IMAGE PROCESSING AND COMPUTER GRAPHICS
RULES

1. Contact:

Marek Kociński
room: 205, second floor (builing B9)
phone: 042 631 26 38
email: marek.kocinski@p.lodz.pl
web: www.eletel.p.lodz.pl/kocinski

2. Subject:

Laboratory 15 hours (weeks 3-6)

1. Monday 16.15 – 20.00, lab. 204
2. Tuesday 12.15 – 16.00. lab 204

Project 15h (weeks 7-10)

3. two hours per week for booth groups – fix a day and hour

3. Rules:

1. Projects should be developed **in Python (not in: C++, Java, Matlab)**

.Recommended: **Python Enthought** distribution (http://www.enthought.com/, free for academic) with many build in modules e.g.:

1.PIL – image processing
2.pyOpenGL – computer graphics
3.vtk – image processing and computer graphics
4.wxPython – Graphical User Interface
5.many, mnay more....

2. Full support and supervision will be available

3. Gropus consists of 2 persons

4. **The final report** should contain **conclusions** not only images. Report is obligatory!

1. Introduction and goal
2. Materials and methods
3. Results
4. Summary and conclusions
5. CD with the source code

5. All project **have to** be finished and passed **before 11th week!!** One week late is equal to 1 mark down!!!

6. Presence during classes is obligated.

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

Introduction to Python

Author: Marek Kociński

October 2010

1 Purpose

Python is powerful, easy to learn with effective object-oriented approach programming language. The Python interpreter and the extensive standard library are freely available in source or binary form for major platforms, and may be freely distributed. This instruction is based on tutorial presented in Python v2.6.6 documentation.

The goal of this exercise is to get familiar with basic operations in Pyhon.

Time

1 × 45 minutes

2 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)

   Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, »>.

2.1 Numbers

Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses can be used for grouping. Comments in Python start with the hash character: #. For example:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
... 7/3
2
>>> 7/-3
-3
```

The equal sign ('=') is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

A value can be assigned to several variables simultaneously:

```
>>> x = y = z = 0  # Zero x, y and z
```

There is build in the conversion function to floating point and integer numbers.

```
>>> a=5
>>> b=7.55
>>> float(a)
5.0
>>> int(b)
7
>>>
```

2.2 Strings

Python can manipulate strings in several ways, with single and double quotes:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Two string literals next to each other are automatically concatenated; the first line above could also have been written word = 'Help' 'A'; this only works with two literals, not with arbitrary string expressions:

```
>>> 'str' 'ing'                    # <- This is ok
'string'
>>> 'str'.strip() + 'ing'    # <- This is ok
'string'
>>> 'str'.strip() 'ing'      # <- This is invalid
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                      ^
SyntaxError: invalid syntax
```

Strings can be subscripted (indexed); like in C, the first character of a string has subscript (index) 0. There is no separate character type; a character is simply a string of size one.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]     # The first two characters
'He'
>>> word[2:]     # Everything except the first two characters
'lpA
```

Indices may be negative numbers, to start counting from the right. For example:

```
>>> word[-1]      # The last character
'A'
>>> word[-2]      # The last-but-one character
'p'
>>> word[-2:]     # The last two characters
'pA'
>>> word[:-2]     # Everything except the last two characters
'Hel'
```

2.3 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list a:

```
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
```

```
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

The built-in function len() also applies to lists:

```
>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

2.4 First 'program'

We can write an initial sub-sequence of the Fibonacci series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

A trailing comma avoids the newline after the output:

```
>>> a, b = 0, 1
```

```
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

Introduction to Image Processing in Python

Author: Marek Kociński

November 2010

1 Purpose

The goal of this exercise is to get familiar with the iterative pixel access in the $2D$ image. What is more student ought to preserve ability to deal with raster images in computer systems.

Useful links:

1. Enthought Python Distribution

2. Python Imaging Library (PIL)

3. Matplotlib Library

Time

2×45 minutes

2 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)
2. Create new script named: *exercise_0_A.py*
3. Import needed modules

```
from PIL import Image
from pylab import *
```

4. Define width and height of the image

```
maxX = 200
maxY = 100
```

5. Create the new image

```
im = Image.new("L",[maxX, maxY])
```

6. Display image

```
figure(1)              # open the new window
ax = axes()          # set axis parameters
ax.set_axis_off()
#display image
```

```
imshow(im, cm.gray,interpolation = 'nearest', origin = 'lower' )
print im.format, im.size, im.mode # print basic image parameters
pix = im.load() # load image to the memory, with acces to each pixel
```

7. Print pixel values for first 10 columns and rows

```
print
for y in range(10):
    for x in range(10):
        print pix[x,y],
    print
```

8. Draw horizontal line on the image

```
linePosition = 5;

for y in range(maxY):
    for x in range(maxX):
        if y == linePosition:
            pix[x,y] = 255
```

9. Display modified image and print pixel values for first 10 columns and rows

```
figure(2)
ax = axes()
ax.set_axis_off()
imshow(im, cm.gray,interpolation = 'nearest', origin = 'lower' )

print
for y in range(10):
    for x in range(10):
        print pix[x,y],
    print
```

10. Show all created imagess

```
show()
```

11. To do:

- draw line at the edge of the image
- draw vertical line
- draw skew line
- draw frame in distance 5 pixels from the edge
- is it needed to look for the whole image in order to draw horizontal line?

- modify script to use **one** *for* loop

The result of above operations is presented in the figure 1.

12. Create new empty script named *exercise_0_B.py* and load needed modules:

```
from PIL import Image
from pylab import *
```

13. Open grayscale image from the file, convert to grayscale and print its parameters

```
im = Image.open("brain.bmp").convert(''L'') # open the file
print im.format, im.size, im.mode # print basic parameters
```

14. Get the image width and height and display image

```
maxX, maxY = im.size
print 'maxX = ', maxX, 'maxY = ', maxY

figure(1)
ax = axes()
ax.set_axis_off()
imshow(im, cm.gray, interpolation = 'nearest', origin = 'lower' )
```

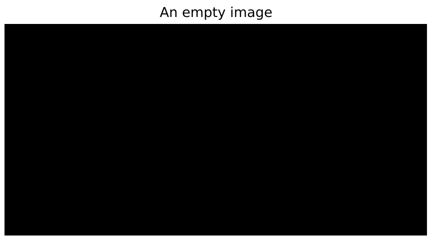
15. Display the image

```
f1 = figure(1)
f1.suptitle('Selected slice from MRI examination',size=18,color='red')
s1 = subplot(131)
imshow(im, cm.gray, interpolation = 'nearest', origin = 'image' )
title('Veins in the human brain')
s1.set_axis_off ()
```

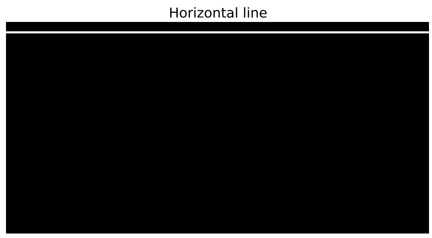
16. Print piksel values for square in the middle of the image

```
pix = im.load()

for y in range(maxY/2 - 5,maxY/2 + 5):
    for x in range(maxX/2 - 5,maxX/2 + 5):
      print '%3d' %pix[x,y],

    print
```

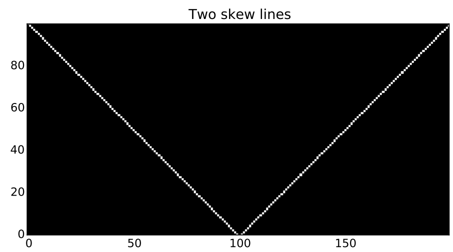
17. To do:
Change pixel values in the middle square of the image to black, white or gray. Display the original image using different colormaps (pseudo colors) e.g. cm.jet, cm.hot, . . . (fig. 2)



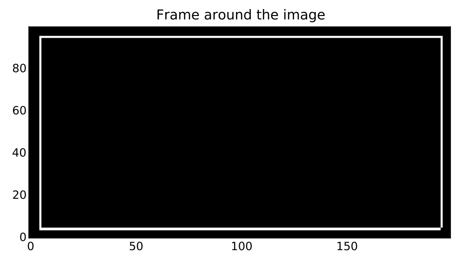
empty image



horizontal line



two skew lines



frame around borders in distance equal to 5 pixels

Figure 1: Operations on pixels

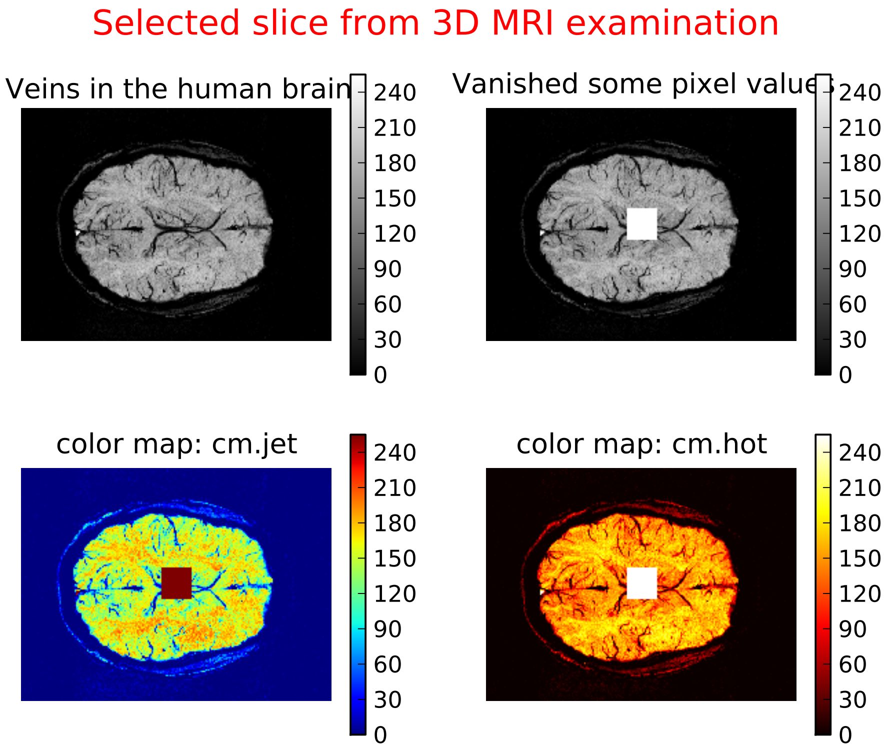


Figure 2: Image grayscale modifications and pseudocolors

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

Python Imaging Library 1

Author: Marek Kociński

March 2010

1 Purpose

To get acquainted with Python Imaging Library (PIL) and Python itself: loading images from file, saving images to file, basic operations and filtration.

Time

4×45 minutes

2 Introduction

The Python Imaging Library (PIL) adds image processing capabilities to Python interpreter. This library supports many file formats, and provides powerful image processing and graphics capabilities. The most important class in the PIL is the Image class. This class is defined in the module in the same name.

3 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)
2. It is convenient to create separate scripts for each processing task. Open new Editor Window (File → New Window) and write your code into it.
3. Import Image module: *import Image*
4. Load image *lenna.bmp*, display it and print basic information about it:

```
im = Image.open("lenna.bmp")
print im.format, im.size, im.mode
im.show()
```

5. Use *convert()* method to convert image representation from RGB to 8-bits gary-level im_L (use *"L"* option as a method parameter). Find out about different image modes. Save result image into file

```
im_L.save("lenna-gray-level.png","PNG")
```

6. Crop the region from the gray-level image and paste it into RGB image. Use *crop(), paste()* and *convert()* methods. Hint: define (left, upper, right, lower) cooridinates of your region: box=(100,100,400,400). It is possible to display region itself too. The results is presented in fig. 1.

```
box = (100,100,400,400)
region = im.crop(box)
region.show()
```

7. Blendig between two images or regions is possible. Method *blend()* creates a new image by interpolating bewtween two images or regions, using a constant alpha. Both images must have the same size and mode. Create second region box2=(200,200,500,500) and blend it with box. (Fig. 2)

8. Geometrical operations are avaliable usign finctions: *transpose(), rotate()* . Read documentation of this functions, pay attention to possible input parameters. Transform loaded image into form presented in figure 3. Dispaly **FULL** rotated image.

9. Load “flowers.bmp” image. Spit it into tree separete components R,G,B and change the band order to B,G,R (Fig. 4).

```
img = Image.open(''flowers.bmp'')
r,g,b = img.split()
img1 = Image.merge(''RGB'', (b,g,r))
img1.show()
```

10. Create new image (*Image.new()*) with the same size as *flowers.bmp*. Resize it (use one of the avaliable interpolation methods) and divide into R,G,B bands. Copy each band into newly created image to achieve results presented in the figure 5. Check the difference among NEAREST, BILINEAR and BICUBIC interpolation methods.

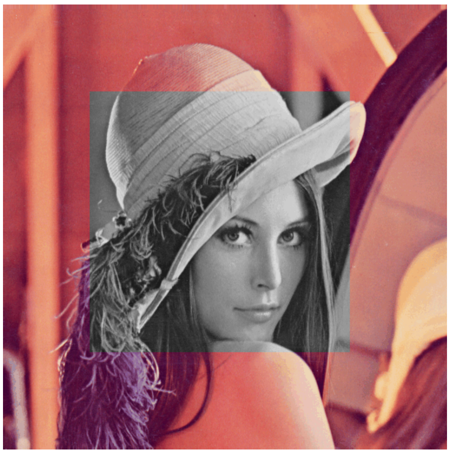


Figure 1: Crop—paste result.

Figure 2: Blending two regions with $\alpha = 0.5$.

Figure 3: Transformations: flipping and rotation ot the image.

RGB

BGR

Figure 4: Image *Flowers* presented with different color components order

(a) Each band presented as gray-level image

(b) Each band presented as RGB image

Figure 5: R,G,B bands of *Flowers.bmp* presented as separate images

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

Python Imaging Library 2

Author: Marek Kociński

March 2010

1 Purpose

To get acquainted with Image filtering using build-in methods. The concept of the thresholding of the gray-level and RBG images will be introduced.

Time

3×45 minutes

2 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)
2. Open new Editor Window (File → New Window) and write your code into it.
3. Import needed modules, e.g. *Image*
4. Open "goldhill.bmp" image and convert it to the 8 bit grayscale. The brihteness and contrast can be manipulated by chaning value of the each pixel within the image, for example with the *point()* function. In this operation an anonymous function (that is not bound to a name) using a construct called "lambda" is used. Task: multiply each pixel by 1.2 (Fig. 1).

```
out = im.point(lambda i: i*1.5)
```

5. The concept of the theresholding is as followes: for the gray-level images pixels with value bigger than threshold value are set to 255, whereas pixels with values below the thresh are set to 0. In RGB images each band is thresholded separately (Fig. 2). Introduce yourself to thresholding function listed below and perform thresholding for different thresh values for both types of images. Set separate values for R,G,B bands. What changes are made in output image?

(a) Original image

(b) Each pixel multiplied by 1.5

(c) Each pixel multiplied by 0.3

Figure 1: Operations on the pixel values

(a) Gray-level image, th = 125   *(b) RGB image, th = (125, 125, 125)*

Figure 2: Image thresholding

```
def thresholding(im,thL=125,thRGB=(125,125,125),sh=0):
    """Thresholding function for "L" and "RGB" images.
    thL - threshold value
    thRGB - tuple of threshold values: (thR, thG, thB)"""

    fn="thresholding"
    if(sh): print "Function %s" %fn

    if (im.mode =="L"):
        if(sh): print "***8 pixels, gray scale level image"
        return im.point(lambda i: i>thL and 255)
    if(im.mode =="RGB"):
        if(sh) : print "***RBB image"
        rgb = im.split()
        r = rgb[0].point(lambda i: i>thRGB[0] and 255)
        g = rgb[1].point(lambda i: i>thRGB[1] and 255)
        b = rgb[2].point(lambda i: i>thRGB[2] and 255)
        return Image.merge(im.mode, (r,g,b) )
```

6. Perform various filtration types of the *goldhill.bmp* image using methods from new imported modules (Fig. 4). Use different values for each function. Note the results for values bigger than 1 and smaller than 1. From the functions listed below use at least 5 different filter types.

   - ImageEnhance Module
     - Sharpness()
     - Brightness()

  - Contrast()
  - Color()
- ImageFilter Module
  - MinFilter()
  - MedianFilter()
  - MaxFilter()
  - BLUR
  - CONTOUR
  - DETAIL
  - EDGE_ENHANCE
  - FIND_EDGES
  - SMOOTH
  - SHARPEN
  - Kernel() for laplace kernel= $(1, -2, 1, -2, 5, -2, 1, -2, 1)$

(a) *Sharpness* = 3.5 (b) *Sharpness* = 0.2 (c) *Brightness* = 2.0 (d) *Brightness* = 0.3

(e) *Contrast* = 1.3 (f) *Contrast* = 0.3 (g) *Color* = 2.0 (h) *Color* = 0.3

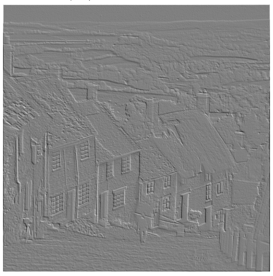
Figure 3: Image Enhance module modifications


(a) *Blurring*


(b) *Contour*


(c) *Detail*


(d) *Edge enhance*


(e) *Embos*


(f) *Find edges*


(g) *Smooth*


(h) *Sharpen*


(i) *Max, kernel size:* 5


(j) *Min, kernel size:* 5


(k) *Median, kernel size:* 5


(l) *Laplace kernel*

Figure 4: Image Filter module functions

7. To detect edes in the image the appropriate procedure should be applied (Fig. 5). The lines or edges in each direction are found separately with use of dedicated mask. Use function *Kernel()* to do filtration of the image with several different masks — apply at least 6.

   - Line detection masks (Fig. 6):

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

(a) *Horizontal* (b) *Vertical* (c) *+45°* (d) *-45°*

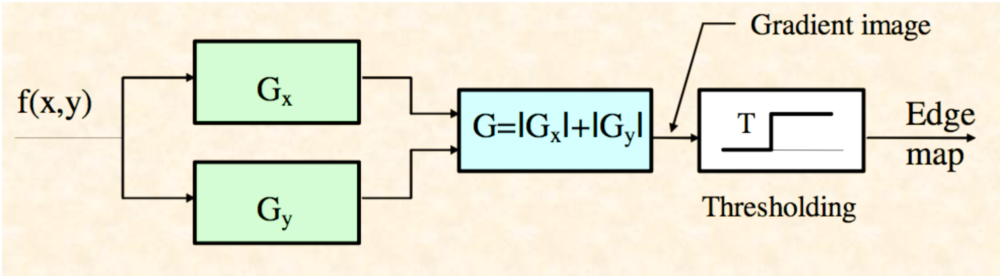
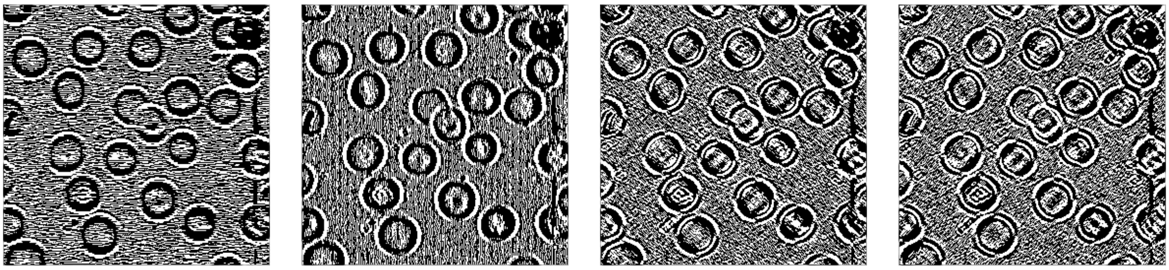


Figure 5: *Edge detection procedure (from* Image Processing *lectures by P.Strumiłło and M.Strzelecki)*



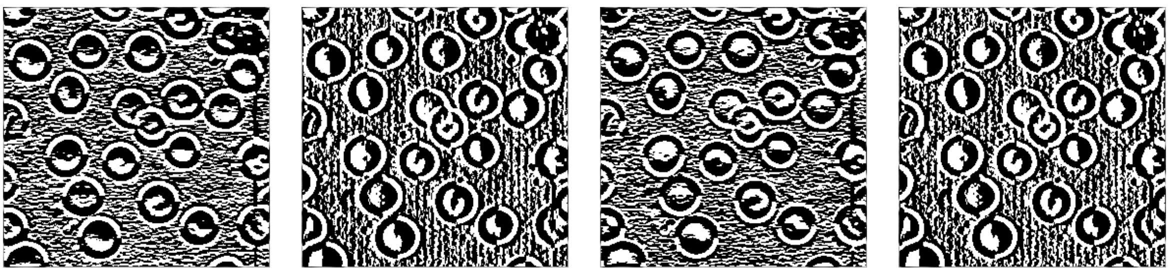
(a) Horizontal *(b) Vertical* *(c) +45°* *(d) −45°*

Figure 6: Line detection

- Edge detection masks (gradient operators) (Fig. 7):

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

(a) Prewitt (horiz.) *(b) Prewitt (vertic.)* *(c) Sobel (horiz.)* *(d) Sobel (vertic.)*

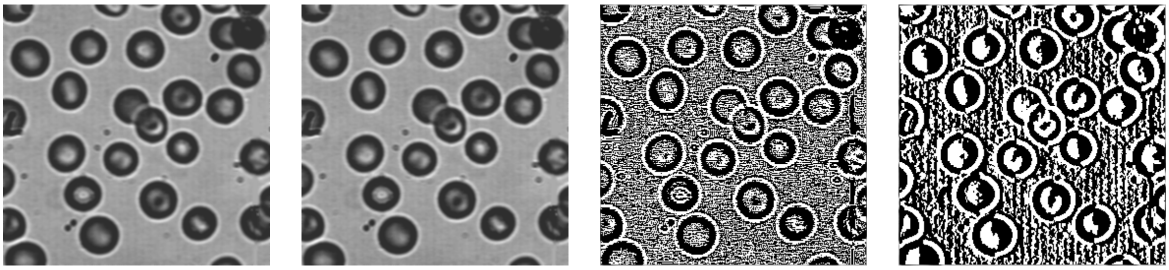


(a) Prewitt (horiz.) *(b) Prewitt (vert.)* *(c) Sobel (horiz.)* *(d) Sobel (horiz.)*

Figure 7: Prewitt and Sobet filters line detection

- Gauss and Laplace masks (Fig. 8).

$$\underset{(a)\ Gauss}{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}} \quad \underset{(b)\ Gauss}{\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}} \quad \underset{(c)\ Laplace\ 1}{\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}} \quad \underset{(d)\ Laplace\ 2}{\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}}$$



(a) Gauss 1 *(b) Gauss* 2 *(c) Laplace* 1 *(d) Laplace* 2

Figure 8: Gauss and Laplace filters

- Corners detection (high-pass filetring) (Fig. 9)

$$\begin{matrix} \begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} & \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} & & \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix} & \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} & \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} \end{matrix}$$

8. Apply the edge detection procedure for image *ksztalty.bmp.* Use *Laplace 1* kernel and procedure presented in the figure 5.

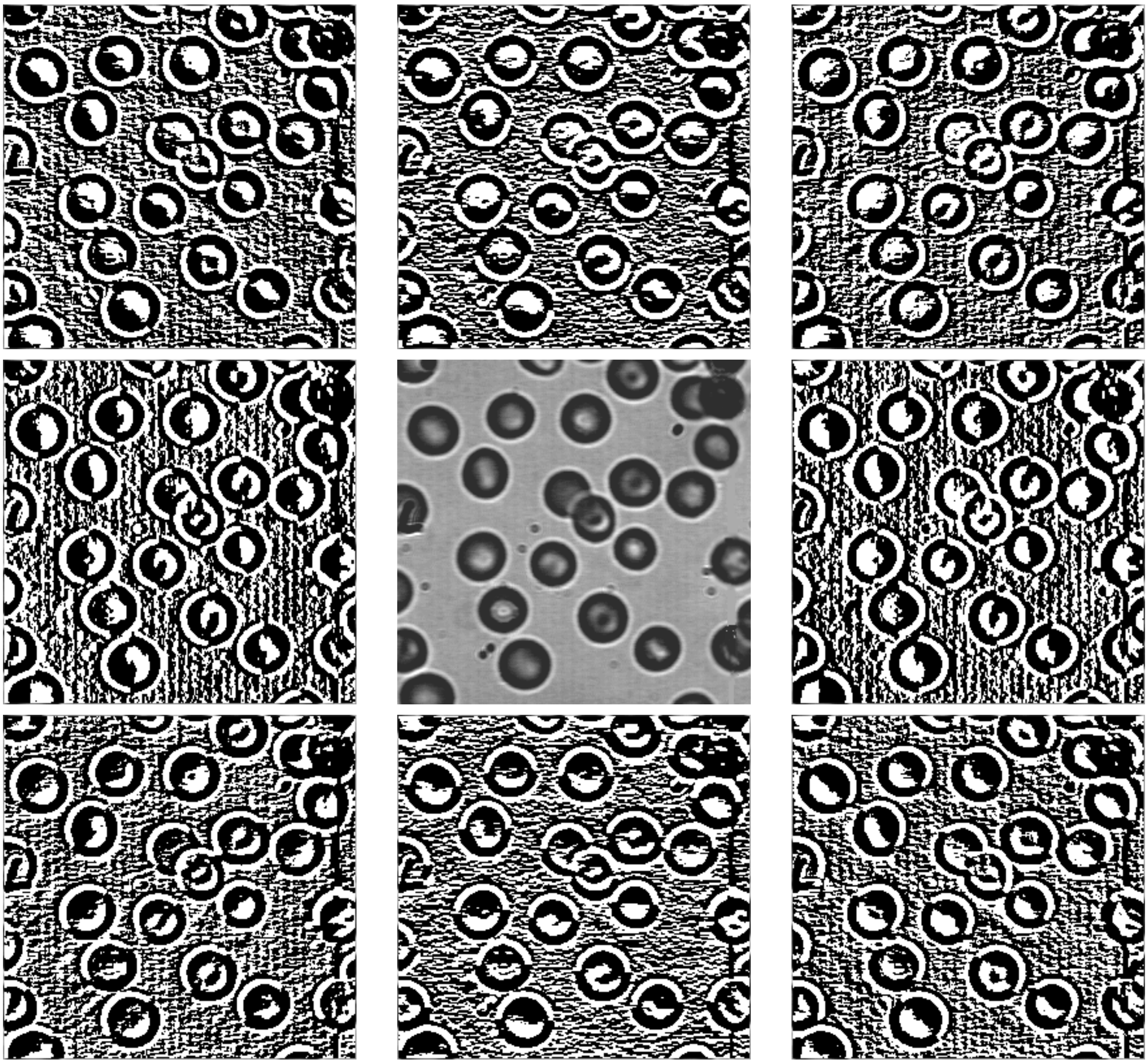
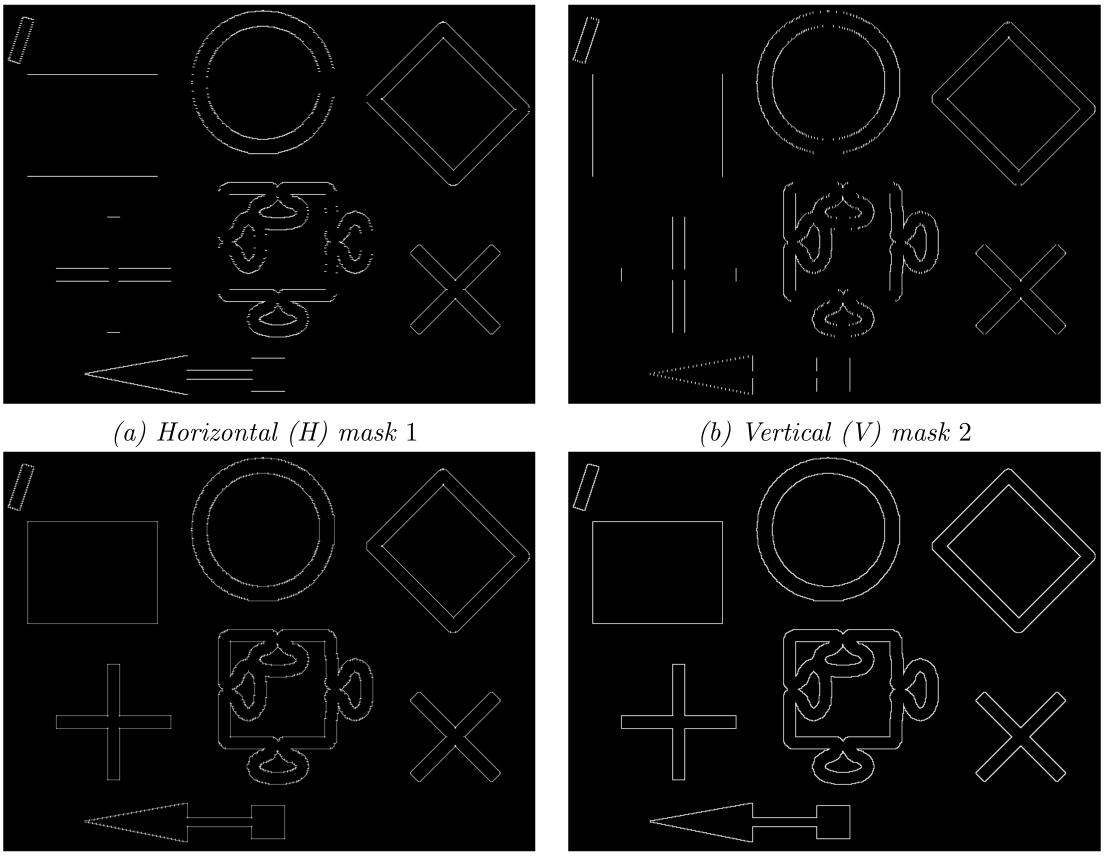


Figure 9: Corners detection filters



(a) *Horizontal (H) mask* 1

(b) *Vertical (V) mask* 2

(c) *Sum of |H| and |V| images*

(d) *Laplace* 1

Figure 10: Edge detections with different masks

Technical University of Lodz

Medical Electronics Division
Technical University of Lodz, Institute of Electronics

INSTITUTE of ELECTRONICS

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

Python Imaging Library 3

Author: Marek Kociński

March 2010

1 Purpose

To get acquainted with Image filtering using direct access to the pixels. The basic morphological operations will be applied to binary images. $3D$ image will be load into memory and selected slice will be displayed as $2D$ image.

Time

3×45 minutes

2 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)
2. Open new Editor Window (File → New Window) and write your code into it.
3. Import needed modules, e.g. *Image*
4. Very often it is needed to get direct access to image data. Function *load()* returns a pixel access object that can be used to read and modify pixels. The access object behaves like a 2-dimensional array, so you can do:

```
pix = im.load()
print pix[x,y]
pix[x,y] = value
```

   Create inversion of the brighteness level of the *goldhill.bmp* image 1.

```
im = Image.open("goldhill.bmp")
im = im.convert(``L``)

sx = im.size[0]
sy = im.size[1]

pix = im.load()
for y in range(sy):
        for x in range(sx):
                pix[x,y] = 255 - pix[x,y]

im.save('neg.bmp')
```

5. Create an function that performs image convolution with 3×3 masks. (Hint: Designing and Implementing Linear Filters in the Spatial Domain).

(a) Gray-level image *(b) Inverted gray-level image*

Figure 1: Pixel operations

```
def convolution2d(img,k, size=3):
    """This function make image convolutin with cernel.

    Kernel as a list from 0 to 8."""
    fn="convolution2d"
    print "Function %s" %fn
    pix = img.load()

    if size == 3:
        print "***Kernel size = 3"
        sx = img.size[0]
        sy = img.size[1]

        nimg = Image.new (img.mode, img.size)
        npix = nimg.load()
        val=[]
        for y in range(1,sy-1):
            for x in range(1,sx-1):
                npix[x,y]= (k[8]*pix[x-1,y-1] + k[7]*pix[x,y-1]
                          + k[6]*pix[x+1,y-1] + k[5]*pix[x-1,y]
                          + k[4]*pix[x,y] + k[3]*pix[x+1,y]
                          + k[2]*pix[x-1,y+1] + k[1]*pix[x,y+1]
                          + k[0]*pix[x+1,y+1])
      return nimg
```

Load image *blood1.bmp* and perform convolution with Prewitt masks:

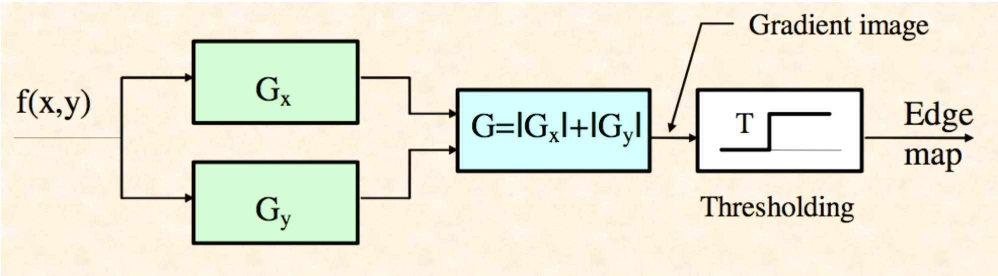


Figure 2: *Edge detection procedure (from* Image Processing *lectures by P.Strumiłło and M.Strzelecki)*

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

(a) Prewitt (horiz.) — k1

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

(b) Prewitt (vertic.) — k3

Pass convolution cernels as the function argument in the following convetnion:

$$k1 = [-1, -1, -1, 0, 0, 0, 1, 1, 1]$$

$$k3 = [-1, 0, 1, -1, 0, 1, -1, 0, 1]$$

Find edges of the image using procedure showed in the Fig. 2. Find appropriate threshold value (Fig. 3).

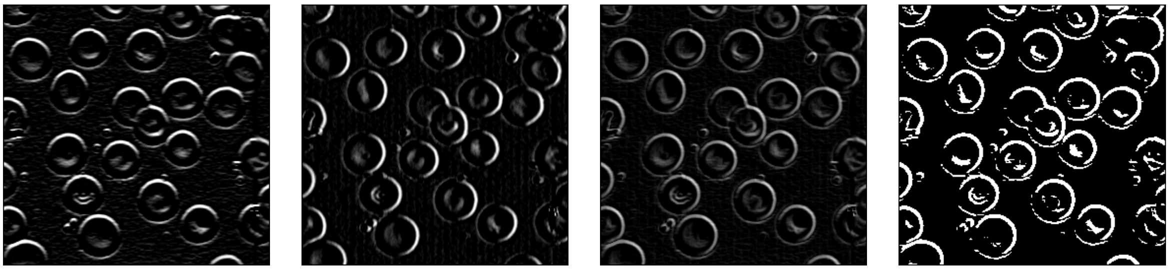
Add booth resulting images using pixel operations.

```
a1 = Image.new(im.mode, im.size)

pix_a1_v = a1_v.load()
pix_a1_h = a1_h.load()
pix_a1 = a1.load()

for y in range(sy):
    for x in range(sx):
        pix_a1[x,y] = (pix_a1_h[x,y] + pix_a1_v[x,y] )/2
```

6. Load images *bin1.bmp* and *bin2.bmp*. For both images apply function dilation *dilaet2D()* and erosion *erode2D()*. Use different structuring element (Fig. 4). Define each element as tupe or list, for example first element, which is *cross* shapled is defined as followes:

$$se1 = (0, 1, 0, 1, 1, 1, 0, 1, 0,)$$

Use at least 6 different structuring elements (Fig. 5).



(a) Prewitt (vert.) (b) Prewitt (horiz.) (c) Sum of gradients (d) After thresholding

Figure 3: Edge detection using Prewitt mask

```
def dilate2D(im, se, sh=1):
    """dilate2D function is to do dilation of img image (2D).
    se - tuple describes any FULL SIZE se.
        - se(n) - ON piksels

    Dilation for WHITE objects with BLACK background!!!   """

    fn="dilate2D(im,se)"
    if(sh): print "Function %s" %fn

    sX,sY = im.size
    imD = im.load()
    nim = Image.new (im.mode, im.size)
    nimD= nim.load()

    seX = 3 #size of se
    seY = 3
    hx = seX/2
    hy = seY/2

    for y in range(hy,sY-hy):
        for x in range(hx,sX-hx):
            if( imD[x,y] == 255 ):
                if(se[0]): nimD[x-1,y-1] =255
                if(se[1]): nimD[x,  y-1] =255
                if(se[2]): nimD[x+1,y-1] =255
                if(se[3]): nimD[x-1,y]   =255
                if(se[4]): nimD[x,  y]   =255
                if(se[5]): nimD[x+1,y]   =255
                if(se[6]): nimD[x-1,y+1] =255
                if(se[7]): nimD[x,  y+1] =255
                if(se[8]): nimD[x+1,y+1] =255
```

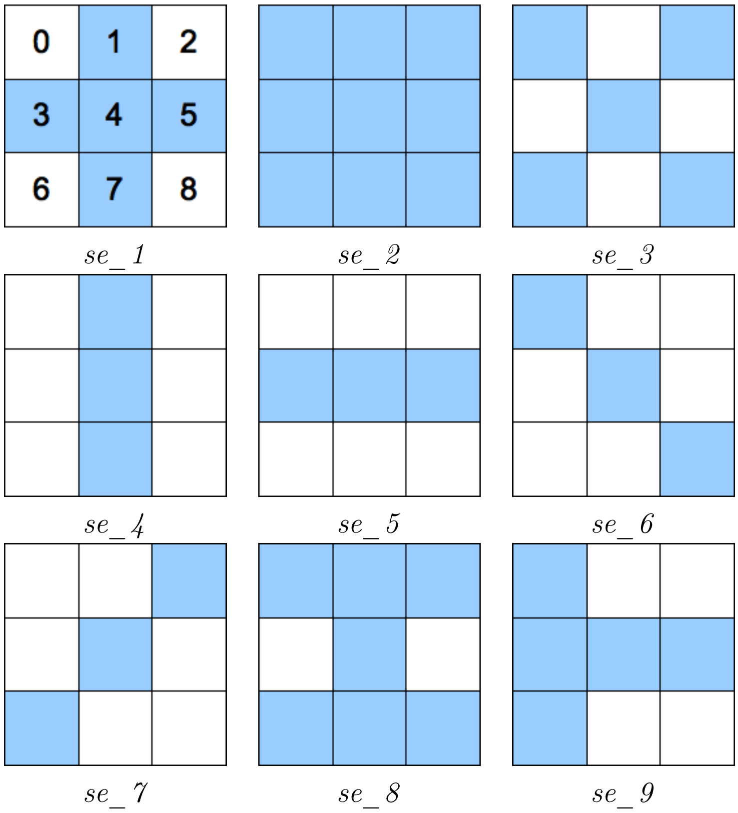


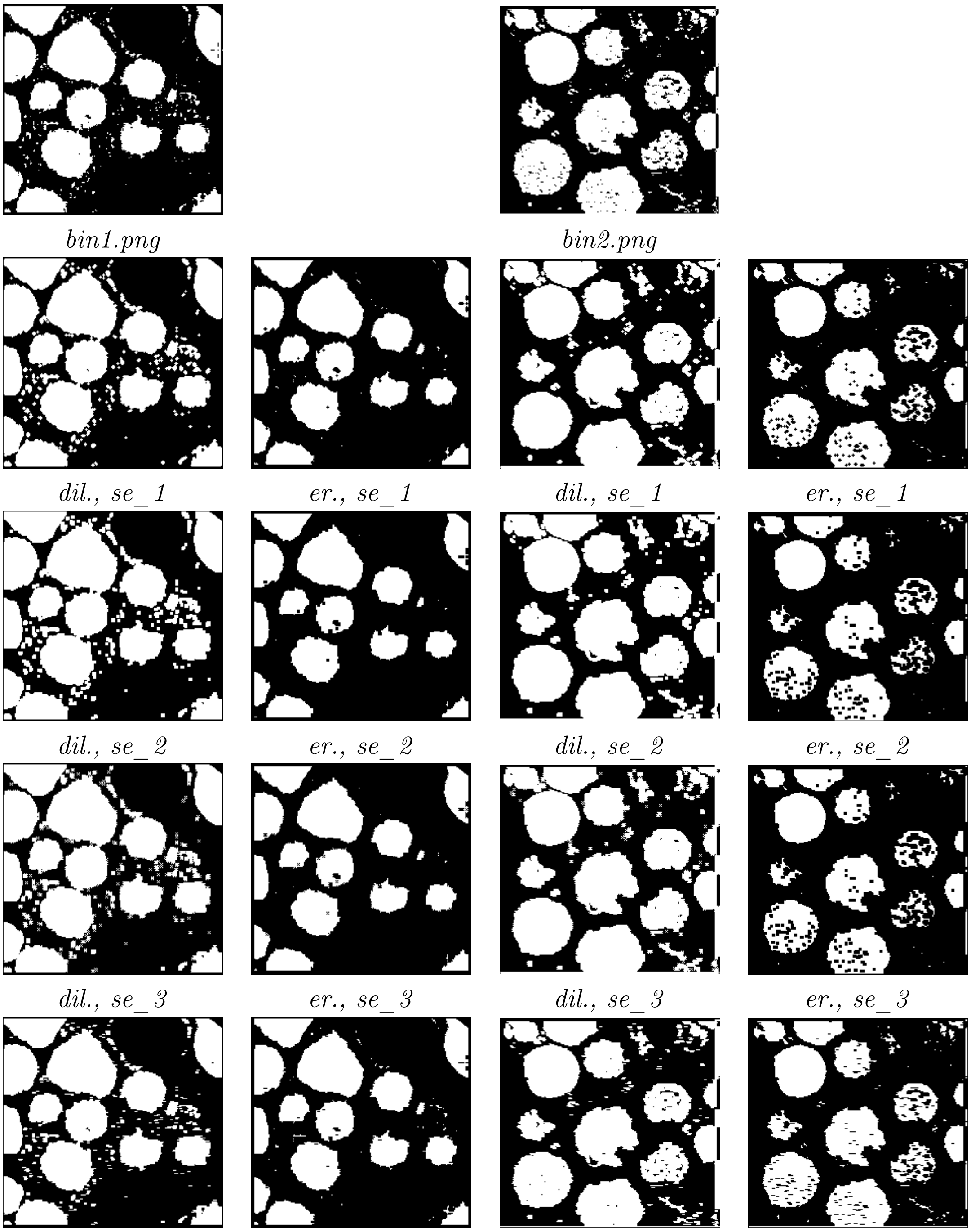
Figure 4: Various structuring elements for morphological operations

```
    if(sh): print "***dilation done..."
    return nim

def erode2D(im,se,sh=1):
    """erode2D function is to make erosion od img image (2D).
    se - tuple describes any FULL SIZE se.
        - se(n) - ON piksels

    EROSION for WHITE objects with BLACK background!!!"""

    fn="erode2D(im,se)"
    if(sh): print "Function %s" %fn

    sX,sY = im.size
    imD = im.load()
    nim = Image.new (im.mode, im.size)
    nimD= nim.load()
```

```
seX = 3 #size of se
seY = 3
hx = seX/2
hy = seY/2

for y in range(hy,sY-hy):
    for x in range(hx,sX-hx):
        x0=[]
        if( imD[x,y] == 255 ):
            if se[0]:x0.append(imD[x-1,y-1])
            if se[1]:x0.append(imD[x,  y-1])
            if se[2]:x0.append(imD[x+1,y-1])
            if se[3]:x0.append(imD[x-1,y])
            if se[4]:x0.append(imD[x,  y])
            if se[5]:x0.append(imD[x+1,y])
            if se[6]:x0.append(imD[x-1,y+1])
            if se[7]:x0.append(imD[x,  y+1])
            if se[8]:x0.append(imD[x+1,y+1])
            if all(x0) : nimD[x,y] = 255

if(sh): print "***erode done..."
return nim
```

7. Define new functions for morphological opening and for morphological closing of $2D$ images (Fig. 6):

   - *MorphologicalOpen2D(....)* which is combination of erosion and dilation of the image
   - *MorphologicalClose2D(...)* which is combination of dilation and erosion of the image
   - *Morphological gradient(...)* which is difference between dilation and erosion



bin1.png *bin2.png*

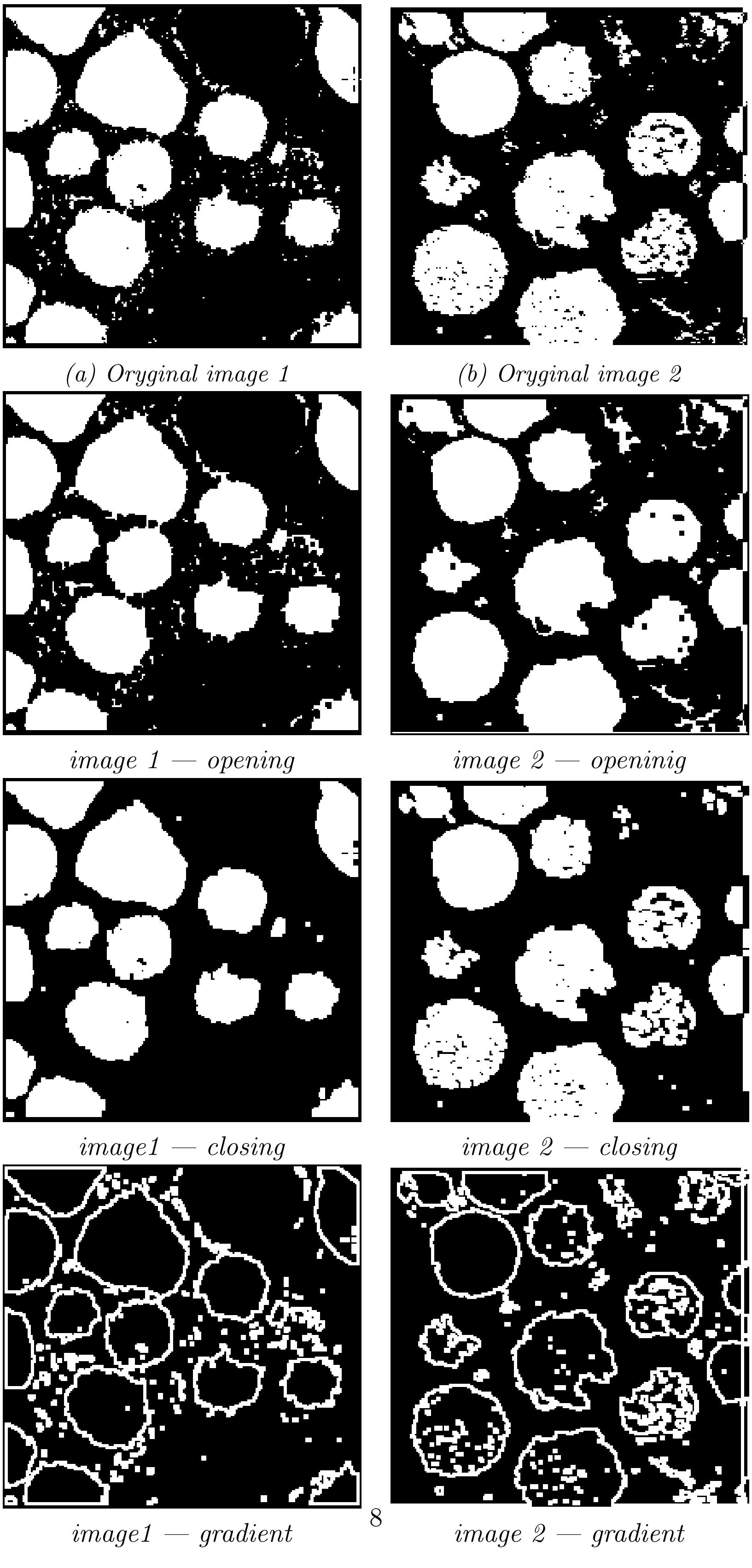
dil., se_1 *er., se_1* *dil., se_1* *er., se_1*

dil., se_2 *er., se_2* *dil., se_2* *er., se_2*

dil., se_3 *er., se_3* *dil., se_3* *er., se_3*

dil., se_5 *er., se_5* *dil., se_5* *er., se_5*

Figure 5: Results of morphological operations of image filtering with various structuring elements



(a) Oryginal image 1 *(b) Oryginal image 2*

image 1 — opening *image 2 — openinig*

image1 — closing *image 2 — closing*

image1 — gradient *image 2 — gradient*

Figure 6: Morphological operations

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

3*D* images

Author: Marek Kociński

March 2010

1 Purpose

To get acquainted with basic manipulation on $3D$ raw image data. The matplotlib module will be used to create publication quality figures.

Time

3×45 minutes

2 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)
2. Open new Editor Window (File → New Window) and write your code into it.
3. Import needed modules, e.g. *Image, array.* Use construction:

   ```
   import scipy as sc
   from pylab import *
   import array
   import Image
   ```

4. Read data from the flie. Each voxel is 8-bit unsigned integr. The size of the image ($3D$ matrix) is $256 \times 256 \times 256$ voxels.

   ```
   tmpfile = "qinp01_3000_036_3_256.raw"

   fileobj = open(tmpfile, mode='rb')
   binvalues = array.array('B')
   binvalues.read(fileobj, 256*256*256)
   ```

5. Convert loaded data to SciPy array structure:

   ```
   data = sc.array(binvalues, dtype=sc.uint8)
   data = sc.reshape(data, (256,256,256))
   fileobj.close()
   ```

6. Print basic information abou array

   ```
   print data.size
   print data.shape
   ```

7. It is possoble to *connect* SciPy array structure with Image object of the PIL module. Select 127 slice in each direction 1, 2, 3, convert to Image structure and show it.

(a) dir. 1 *(b) dir* 2 *(c) dir* 3

Figure 1: Selected slice from $3D$ image

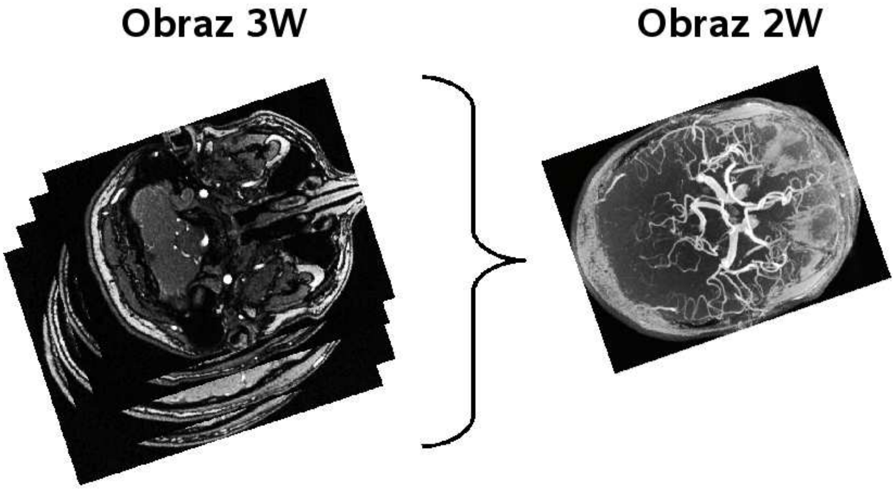


Figure 2: Techinique of MIP creation

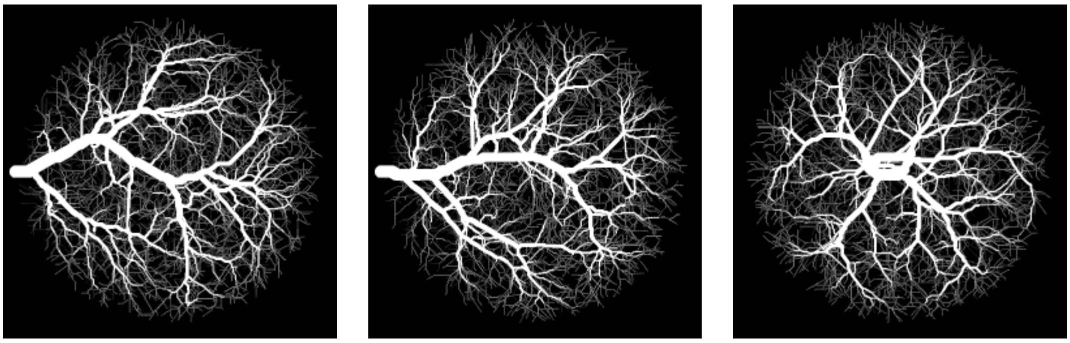
```
im1 = Image.fromarray (data[:,:,127])
im2 = Image.fromarray (data[:,127,:])
im3 = Image.fromarray (data[127,:,:])
```

8. Print maximum, minimum and mean of the data:

```
print data.max()
print data.min()
print data.mean()
```

9. Create MIP (Maximum Intensity Projection) of the image in each direction. The idea of MIP is presented in the Figure 2.

```
mip1 = Image.fromarray (data.max(0))
mip2 = Image.fromarray (data.max(1))
```



(a) dir. 1 *(b) dir* 2 *(c) dir* 3

Figure 3: MIP images of 3*D* data

```
mip3 = Image.fromarray (data.max(2))
```

10. Present 4 images on one figure. It is possible to write this figue in different file formats, like: pdf, eps, png, ps, emf, raw or even vecor graphics svg.

```
fig = figure()

subplot(221)
imshow(im2,cmap=cm.gray)
colorbar()
subplot(222)
imshow(im1)
subplot(223)
imshow(mip2,cmap=cm.gray)
subplot(224)
imshow(mip3)
colorbar()
show()
```

11. Create new Python script. Load 3*D* data as in the previous example. Create one MIP image:

```
im1 = Image.fromarray (data.max(0))
```

12. Invert 3*D* image and create mIP (minimum Intensity Projection)

```
im4 = Image.fromarray (d.min(0))
```

Hint 1: It is not good idea to use three for loops ;-). But if you decide so, the *Ctrl+z* key sequence may occur to be helpful...
Hint 2: Change data precision to *int16*, do invertion, and back to *uint8*.

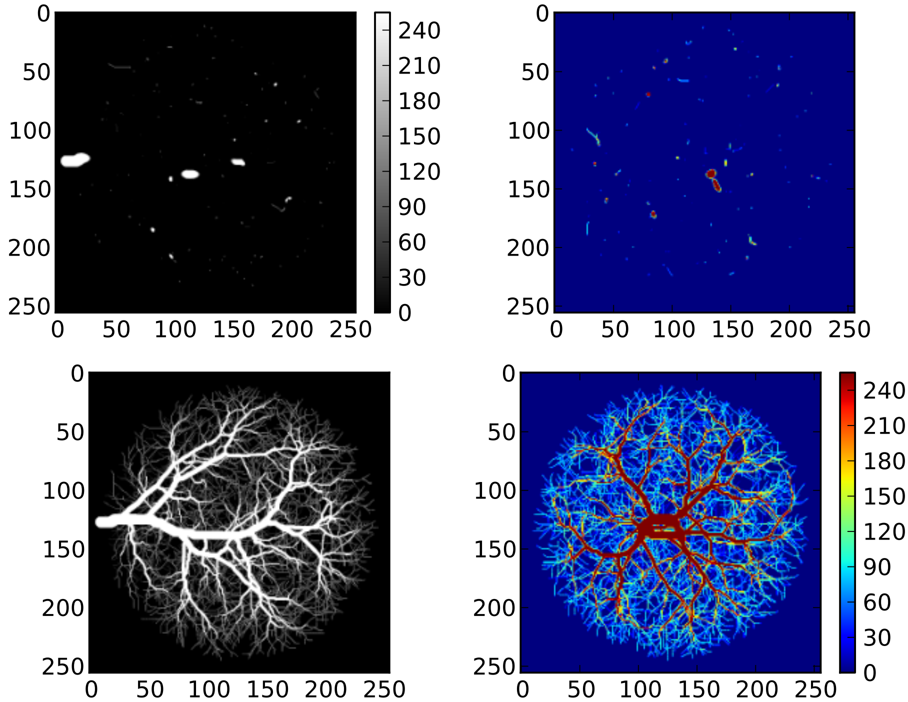
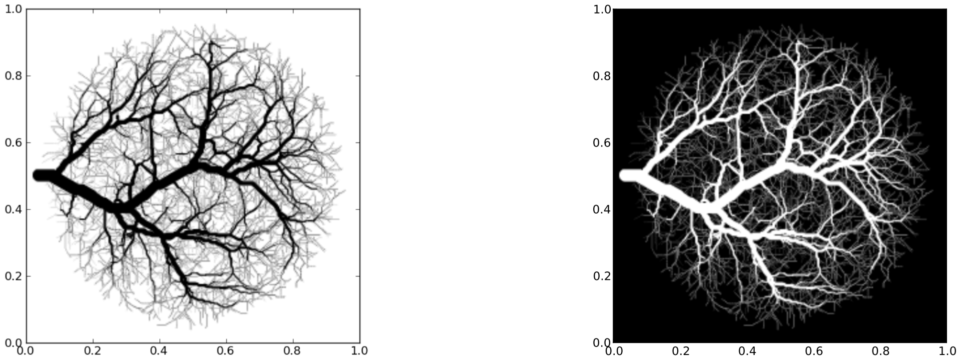


Figure 4: Four images on one figure

```
a = sc.int16 (data)
...
...here invert image...
...
d = sc.uint8 (c)
```

13. By pressing 't' letter toogle between two images showed on one fiugure.

```
print "Press t ... :-)"
extent = (0,1,0,1)
img1 = imshow(im1, extent=extent, cmap=cm.gray )
img2 = imshow(im4, extent=extent, hold=True, cmap=cm.gray )
img2.set_visible(False)

def toggle_images(event):
    'toggle the visible state of the two images'
    if event.key != 't': return
    b1 = img1.get_visible()
```



(a) mIP *(b)MIP*

Figure 5: Toogle between MIP and mIP imges

```
    b2 = img2.get_visible()
    img1.set_visible(not b1)
    img2.set_visible(not b2)
    draw()

connect('key_press_event', toggle_images)
show()
```

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

Graphical User Interface (GUI) with the use of wxPython Library

Author: Marek Kociński

April 2010

1 Purpose

To introduce yourself to creation of Graphical User Interface with the use of wxPython library (http://www.wxpython.org/).

Time

2×45 minutes

2 Materials and links

2.1 Documentation

1. www.wxPython.org
2. wxWidgets 2.8.10: A portable C++ and Python GUI toolkitl
3. new wxPyDocs
4. How to Learn wxPython

2.2 Tutorials

1. The wxPython Linux Tutorial
2. Zetcode wxPython tutorial
3. Getting started with wxPython

3 Tasks

1. Familiarize yourself with source codes of given examples: **ImageOperations** (Fig. 1), **ImageViewer** (Fig. 2), **WindowSizer** (Fig. 3).
2. Using *copy* and *paste* method try to run some of the examples from Tutorials web pages.

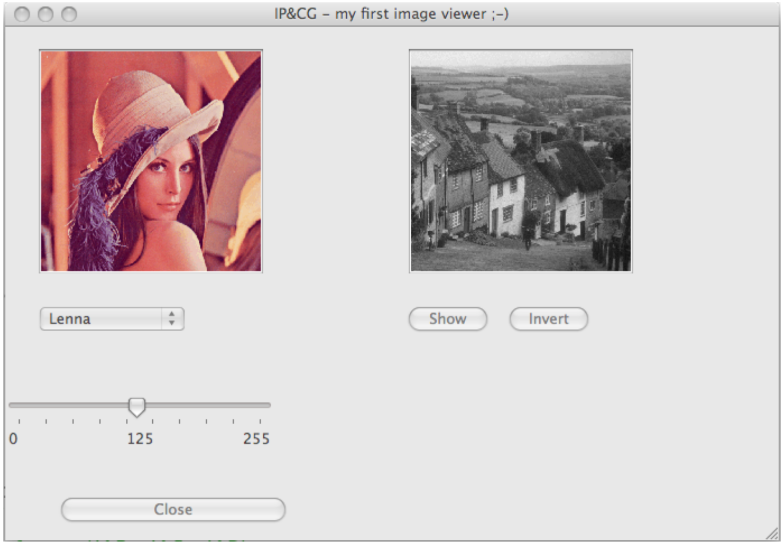


Figure 1: Simple Image operations

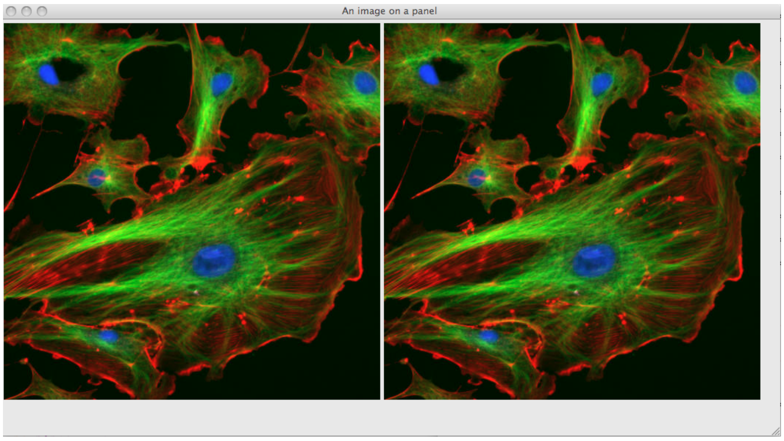


Figure 2: Simple Image Viewer (*found on the Internet*)

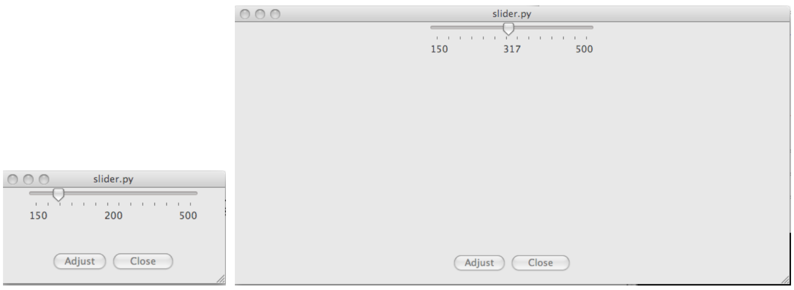


Figure 3: "Image Sizer" (*example from www.wxpython.org*)

Technical University of Lodz

Medical Electronics Division
Technical University of Lodz, Institute of Electronics

INSTITUTE of ELECTRONICS

Technical University of Łódź
Institute of Electronics
Medical Electronics Division

Image Processing and Computer Graphics

The Visualization Toolkit (VTK)

Author: Marek Kociński

April 2010

1 Purpose

To get acquainted with basic capabilities with the Visualization Toolkit (VTK). The VTK is an open-source, freely available software system for $3D$ computer graphics, image processing and visualization. VTK is cross-platform and runs on Linux, Windows, Mac and Unix platforms.

Time

4×45 minutes

2 The Graphics Model

There are seven basic objects that we use to render a scene. Documentation of all objects and classes used in vtk library is available on the webpage: http://www.vtk.org/doc/release/5.4/html/classes.html.

1. *vtkRenderWindow* — manages a window on the display device; one or more renderers draw into an instance of vtkRenderWindow.
2. *vtkRenderer* — coordinates the rendering process involving lights, cameras, and actors.
3. *vtkLight* — a source of light to illuminate the scene.
4. *vtkCamera* — defines the view position, focal point and other viewing properties of the scene.
5. *vtkActor* — represents an object rendered in the scene, including its properties (color, shading type, etc.) and position in the words coordinate system. (Note: vtkActor is a subclass of vktProp. vtkProp is a more general form of actor that includes annotation and 2D drawing classes.)
6. *vtkProperty* — defines the appearance properties of an actor including color, transparency, and lighting properties such as specular and diffuse. Also representational properties like wireframe and solid surface.
7. *vtkMapper* — the geometric representation for an actor. More than one actor may refer to the same mapper.

3 Tasks

1. Open Python interpreter window (Start→ Programy→ EPD32-6.0.2 → IDLE)
2. Open new Editor Window (File → New Window) and write your code into it.

3. Import needed modules, e.g. *vtk*.
4. Count distance between two points. In this example additional package *math* is needed.

```
import vtk
import math

p0 = (0,0,0)
p1 = (1,1,1)

distSquared = vtk.vtkMath.Distance2BetweenPoints(p0,p1)
dist = math.sqrt(distSquared)

print "p0␣=␣", p0
print "p1␣=␣", p1
print "distance␣squared␣=␣", distSquared
print "distance␣=␣", dist
```

5. Draw triangle on the black background (Fig. 1). Pay attention to used pipeline of the basic vtk objects in the graphic model.

```
import vtk

# create a rendering window and renderer
ren = vtk.vtkRenderer()
renWin = vtk.vtkRenderWindow()
renWin.AddRenderer(ren)

# create a renderwindowinteractor
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

# create points
points = vtk.vtkPoints()
points.InsertNextPoint(1.0,0.0,0.0)
points.InsertNextPoint(0.0,0.0,0.0)
points.InsertNextPoint(0.0,1.0,0.0)

triangle = vtk.vtkTriangle()
triangle.GetPointIds().SetId(0,0)
triangle.GetPointIds().SetId(1,1)
triangle.GetPointIds().SetId(2,2)

triangles = vtk.vtkCellArray()
```

```
triangles.InsertNextCell(triangle)

# polydata object
trianglePolyData = vtk.vtkPolyData()
trianglePolyData.SetPoints( points )
trianglePolyData.SetPolys( triangles )

# mapper
mapper = vtk.vtkPolyDataMapper()
mapper.SetInput(trianglePolyData)

# actor
actor = vtk.vtkActor()
actor.SetMapper(mapper)

# assign actor to the renderer
ren.AddActor(actor)

# enable user interface interactor
iren.Initialize()
renWin.Render()
iren.Start()
```

6. Draw a sphere, use *vtkSphereSource* class. Change some of the parameters: *PhiResolution*, *ThetaResolution*, *Radius* and *Position* of the sphere in the $3D$ space.

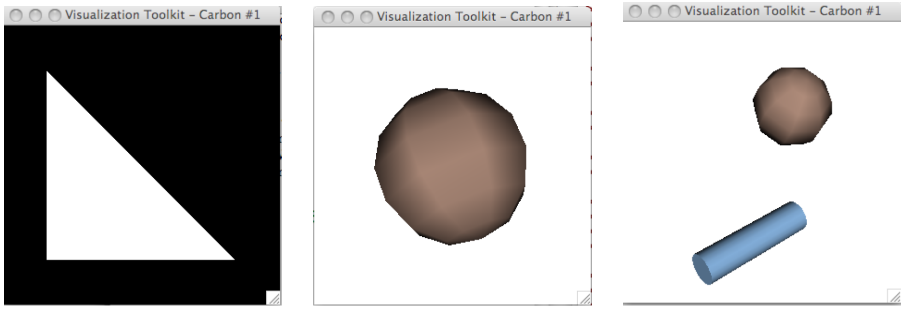
```
# create source
source = vtk.vtkSphereSource()
source.SetCenter(0,0,0)
source.SetRadius(5.0)
```

7. Change some of the surface properties of the sphere with the use of *GetProperty()* object:

    - *SetColor()* — RGB color in range (0.0–1.0)
    - *SetDiffuse()* — in range (0.0–1.0)
    - *SetSpecular()* — in range (0.0–1.0)
    - *SetSpecularPower()* — in range (0–255)

    and background color using *SetBackground(...)* method on the **renderer** object (Fig. 1).

8. With the use of *vtkCylinderSource* object draw cylinder. Use additional *vtkPolyDataMapper* and *vtkActor* for this purpose (Fig 1).



(a) Triangle *(b) Sphere* *(c) Sphere and cylinder*

Figure 1: Basic objects in the $3D$ space

9. It is possible to divide *RenderWindow* among few *Renderers* (see Fig. 2).
   (a) create 4 renderers (*vtkRenderer* class)
   (b) set different colors for each of them with the use of *SetBackground(...)* function
   (c) put every renderer in appropriate position inside RendererWindow
      - *ren1.SetViewport(0.0,0.0,0.5,0)*
      - *ren2.SetViewport(0.5,0.0,1.0,0.5)*
      - *ren3.SetViewport(0.0,0.5,0.5,1.0)*
      - *ren4.SetViewport(0.5,0.5,1.0,1.0)*
   (d) add each renderer to renderer window (use *AddRenderer(...)* function)
   (e) create 4 different $3D$ objects to render in every renderer:
      - Cone
         – use: *vtkConeSource, SetCenter(...),SetHeight(...), SetRadius(...), SetResolution(...), SetAngle(...)*
      - Cube
         – use: *vtkCubeSource, SetXLength(...), SetYLenght(...), SetZLength(...), SetCenter(...)*
      - Use other objects e.g.: *vtkArrowSource, vtkTextCource, vtkDiskSource, vtkEarthSource, vtkTexturedSphereSource, vtkPlaneSource,...*
   (f) for each $3D$ object create mapper and actor (*vtkPolyDataMapper, vtkActor*)
   (g) add actors to the rendreres (*AddActor(...)*)

10. *VTK* has implemented many components to image processing. To read and display $2D$ image it is enough to run code as follows (Fig. 3)

```
import vtk

reader = vtk.vtkBMPReader()
reader.SetFileName ("lenna.bmp")
```

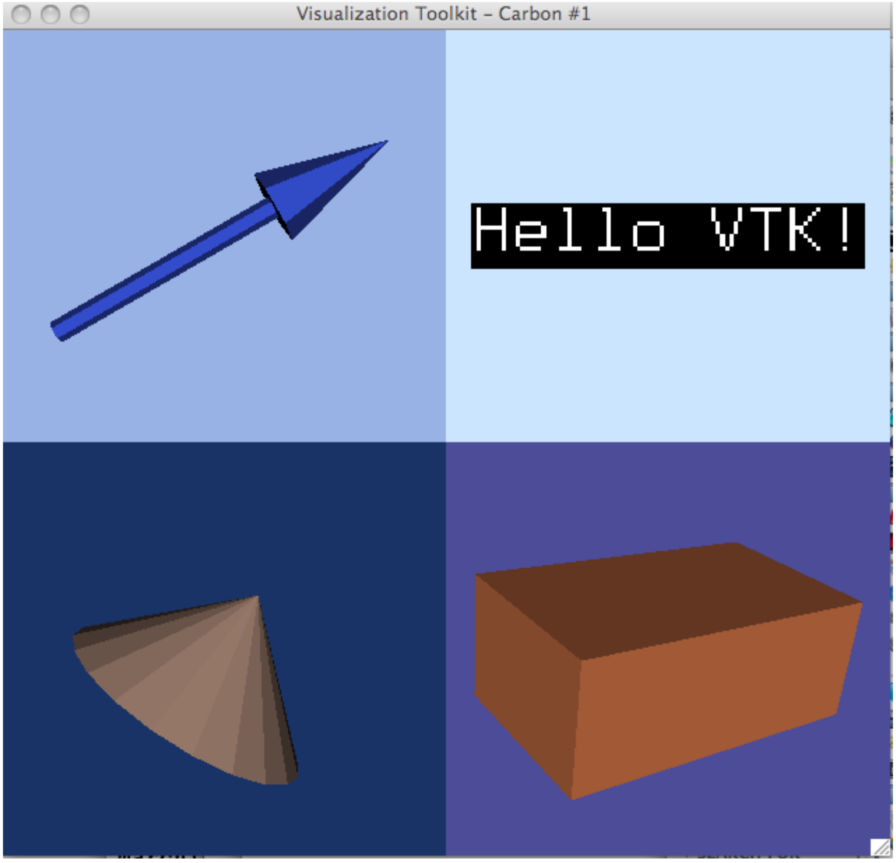


Figure 2: *Four renderers in one window*

```
iren = vtk.vtkRenderWindowInteractor ()

viewer = vtk.vtkImageViewer2 ()
viewer .SetupInteractor (iren)
viewer .SetInputConnection (reader.GetOutputPort ())
viewer .SetColorLevel (125)
viewer .SetColorWindow (255)
viewer .Render ()

iren .Start()
```

11. It is easy to warp image in the direction perpendicular to the image plane using the visualization filter *vtkWartScalr*. Set few values for *SetScaleFactor* (Fig. 4).

```
import vtk
```

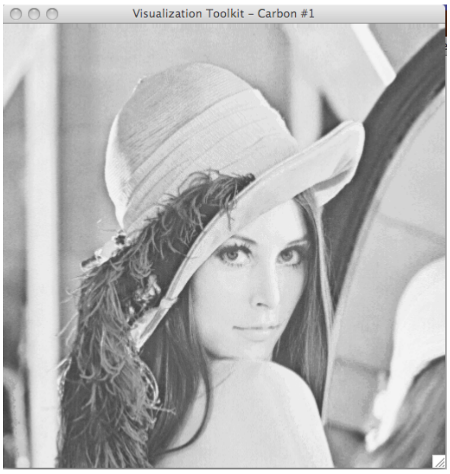


Figure 3: Imaged displayed with the use of *vtkImageViewer2*. Mouse buttons manipulation changes contrast and brightness of the image

```
reader = vtk.vtkBMPReader()
reader.SetFileName ("lenna.bmp")

imgGeometry = vtk.vtkImageDataGeometryFilter()
imgGeometry .SetInput (reader .GetOutput ())

warp = vtk.vtkWarpScalar ()
warp .SetInput (imgGeometry .GetOutput ())
warp .SetScaleFactor (0.7)

wl = vtk.vtkWindowLevelLookupTable ()

mapper = vtk.vtkPolyDataMapper ()
mapper .SetInputConnection (warp .GetOutputPort ())
mapper .SetScalarRange (0,2000)
mapper .ImmediateModeRenderingOff ()
mapper .SetLookupTable (wl)

imageActor = vtk.vtkImageActor ()
imageActor .SetInput (reader .GetOutput ())

warpActor = vtk.vtkActor ()
warpActor .SetMapper (mapper)

ren1 = vtk.vtkRenderer()
ren1 .SetBackground (0.2,0.2,0.4)
```

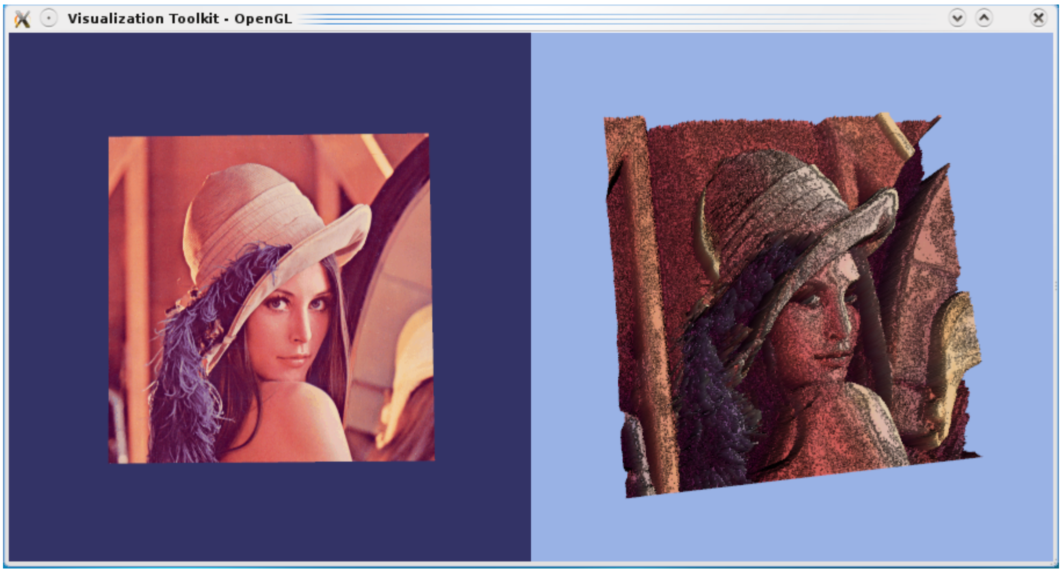
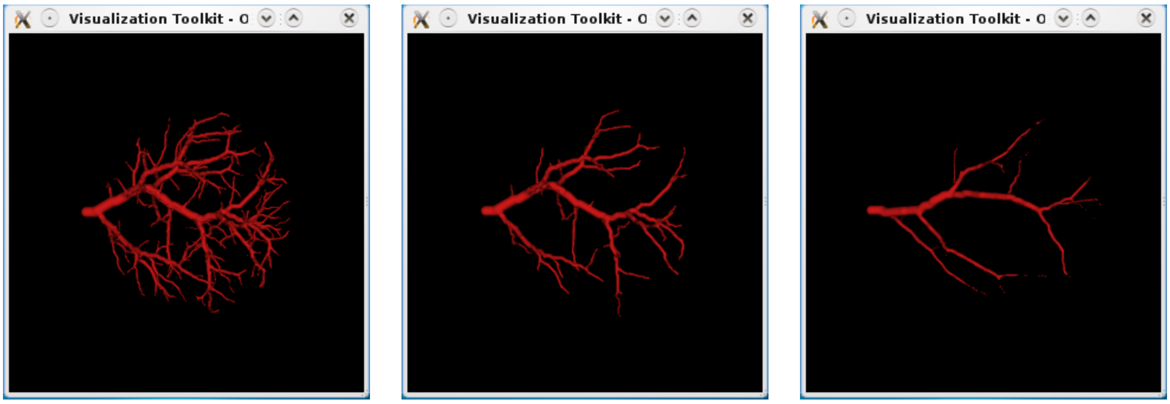


Figure 4: Imaged displayed with the use of *vtkImageViewer2* and warped the data in the direction perpendicular to the image plane

```
ren1 .AddActor (imageActor )
ren1 .SetViewport (0.0, 0.0, 0.5, 1.0)

ren2 = vtk .vtkRenderer()
ren2 .SetBackground( 0.6, 0.7, 0.9)
ren2 .SetViewport (0.5, 0.0, 1.0, 1.0)
ren2 .AddActor (warpActor )

renderWindowInteractor = vtk.vtkRenderWindowInteractor ()
renWin =vtk.vtkRenderWindow ()
renWin .AddRenderer (ren1)
renWin .AddRenderer (ren2)
renWin .SetInteractor (renderWindowInteractor )
renWin .SetSize (900,450)
renWin .Render ()

renderWindowInteractor .Start ()
```

12. Iso-surface extraction is possible with the use of *vtkMarchingCubes* algorithm. Run follwing code. Play with *SetValue(...)* function in range (10–255) (Fig. 5).

```
import vtk

imageReader = vtk.vtkImageReader()
```

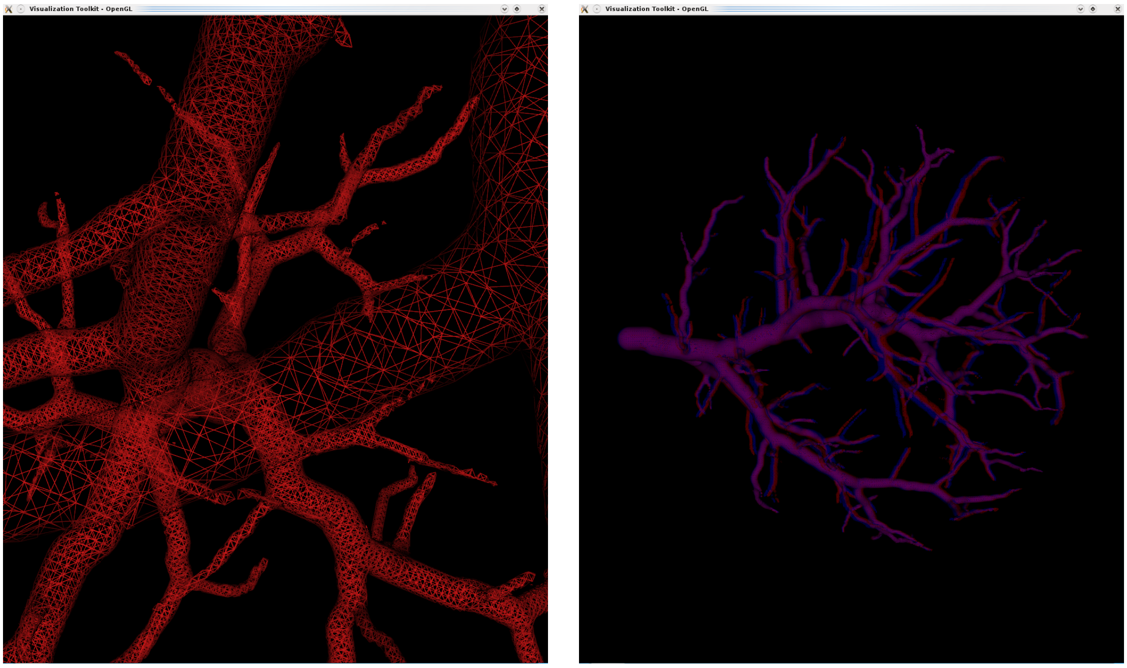


(a) iso-value=50　　*(b) iso-value=100*　　*(c) iso-value=200*

Figure 5: Extraction of surface of vascular tree with different iso-value

```
imageReader.SetFileName("qinp01_3000_036_3_256.raw")
imageReader.SetDataScalarTypeToUnsignedChar()
imageReader.SetDataByteOrder(0)
imageReader.SetFileDimensionality(3)
imageReader.SetDataOrigin(0,0,0)
imageReader.SetDataSpacing(1,1,1)
imageReader.SetDataExtent(0,255,0,255,0,255)
imageReader.SetNumberOfScalarComponents(1)
imageReader.Update()


shrinker = vtk.vtkImageShrink3D()
shrinker.SetInput(imageReader.GetOutput())
shrinker.SetShrinkFactors(1,1,1)
shrinker.AveragingOn()

gaussian = vtk.vtkImageGaussianSmooth()
gaussian.SetDimensionality(3)
gaussian.SetStandardDeviations(1.0, 1.0, 1.0)
gaussian.SetRadiusFactor(1.0)
gaussian.SetInput(shrinker.GetOutput())

marching = vtk.vtkMarchingCubes()
marching.SetInput(gaussian.GetOutput())
marching.SetValue(1,100)
marching.ComputeScalarsOff()
marching.ComputeGradientsOff()
marching.ComputeNormalsOff()
```

```
decimator = vtk.vtkDecimatePro ()
decimator .SetInput (marching .GetOutput ())
decimator .SetTargetReduction (0.1)
decimator .SetFeatureAngle (60)

smoother = vtk .vtkSmoothPolyDataFilter()
smoother .SetInput (decimator.GetOutput())
smoother .BoundarySmoothingOn()
smoother .FeatureEdgeSmoothingOn ()

normals = vtk.vtkPolyDataNormals ()
normals .SetInput (smoother .GetOutput())
normals .SetFeatureAngle (60)

stripper = vtk.vtkStripper ()
stripper .SetInput (normals.GetOutput ())

mapper = vtk.vtkPolyDataMapper ()
mapper .SetInput (stripper .GetOutput())
mapper .ScalarVisibilityOff ()

surf = vtk.vtkProperty ()
surf .SetColor(0.8,0.1,0.1)


actor = vtk.vtkActor()
actor .SetMapper (mapper)
actor .SetProperty (surf)

ren1 = vtk.vtkRenderer()
ren1 .AddActor (actor)

renWin = vtk.vtkRenderWindow ()
renWin .AddRenderer (ren1)

iren = vtk.vtkRenderWindowInteractor ()
iren .SetRenderWindow (renWin)

renWin.Render()
iren .Start ()
```



(a) *mesh* of extracted vascular tree　　　　*stereo* mode

Figure 6: Different modes implemented in *vtk*

13. By pressing “w”/“s” key, one can switch between *wire* and *surfce* mode. 3*D stereo normal* mode is accessible with key: “3”