFilter()

1) Clear the console from all messages. Clear all variables from memory. Set stacksize to max value.
2) Read image **OLI234.jpg.**
3) Create new variable and copy to it OLI234.jpg and convert it to type double. Add one (1) to all matrix elements to prevent logarithmic operations on zeros.
4) Split RGB channels of your new variable. Use for this Scilab syntax:

```
>Rchannel = hypermat(:,:,1);
>Gchannel = hypermat(:,:,2);
>Bchannel = hypermat(:,:,3);
```

5) Process all pixels in each channel with the given logarithmic function [3]:

$$processed\_channel = c * \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} log10\big(I(i,j)\big),$$

where:

I(i,j): pixel in location i,j of image matrix,
M,N: width and height of the matrix,
c: constant scalar derived from equation:

$$c = \frac{MAXval}{log_{10}(MAXval + 1)},$$

where:

MAXval: 255 for uint8 (scale from 0 to 255).

Hints:

a) You may calculate ***c*** as the individual variable. It should be equal to 105.88646.
b) You can perform all calculations by the single line of the code… but Scilab's stack cannot handle this data. From the other side you may use for loop and go through the matrix element by element but this will be long operation. But there is also other possibility: to use for loop AND Scilab's syntax. You can process each ROW in the loop as the whole.

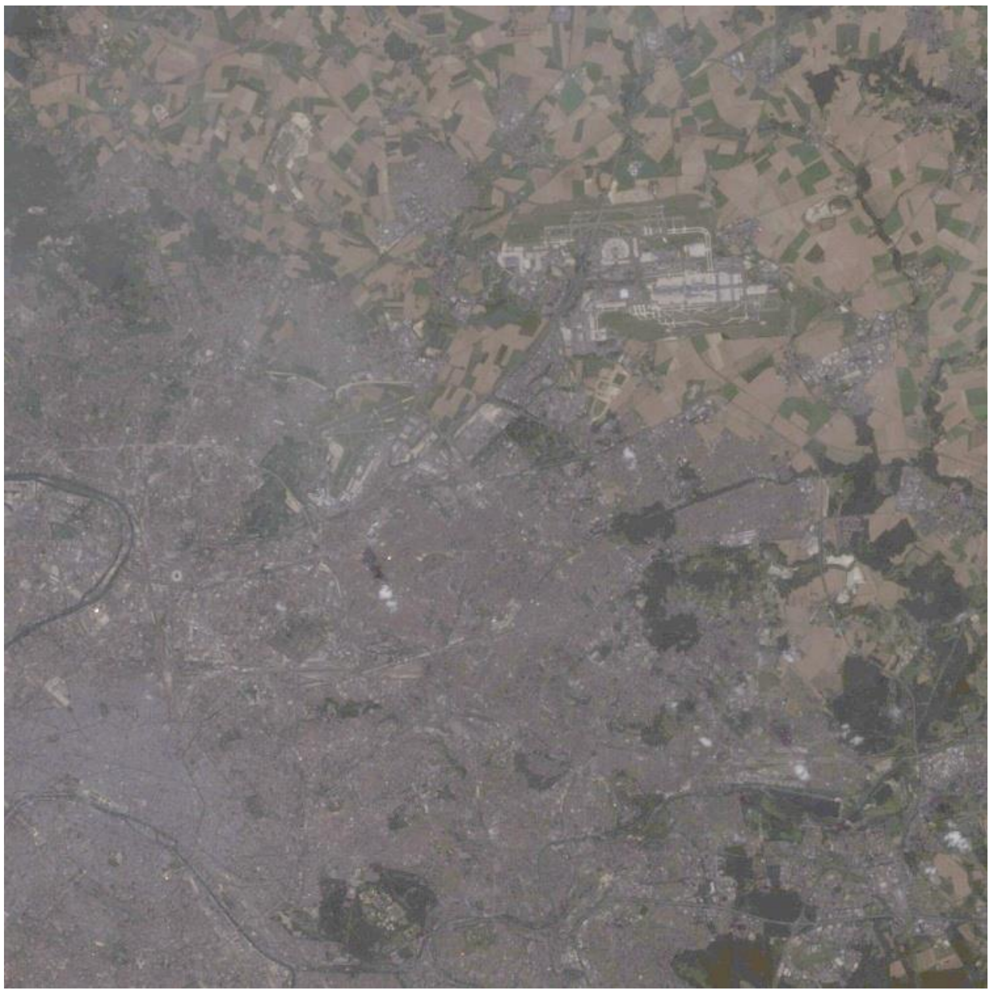
c) This loop will be look like:

```
>for i=1:1:numer_of_rows
>processed_image(i,:)=c*log10(processed_image (i,:));
>end
```

6) When you will process each channel put them into your RGB hypermatrix. Until this moment all your variables should be of a double type. For proper showing and writing it you should change the type to uint8. The best option is to change it *only* inside visualizing and writing functions like here:

```
>WriteImage(uint8(some_image), "path_to_save.jpg");
```

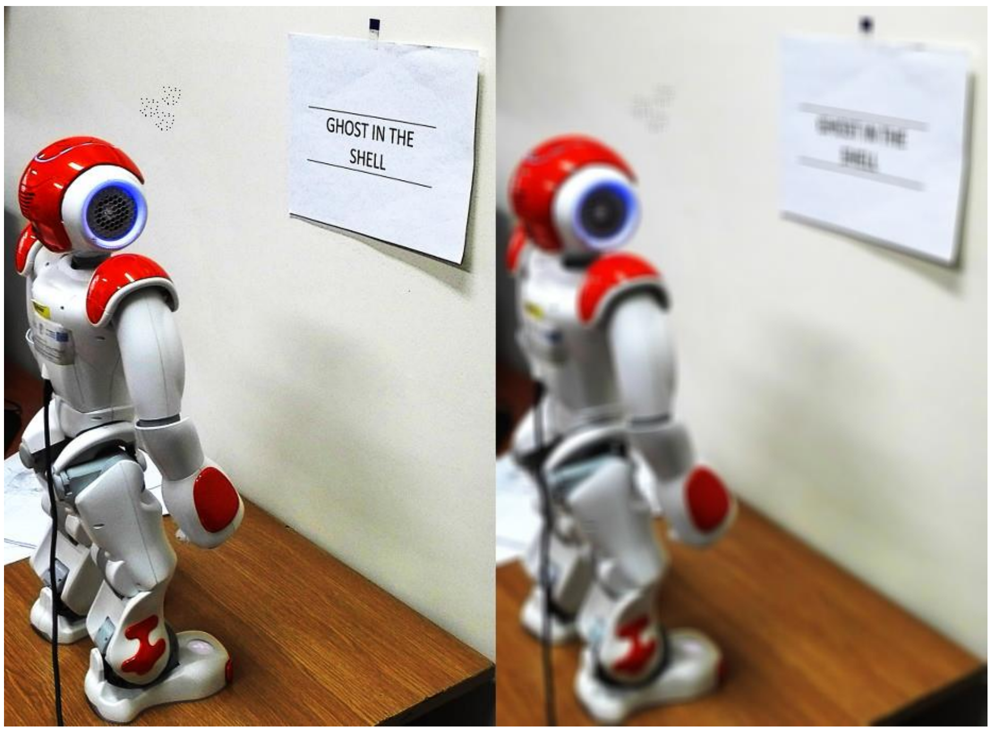
7) As the output you should obtain this image:



In the second part of this exercise you will use image filters: average (mean) filter and median filter. They are also local operators because output pixel value depends on the input pixel at the same location and neighborhoods of this pixel.

The mean filter is usually 3x3 matrix (sometimes larger – 5x5 or 7x7). You may pass it through image many times and blur it more and more [4].

The example of mean filter:

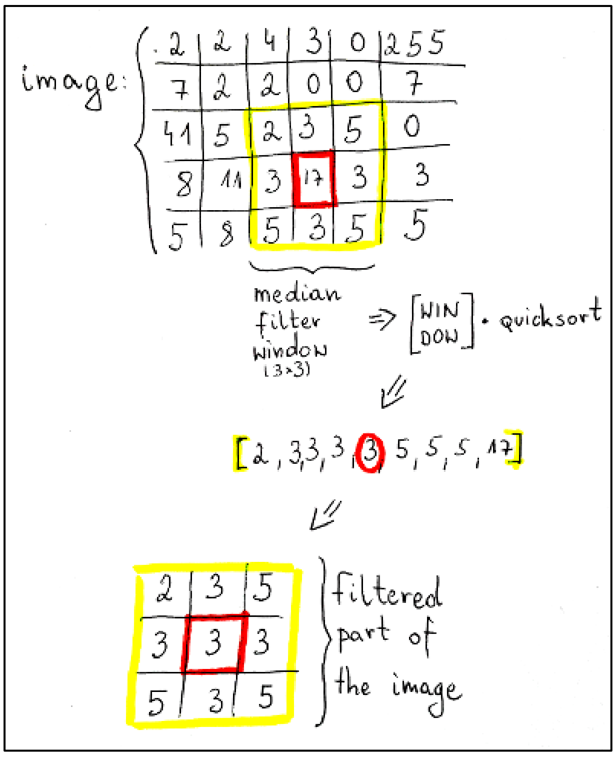


On the left: base image. On the right: image after processing by mean filter with the kernel size 9x9. The noisy region above robot's head (group of black pixels) is blurred but not removed.

Mean filter is convolved with the image (Scilab has function for this: **conv2()**). *Convolution is the process of multiplying each element of the image with its local neighbors, weighted by the kernel. For example, if we have two three-by-three matrices, one a kernel, and the other an image piece, convolution is the process of flipping both the rows and columns of the kernel and then multiplying locationally similar entries and summing* – and this nice description comes from Wikipedia with additional example how convolution work [5].

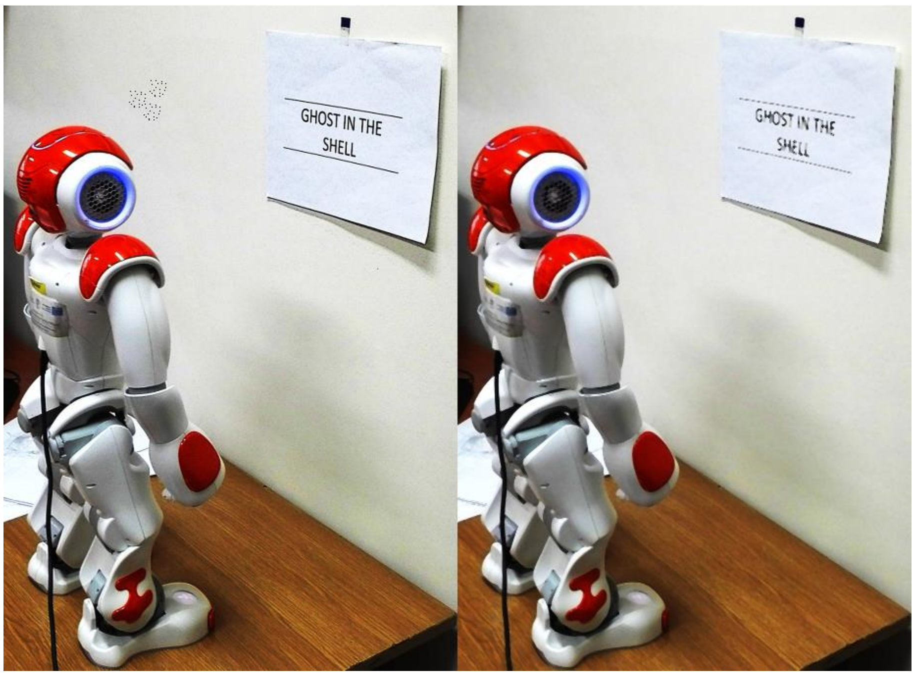
Mean filter may be used as the simplest method for reducing noise on image as well as for smoothing high frequency components when they may produce false edges.

The good choice is to forget about mean filter and use median filter. Algorithm is mathematically simpler than algorithms for other filters because there is no operation of the convolution. Although computationally is more expensive (because we must check all matrix elements and sort them) [6].



Median filter may be used for filtering different types of noise in general but it is the best choice for images corrupted by '*salt and pepper*' noise. This is not common type of noise. It appears when a CCD matrix pixel is corrupted (overexposed) or when the data transmission is unstable. A situation of that kind occurs with the satellite imagery (data transmission errors) or MRI scans.

Median filter works very well with the salt and pepper noise. Example of median filter is below:



On the left: base image. On the right: image after processing by median filter with the kernel size 3x3. The noisy region above robot's head (group of black pixels) is removed. The most of the image content are preserved but horizontal lines and text on the paper sheet on the wall are eroded.

We will apply mean and median filter to the image corrupted by Gaussian white noise and calculate MSE and PSNR values for different filters and filter sizes.

8) Change your final color image from RGB to Grayscale (and preserve the double type). Then create matrix of random numbers. This matrix should have same size as your gray image and values inside should be rescalled to interval between 0 and 255 and then rounded.
9) Create new corrupted image (8% of noise and 92% of your base gray image).
10) Show this noisy image in the new figure.
11) In this step you will create average filter kernels. The 3x3 kernel is provided below, your role is to create additional 5x5 kernel and 7x7 kernel:

```
> kernel3 = ones(3,3).*1/(3^2); // size 3x3, all elements
divided by (1/3^2) -> sum of elements must give 1
```

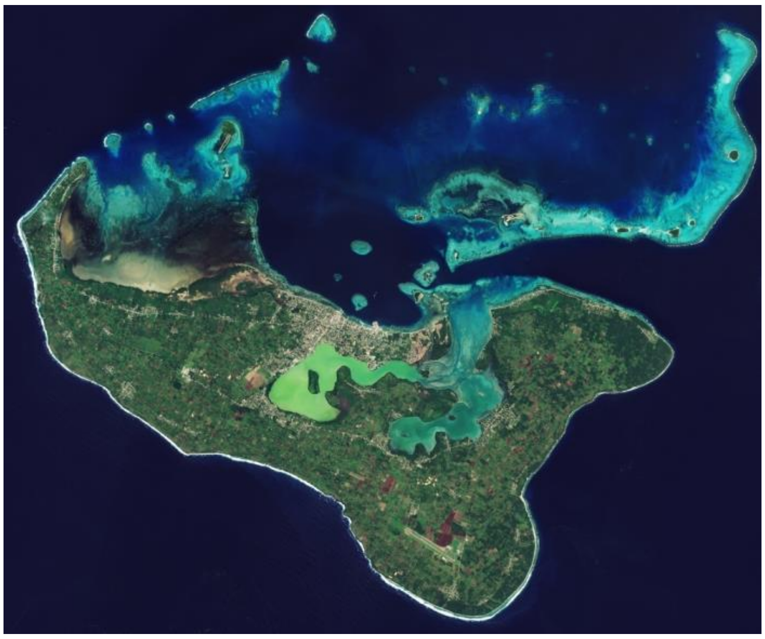
12) When your kernels are prepared you may start convolution with the base noisy image. Create three new images. The first image is the result of convolution base noisy image with kernel 3x3. The second image is the output of convolution of base noisy image with kernel 5x5 and the last one with kernel 7x7. Third parameter in conv2() function should be a string „***same***” to preserve size of the base image.
13) Then create additional three images – for median filter. Process is similar to this from the previous point with small exceptions: there is no need for the kernels and operation is called from the IPD function **MedianFilter**(noisy_image, [x x]).
14) Now you have 7 corrupted images and one without any noise. One image is noisy image, three images are a noisy image processed by mean filters and other three images are a noisy image processed by median filters.
15) Calculate MSE and PSNR between these 7 corrupted images and original grayscale image. You should obtain this output:

```
--> MSE

>>36.386131   24.708048   47.628276   64.936506   16.180592   31.322727   41.881703

--> PSNR

>>32.521445   34.202419   31.352155   30.005914   36.040859   33.172208   31.91056
```

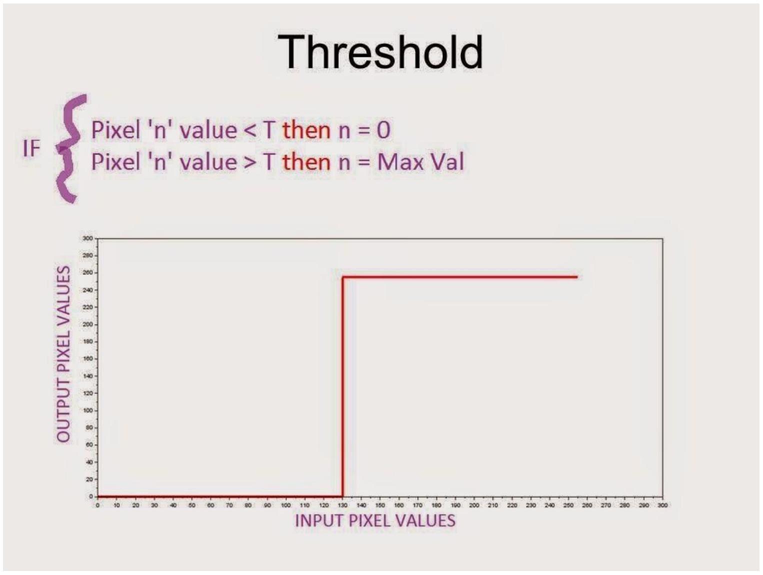
Exercise 4: Segmentation and morphological operations

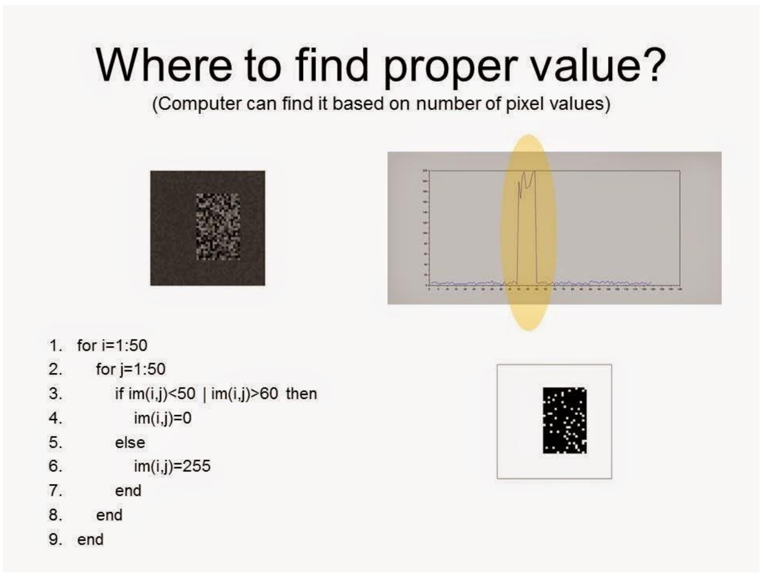
Segmentation is an important topic, widely used in an object classification and image enhancement. In this exercise we will use thresholding (the simplest segmentation method) and dilation for separate large unnecessary areas from the image content.

We will use Tonga Island satellite imagery provided by ESA [7]. Rescalled image is availble here: https://github.com/szymonMolinski/Scilab-workshops



Threshold is described by simple linear function. I have read in some discussion that this simple threshold is not used in practice (substituted by adaptive threshold or Otsu's method). But this is not entirely true: in practice is even better to use the simplest and the FASTEST function. Everything depends on the scene characterization but with a good lightning and high contrasts don't think about anything else than simple binary thresholding.





Where to find proper value?

(Computer can find it based on number of pixel values)

```
for i=1:50
   for j=1:50
      if im(i,j)<50 | im(i,j)>60 then
         im(i,j)=0
      else
         im(i,j)=255
      end
   end
end
```

How to find proper "T" value? Information about that are plotted on histogram. As you can see on the illustration above a histogram has peak between 50 and 60. This peak is a background of an artificial image. Function for thresholding (naive implementation) is setting all pixel values smallest than 50 and higher than 60 to 0 (black) and between them to 255 (white). Black region is the Region of Interest (ROI) - it can be white or gray - you choose output value for thresholding.

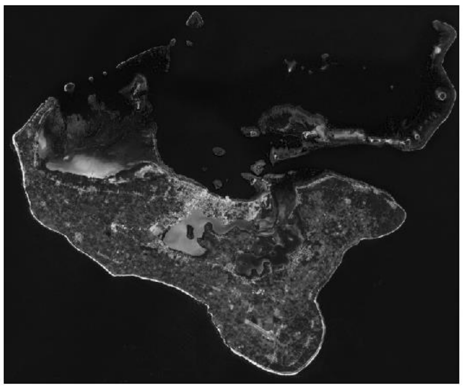
This kind of operations works very well with the regions of great homogeneity of intensity values but if your image is full of details it is hard to find threshold value from the global histogram. Then you can use local thresholding instead of global [8].

Ok, enough theory. Let’s practice!

Scilab functions needed for exercise: double(), figure(number), uint8(), subplot(), plot(), size(), zeros(), dsearch(), max(), if statement, ceil() , clc, clear, stacksize(‘max’)

IPD functions needed for exercise: ReadImage(), ShowImage(), ShowColorImage(), CreateHistogram(), CreateStructureElement(), DilateImage()

1) Use **clc** / **clear** and set **stacksize** to maximum value.
2) Read new image as the double type.
3) Split image into different channels.
4) Show each channel (as the uint8 grayscale image). You should obtain these views presented on the next page. Our task is to delete not interesting content – ocean waters. It may be done on converted RGB to Gray image but usually it is better to work with different channels when you have access to the multispectral images. Why? Because interesting content may be better visible in different wavelengths. In our case the best image for deleting the water is red channel: difference between ocean and land is large in this case.



Images of red (up), green (middle) and blue (down) channels.

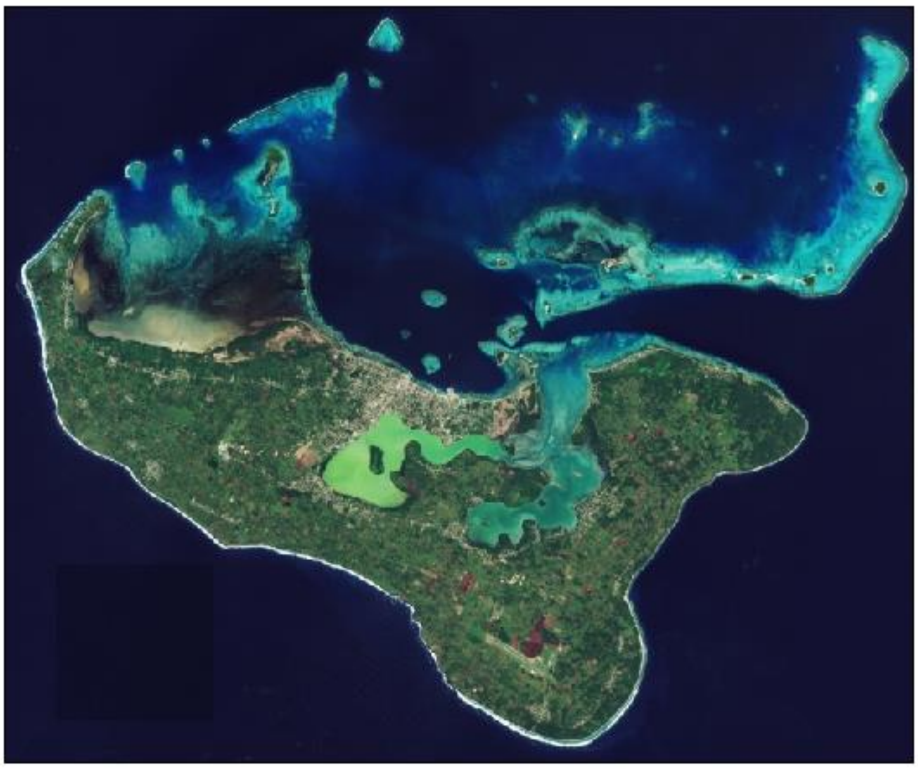
5) There is the other way to find values for deleting. You can do it with a histogram (also automatically). But before that we will cut a region of interest from our images and compare histograms of each channel as a whole to the part representing only region of interest. Our ROI limits are presented below:

```
>areaRow = [430:1:430+120];
>areaCol = [40:1:40+120];
```

And the visualization of this region may be done with this code:

```
>figure()
>aI = RGB_image;
>aI(areaRow, areaCol,:) = round(aI(areaRow,
areaCol,:)*0.8);
>aI = uint8(aI);
>ShowColorImage(aI, "ROI");
```

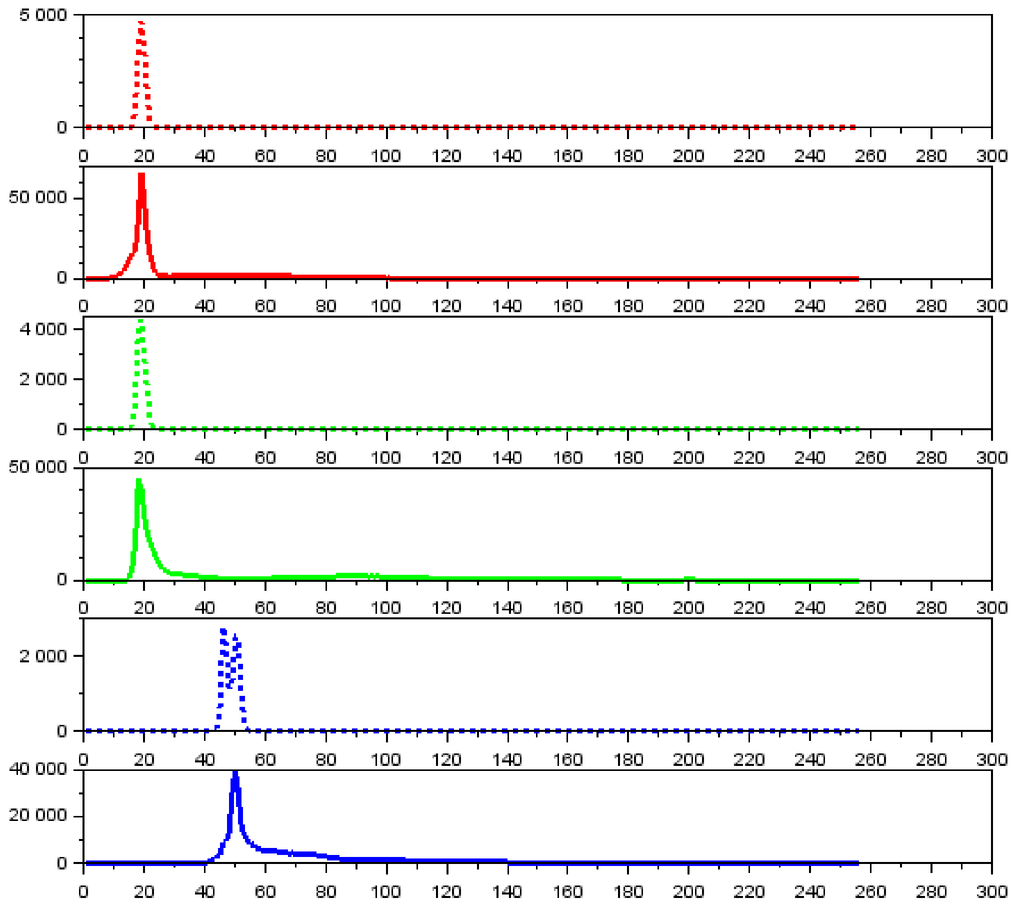
We can see which region we choose for our comparison (darker rectangle in the down-left side of image):



6) Based on the ROI limits create 6 histograms and show them on the subplot. Three histograms for ROI of each channel and three histograms of each channel as a whole. The example for the red channel is presented below:

```
>hist_red = CreateHistogram(uint8(red_channel(areaRow,
areaCol)));
>hRbig = CreateHistogram(uint8(red_channel));
>figure()
>subplot(611)
>plot(hist_red)
>subplot(612)
>plot(hRbig)
```

As the output you should obtain:



Histograms of ROIs and channels of processed image. From up: ROI of red channel, red channel, ROI of green channel, green channel, ROI of blue channel, blue channel.

The peak for the whole image is in the same place as for the ROI located above an ocean. The conclusion is that sea area is the biggest one on the image. We may choose red or green channel as the histograms show. What is the value of threshold in your opinion? Try to find it in the next steps.

7) You have sliced images and histograms. Probably you choose one channel for segmentation. Now you must create threshold function and this is not an easy step. You will use **dsearch()** function, **for** loop and **if** statement. You should also take a use from the **ceil()** method. And don't forget about **zeros(),size()** and **max().**

   Hints:

   a) dsearch() may be overloaded for finding values in matrix which are the same as the base values. How does it work? Example below.

   ```
   --> MATRIX = [1 2 3 4 5; 1 2 3 4 5; 5 4 3 2 1; 1 5 2 4 3];
   -->value_vector = [3:1:5];
   -->[i_bin, counts, outside] = dsearch(MATRIX, value_vector);
   -->i_bin
   >>0.   0.   1.   1.   2.
    0.   0.   1.   1.   2.
     2.   1.   1.   0.   0.
    0.   2.   0.   1.   1.
   ```

   "*i_bin*" matrix shows where are the same values of *value_vector* in *MATRIX*. But their exceeds 1 (when larger vectors are taken into account they are much larger) and for proper threshold we need 0 – 1 values.
   That's why we must postprocess such matrix:

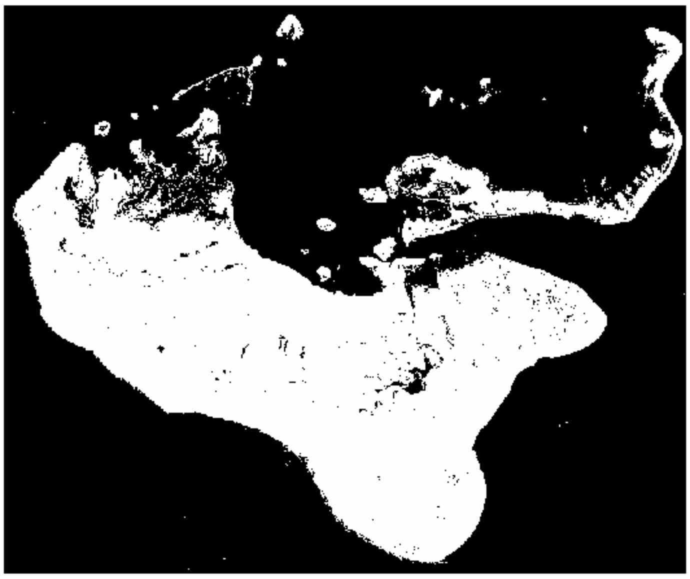
   ```
   -->i_bin = ceil(i_bin/max(i_bin));
   -->i_bin
   >>   0.   0.   1.   1.   1.
        0.   0.   1.   1.   1.
        1.   1.   1.   0.   0.
        0.   1.   0.   1.   1.
   ```

   b) Do you remember about processing only rows of the matrix in the for loop? This is the good moment for this.

   c) Your value vector should have values which are not related to the ocean (for this exercise). It is builded as **vector = [down_limit : 1 : 255]**.

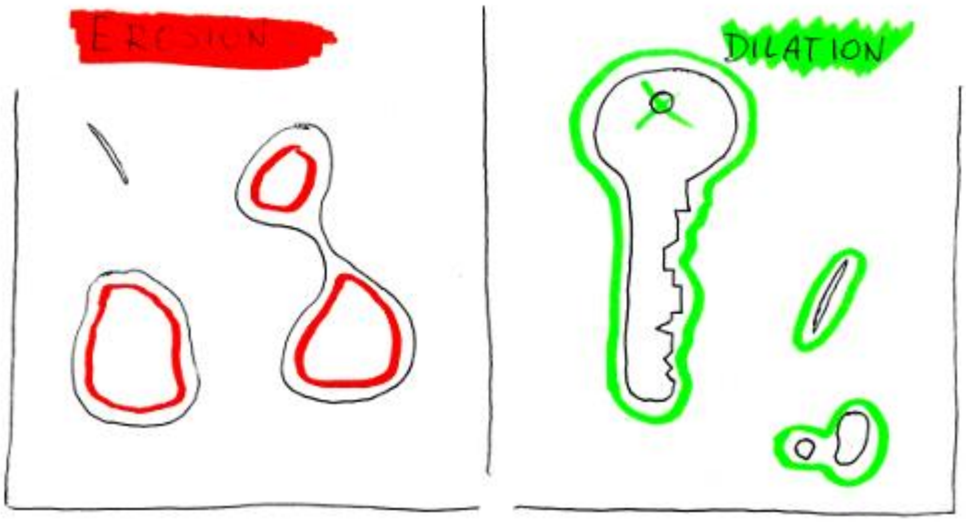
   d) You can optimize calculations by two things. First of all you may preallocate memory for the matrix which will show threshed values (outside the for loop). Then inside for loop you may check each i_bin vector max value. If this is zero you should copy row immediately to your preallocated matrix. Otherwise use the equation with ceil function.

8) Show your image as the grayscale double. Your output should be similar to this:



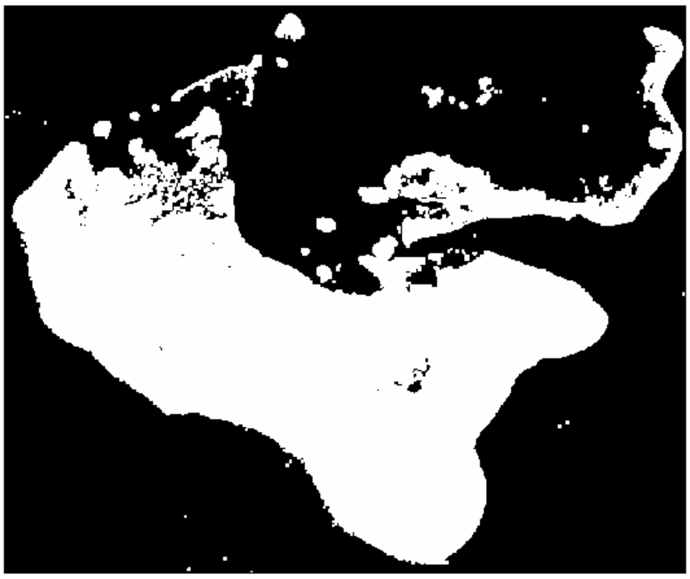
9) Do you see small black holes inside the island? We can delete them by fast morphological operation – dilation.

10) There are two basic morphological operations: dilation and erosion. Dilation and Erosion are morphological filters and they are the core of the much larger group of morphological operations. Dilation and erosion are used for removing larger details. Objects after erosion are smaller. The smallest objects may be removed from the image. Also connections between them may disappear (which may be good or not). Dilation is opposite to erosion. Small objects are expanded, new connections are created, holes are filled and detailed edges are polished [9].



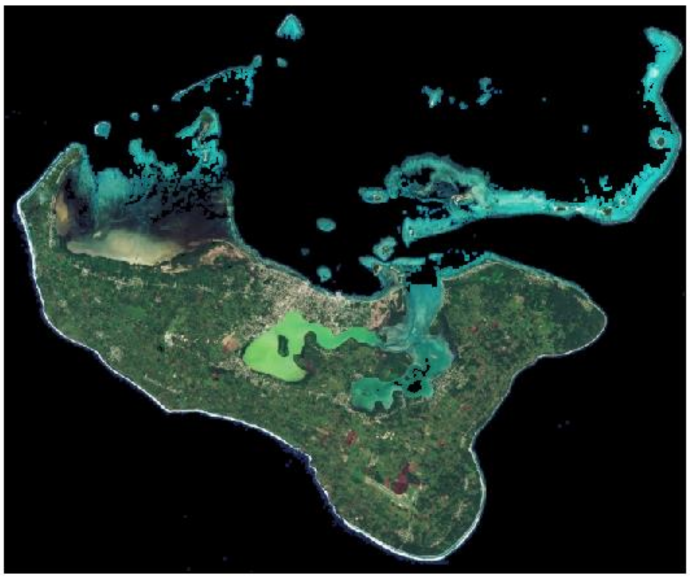
Erosion: objects are disconnected, small objects are removed. Dilation: objects are expended and merged, holes are filled.

We will use dilation (one time). Dilation need kernel – usually as a square matrix, but for some application you may use circle or ellipse kernel.
In this step you should: Create the structure element as a 'square' of size 3x3 and dilate the threshed image by it. Then show it. You should obtain this image:



11) Convert dilated image to the uint8. Multiply each element of dilated image with each unprocessed channel of the base image.

12) Merge channels and show a color image:



Exercise 5: Segmentation methods (part 2)

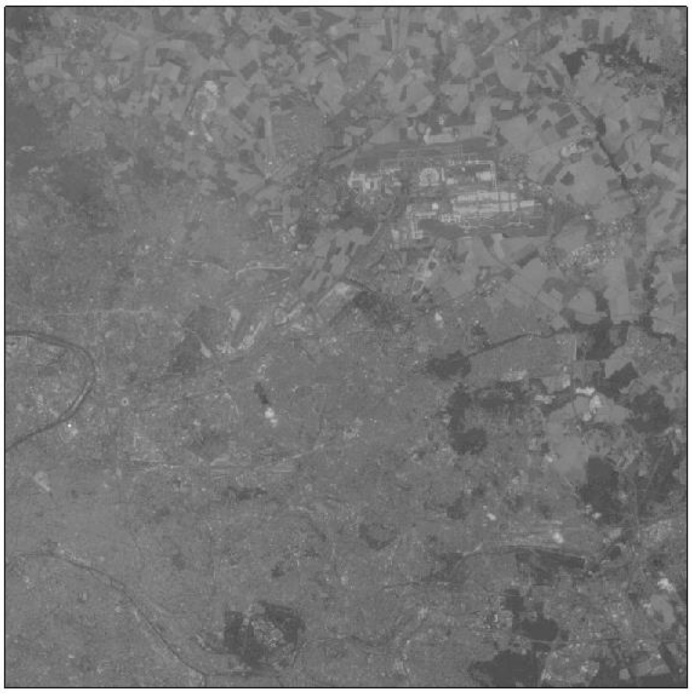
This exercise is heavily based on the tutorial provided with the IPD Toolbox [10]. We will focus only on the experiments in this part – if you want to know more, go through the official IPD tutorial.

Purpose of this part is to show you many more functions of IPD but also to build your awareness about limitations of some methods when we spare with the remote sensing data.

Scilab functions needed for exercise: double(), figure(number), uint8(), jetcolormap(), length(), unique()

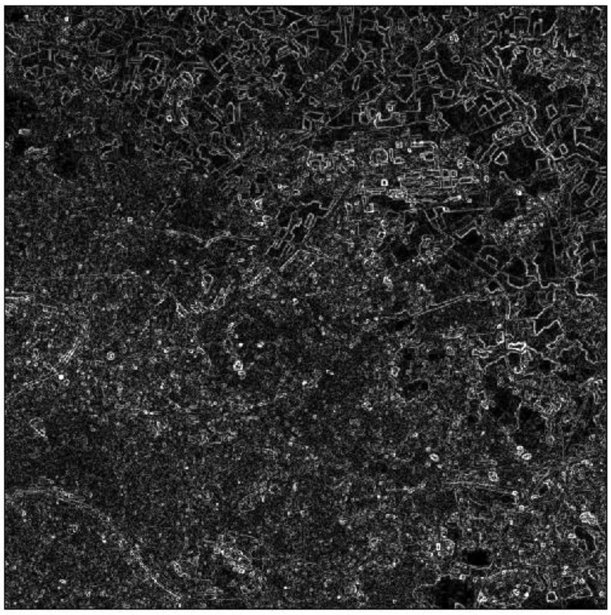
IPD functions needed for exercise: RGB2Gray(), ReadImage(), ShowImage(), EdgeFilter(), EDGE_SOBEL, SegmentByThreshold(), DistanceTransform(), SearchBlobs(), Watershed()

1) For the first step of this exercise you probably know what to do from previous ones. (Clear Console, clear Variables, set stacksize to maxium).
2) Read processed image from the example 3 and convert it to the double, then convert it to the grayscale and show a grayscale image:

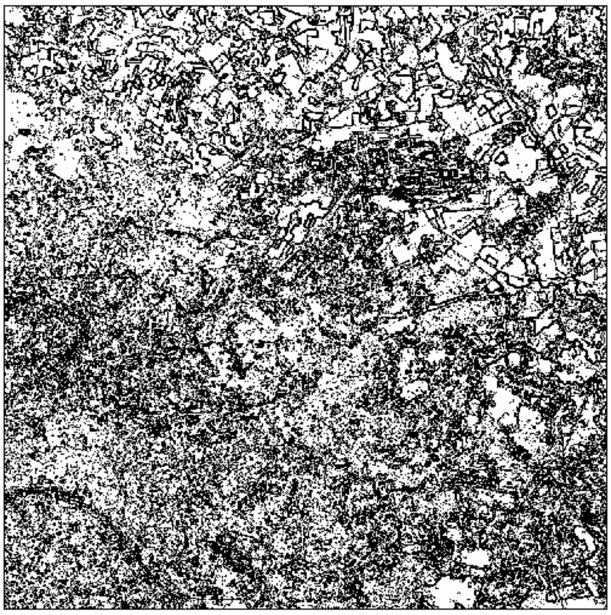


3) This is an edge detection time! But before that you need to set global variable "EDGE_SOBEL" (operator Sobel is used for edge detection [11]). Then we will apply thresholding to the output to preserve only edges.

```
// Edge detection and gradient image

global EDGE_SOBEL;
gradientImage = EdgeFilter(I, EDGE_SOBEL);

figure(2)
ShowImage(gradientImage, "Gradient Image");

edges = ~SegmentByThreshold(gradientImage, 40)

figure(3)
ShowImage(edges, "Edges Image");
```



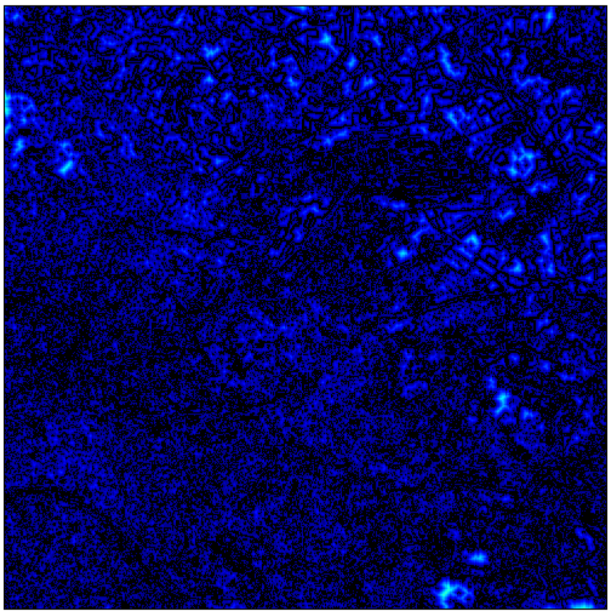
Sobel gradient image



Detected edges

4) In the next step we will create Distance matrix -> this is based on the morphological operation of distance transform [12]. We will use also Scilab Look-Up color tables for a better visualization of the data.

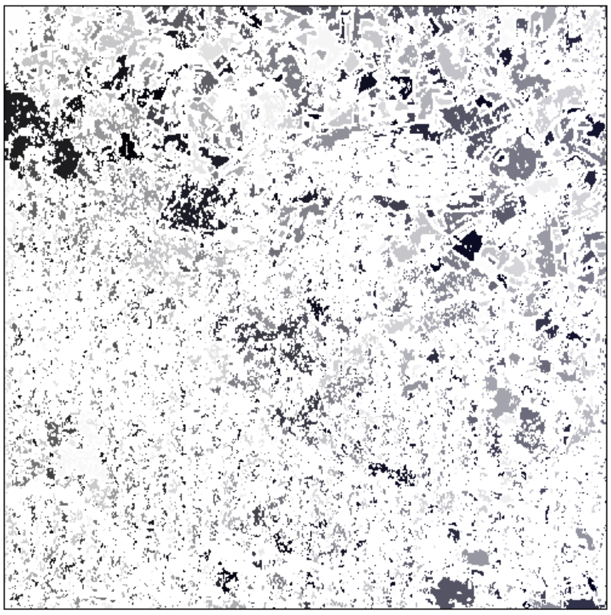
```
// Distance transform

distance = DistanceTransform(edges);

figure(4)
distance = double(distance);
ShowImage(distance, "Distance Image", jetcolormap(32));
```



Distance matrix. Brighter regions are more far away from the boundaries.

5) The next step is a blob detection (object detection). In this step you can change the second parameter of the segmentation operation to see the differences in the output classification image.

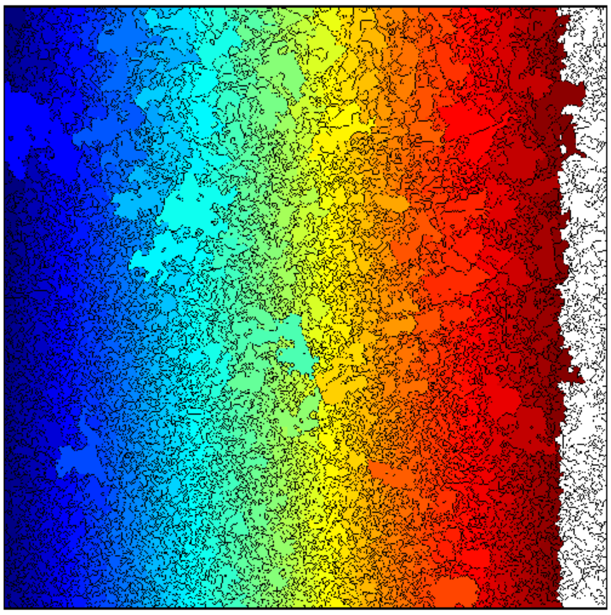
```
// Blob detection

thresholdImg = SegmentByThreshold(distance, 2);

marker = SearchBlobs(thresholdImg);

figure(5)
ShowImage(marker, 'Result of Blob Detection');
```



Markers (blobs) on the image.

6) The last step is to apply the watershed transform to the image [13]. We will use for this our gradient image and the blobs image.

```
water = Watershed(uint8(gradientImage), marker);

figure(6);

ColorMapLength = length(unique(water));

ShowImage(double(water), ...
          'Result of Watershed Transform', ...
jetcolormap(ColorMapLength));
```



Detected areas

7) Try to experiment in the step 5 with the higher threshold values and see what will be different in the last output image.
8) Congratulations! That is the end of your course. Now everything is up to you – Scilab 6.0.0 awaits for the newcomers ☺

References

References are grouped as they appear in the text:

[1] http://www.cecill.info/index.en.html ->CeCILL license

[2] https://creativecommons.org/licenses/by/4.0/ -> Creative Commons license

[3] http://homepages.inf.ed.ac.uk/rbf/HIPR2/pixlog.htm ->Logarithm Operator

[4] http://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm -> Mean Filter

[5] https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution -> Convolution

[6] http://homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm -> Median Filter

[7] http://www.esa.int/spaceinimages/Images/2016/07/Tonga-> Tonga Island, ESA

[8] http://homepages.inf.ed.ac.uk/rbf/HIPR2/threshld.htm -> Thresholding

[9] http://homepages.inf.ed.ac.uk/rbf/HIPR2/morops.htm -> Morphology

[10] https://atoms.scilab.org/toolboxes/IPD - > IPD Toolbox

[11] https://en.wikipedia.org/wiki/Sobel_operator -> Sobel Operator

[12] http://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm -> Distance transform

[13] https://en.wikipedia.org/wiki/Watershed_(image_processing) -> Watershed transform

Useful Links

1. SCILAB: http://www.scilab.org/
2. SCILAB – once again (with tutorials and news): http://scilab.io/
3. Image Processing: http://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm
4. My blog about image processing and programming in Scilab: http://imgsimon.blogspot.com/

Acknowledgments

I should like to stress that this is the first version of the tutorial. I will be very grateful if you – as a reader – send me your comments as well as all detected issues (write to s.molinski@protonmail.ch).

I would like to thank Ania Molińska (programmer and my companion) for taking a time to check this text.