Version: February 7, 2018

Image Processing in NUMPY

Exercises

1 Introduction

During this exercise, you will become familiar with image processing in PYTHON. Afterwards you will hopefully have a better feel for the connection between an algorithm described as mathematical formulas (as e.g. in the book *Computer Vision: Algorithms and Applications*) and how such an algorithm might be implemented.

1.1 Matlab vs python

In Python the NUMPY and SCIPY packages are used to handle mathematical operations and images respectively. Unlike Matlab, Python is a general programming language used for many things other than numerical computations. This means that you usually need to type slightly more for some mathematical operations, and far less for things like reading and writing files.

The NUMPY package is a Python interface to C and C++ libraries that perform most of the heavy lifting, this means that large operations should be performed by the appropriate NUMPY call rather than explicit looping over matrix elements. Doing explicit loops is of course still an option, but it will almost always be slower than calling NUMPY directly.

The numpy documentation is availible at: `https://docs.scipy.org/doc/`.

This exercise is also meant as an opportunity for you to refresh your knowledge of Fourier transforms, and the PYTHON environment. At the same time you will also learn some simple image-processing tricks.

1.2 Preparations

Before the exercise you should have read through this exercise guide, and do not miss the mini-reference of useful NUMPY operations at the end of the document. To have access to the appropriate version of python on the university system you need to setup some modules. This is done by typing the following lines in a terminal (or adding them to your startup script):

```
module load ext/ISY
module load courses/tsbb15
```

1.3 Starting Python

There are in practice three ways of using python, trough an interpreter such as the regular one (started by typing python in the terminal) or (started by typing ipython in the terminal) ipython, by writing a script file (ending in .py) and calling the interpreter explicitly, or finally by using a notebook (not really supported on LiU systems at the moment).

The interpreter is started by simply typing `python` into a terminal window, or alternatively `ipython`. The ipython interpreter is a bit more ergonomic for interactive use but either one should be fine here. Running a scriptfile is done by typing `python script.py` in a terminal window.

The first thing that is needed is to import some libraries, such as NUMPY and SCIPY and MATPLOTLIB for visualization. This is done by typing:

```python
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

either in a Python interpreter or at the top of a Python file/notebook. It will import `numpy` with the `np` alias, meaning you would write np instead of numpy, and `plt` instead of `matplotlib`. These aliases are in practice the standard convention for importing matplotlib and numpy.

2 Python/Numpy/Scipy

This part demonstrates some general features of NUMPY/PYTHON.

2.1 The most useful function: the help function

When need to know something about a Python object it is useful to call the help function. This particularly applies when typing directly into the Python interpreter. For any function, class or module simply type:

```
help(numpy)
```

For exiting the help text viewer (in the terminal) press q. Alternatively the numpy documentation at `https://docs.scipy.org/doc/` is well written and generally useful.

3 Simple array operations and creation

A numpy array is created by calling the np.array function, or one of the specialized functions for creating pre-filled arrays of some kind such as np.random.randn.

```
row_vector = np.array([1,2,3], dtype='float32')
small_matrix = np.array([[0,1,2],[3,4,5],[7,8,9]], dtype='float64')
identity_matrix = np.identity(3)
zeros = np.zeros((3,3))
```

This code creates a single 3 element vector, as well as some 3x3 arrays with specified data. As can be seen on the second line, creating a matrix is done by specifying each dimension as a separate list. The `dtype='float32'` argument creates an array with the type `float32`, if nothing is specified the `float64` datatype is used. There are a few other types such as integers or bytes (8-bit integers).

For this example each list is a row, with the number of elements in each list being the number of columns. Since numpy can have arbitrary number of dimensions for the arrays it is also possible to specify for example a 3-index array like:

```
array3x3 = np.array([[[1,2,3],[1,2,3],[1,2,3]],
                     [[123,123,123],[321,321,321],[1,3,2]],
                     [[5,6,7],[8,9,10],[11,12,13]]])
```

Indexing a matrix can be done a number of ways, more about that later.

3.1 Simple plotting

Plotting a curve can be done by calling:

```
#create a sine curve function (or more correctly sample it)
sine_curve = np.sin(np.linspace(0,1,1000)*np.pi*2.0)
#Create the figure
plt.plot(sine_curve)
#Display the figure
plt.show()
```

The plot function can take a large number of different parameters, look them up at: `http://matplotlib.org/api/pyplot_api.html`

Since this is a course in computer vision it is important to know how to read images:

```
import matplotlib.image as mpimg

#Read image:
image = mpimg.imread('filename.png')

#Show the image in a new window:
plt.figure() #This opens a new window.
plt.imshow(image) #Displays the image in the window.
plt.show() #Tells the system to open the windows.
```

Question: How would you make a plot with a magenta dash-dotted line with hexagrams at each data point? (hint: check the help function for plot)

3.2 Operation timing in ipython

When using the Ipython interpreter, or a notebook some additional commands are availibile to for example benchmark different functions. For example, to measure how long it takes to compute a line one can write:

```
py = range(1,10000)
%timeit [i+1 for i in py]
```

This will create a long list of numbers and add one to each one in turn. The corresponding operation using numpy is:

```
%timeit numpy = np.arange(1,10000) + 1
```

In both cases the %timeit command tells the interpreter to run the following line multiple times and compute the mean and standard deviation of the runs. Note that commands starting with % are specific to ipython, and will not work when using the normal python interpreter.

3.3 Indexing and masks

In order to use the vector operations you may have to extract parts of matrices. In NUMPY indexing of arrays can be done in the same way as for lists or other python sequences, as well as in a number of additional ways. Experiment a bit with this, for example::

```

A = np.random.rand(9,9)
print(A)

print(A[1,2])
print(A[0:2,1])
print(A[0:3,0])
print(A[-1,-1])
print(A[-2,0:3])
mask = A > 0
print(A[mask])
mask = A == 0
print(A[mask])

#what is shape and size?
print(A.shape[0] * A.shape[1] - A.size)
```

Note that the first index is 0, and the last is `length` − 1 just as in most programming environments (exceptions are MATLAB and the PASCAL language).

3.4 Reshaping and repeating matrices

Have a look at the `reshape`, `tile` and `flatten` commands. What do the following commands do?

```
matrix_as_vector = np.reshape(A,(1,A.size))
matrix_as_vector = A.flatten()
vector_as_matrix = np.reshape(matrix_as_vector,(3,3))
repeated_vector = np.tile(matrix_as_vector, (1,3))
```

3.5 Python function

Write a function called imagebw that will show a grayscale image. This function should take two parameters, the first parameter is the image to show, the second is how to display it. The function should change to a grayscale colormap, if the second parameter is 0 the colors should be mapped so that 0 is black and 255 is white. If the second parameter is > 0 the colors should be mapped so that the value of the parameter is used as black, and the maximum image value is white.

4 Test-pattern generation

We start by generating two simple gradients (note that arange does not include its endpoint, thus the seemingly assymetric range):

```
(x,y) = np.meshgrid(np.arange(-128,129,1),np.arange(-128,129,1))

plt.figure(1)
plt.imshow(x)
plt.figure(2)
plt.imshow(y)
```

Now, use these arrays to generate an image of the function $r = \sqrt{x^2 + y^2}$.
Question: Write down the commands you use here:

Question: What is the function r?

We will now use this function to generate our first test image:
Generate the function $p1 = \cos(r/2)$ and display it.
Generate the Fourier transform of this pattern as variable using np.fft.fft2 $P1$, and plot it.
Question: What does the Fourier transform look like? Explain why:

Question: What is the difference between `np.fft.fftshift` and `np.fft.ifftshift`

We will now create a test pattern with a slightly more interesting frequency content. Use the variable `r` to create the function $p_2 = \cos(r^2/200)$.

Plot the image `p2`, and its Fourier transform `P2`.

Question: The function $p_2 = \cos(r^2/200)$ appears to be rotationally symmetric, and its Fourier transform should then be rotational symmetric. Why then isn't the Fourier transform of `p2` rotationally symmetric?

To get a slightly better result, we now cut off the outermost periods of the cos() signal:

```
p2 = np.cos((r**2 / 200.0)) * ( (r**2 / 200.0) <  (22.5 * np.pi ) )
P2 = np.fft.fft2(p2)
#** is the exponentiation operation, so x**2 = x^2
```

You should now get a somewhat more correct representation of the transform `P2`, but note that the superimposed ringings are still there.

To generate a test image with a controlled frequency content is actually not that easy. The main problem is to create a test image which locally contains one frequency and has a nice behavior in the transition from the dc part to the low frequency part.

5 Filtering

We will now low-pass filter our test image. We start with filtering in the Fourier domain. Here the following variables will probably come handy:

```
u=x/256*2*np.pi
v=y/256*2*np.pi
```

Question: How do you generate a low-pass filter in the Fourier domain?

Assuming that the Fourier domain spans the range $[-\pi, \pi]$, use the method you describe above to generate an ideal LP-filter with the cut-off frequency at $\rho = \pi/4$ (Hint: A scaled r, and logical expressions are useful here.).

Use your filter on the transform `P2` of your test image `p2`.

Question: How is filtering in the Fourier domain computed?

Compute the inverse Fourier transform of the result from your filtering, and compare this with the original image `p2`.

Question: Since the image `p2` contains radially increasing frequency, one could expect that the result was attenuated outside a certain radius, and largely unaffected in the image centre. Why didn't this happen here?

We will now apply a box filter in the spatial domain. Instead of multiplication, the filtering now consists of a convolution:

```
#import convolution function
from scipy.signal import convolve2d as conv2

lp = np.ones((1,9),dtype='float32')/9.0
p2fs = conv2(lp, np.transpose(lp))
#Transpose can also be called like:
#lp.T
```

Question: This is not good either. Why does this filter behave differently in different parts of the image?

Now, try a Gaussian filter instead:

```
sigma = 3.0
lp = np.atleast_2d(np.exp(-0.5 * np.square(np.arange(-6,7,1)/sigma)))
lp = lp / np.sum(lp) #normalize the filter
```

Question: Why does this filter work better? (Hint: Look at the filter `lp'*lp`.)

6 Multi-dimensional arrays

We will now read a colour image from disk:

```
  image = mpimage.imread('some_image_file.png').astype('float32') / 255
```

The astype('float32') converts the image to float, and the division with 255 makes sure the range is $[0, 1]$ rather than $[0, 255]$.

Check the size of the array containing the image

```
  print(image.shape)
```

```
plt.imshow(image)
plt.imshow(image[:,:,1])
```

Question: Which component is which?

Convert the image to grayscale using the formula $gray = 0.299R + 0.587G + 0.114B$, and plot it where the colour image now is. Compare the grey-scale image with the RGB components. (This is best done in a function)

6.1 Complex valued images

We will now combine the results of two Gaussian derivative filterings into one complex-valued field describing image orientation. Computing the derivative of a discrete signal is in general impossible, since it is the limit of a difference quotient, and we only have a sampled signal. It is however possible to compute a *regularised derivative*, i.e. the derivative convolved with a smoothing kernel:

$$\frac{\partial}{\partial x}(f * g(\sigma))(x) = (f * \frac{\partial}{\partial x}g(\sigma))(x) = (f * \frac{-x}{\sigma^2}g(\sigma))(x)$$

Where $g = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{x^2}{2\sigma^2}}$. Thus, the regularised 1D derivative filter becomes:

```
df = np.atleast_2d(-1.0/np.square(sigma) * np.arange(-6,7,1) * lp)
```

Using `lp` and `df`, we can compute regularised partial derivatives along the x and y axes. Apply them on the test pattern `p2`. Now we store the results of filtering with these kernels as a complex field, and visualise the result using the `gopimage` command:

```
dx = conv2(conv2(paprika_red,df,mode='same'),lp.T,mode='same')
dy = conv2(conv2(paprika_red,lp,mode='same'),df.T,mode='same')
z = dx + 1i*dy

gopimage(z)
```

Each complex value in `z` can now be seen as a vector $(\mathrm{Re}(z)\ \ \mathrm{Im}(z))^T$ that approximates the image gradient.

Use the zoom tool to locate pixels of different colours, then extract the corresponding complex values from `z`. For instance you could try:

```
print(z[139,92])
```

Question: Describe what the colours mean:

Now, double the argument of the gradient image:

```
z2 = np.abs(z) * np.exp(1i * 2 * np.angle(z))
gopimage(z2)
```

Question: Compare the results, and explain why the double-angle representation is useful.

Have a look at the gradient magnitude, by computing the absolute value of the `z` variable.

Question: Why are there periodic variations in the magnitude?

6.2 2D tensor images

If we want to estimate the local orientation, the periodic variation in magnitude we saw above is not desirable. One way to remove the periodic variations is to average the double angle representation **z2**, why is it not possible to average **z**? Another way to avoid this problem is to replace the complex valued image **z** with a tensor image **T**. Each position in the tensor image **T** contains the outer product of the vector $\mathbf{f} = \begin{pmatrix} f_x & f_y \end{pmatrix}^T$, i.e. $\mathbf{T}(\mathbf{x}) = \mathbf{f}(\mathbf{x})\mathbf{f}(\mathbf{x})^T$. If we perform averaging in the tensor representation, we can smear out the magnitude such that it is largely independent of the local signal phase.

$$\mathbf{T} = \begin{pmatrix} f_x \\ f_y \end{pmatrix} \begin{pmatrix} f_x & f_y \end{pmatrix} = \begin{pmatrix} f_x f_x & f_x f_y \\ f_y f_x & f_y f_y \end{pmatrix} \quad \text{and} \quad \mathbf{T}_{\text{lp}}(\mathbf{x}) = (\mathbf{T} * g_x * g_y)(\mathbf{x})$$

Since **T** is symmetric, we only need to store three of the components in numpy.

Now generate the tensor and look at it:

```
  T = zeros((256,256,3))
  T[:,:,0] = dx * dx
  T[:,:,1] = dx * dy
  T[:,:,2] = dy * dy

  showtensor(T)
```

Next we apply Gaussian smoothing with the appropriate standard deviation σ on all the tensor components. Try to find a σ-value that gives just enough smoothing to remove the magnitude variation.

```
sigma = 3.0
lp = np.atleast_2d(np.exp(-0.5 * (np.arange(-10,11,1)/sigma)**2))
lp = lp / np.sum(lp) #normalize the filter

Tlp[:,:,0] = conv2(conv2(T[:,:,0], lp, mode='same'), lp.T, mode='same')
Tlp[:,:,1] = conv2(conv2(T[:,:,1], lp, mode='same'), lp.T, mode='same')
Tlp[:,:,2] = conv2(conv2(T[:,:,2], lp, mode='same'), lp.T, mode='same')
```

Question: Write down the required `sigma` value:

Question: The actual local orientation property is an angle modulo π. How is this angle represented in the tensor?

6.3 Images as vectors

This last exercise emphasizes the fact that we can treat an image as a vector instead of a multidimensional matrix. Treating an image as a vector has the advantage that we only need to keep track of one index instead of several. Load `mystery_vector.npy` to obtain an image stored as a vector. The exercise is to find the maximum value and the corresponding position of this value given in image coordinates. Finally reshape the vector to the original image. The original image consists of 320 columns and 240 rows. **HINT:** One useful command is `unravel_index`.

Question: What is the position of the maximum value in image coordinates?

A Some useful numpy operations

- The * symbol does elemt-wise multiplication.
- The @ symbol (python 3.5+) does matrix multiplication. (not availible on liu machines)
- The .dot operation on two arrays performs matrix multiplication, like np.dot(A,B).
- A.shape contains a tuple containing the size of each dimension.
- A.T contains the transpose of A.
- Indexing starts at 0, and ends at A.shape - 1.
- A range of numbers is specified as np.arange(0,10) = np.array([0,1,2,3,4,5,6,7,8,9]).
- Evenly spaced numbers between (inclusive) two numbers is generated as: np.linspace(0,1,10) = array([1. ,0.875, 0.75, 0.625, 0.5]).
- Taking the power of a number can be done by **, like 2**2 = 4, or 3**3 = 27, also works for numpy arrays.