Image Processing and Computer Vision

Lab: Python for IP & CV

This is the first year that the course is Python/ Numpy /OpenCV based instead of Matlab based. This is absolutely *not* because Matlab per se is not suitable for an image processing and computer vision course. In fact a lot of scientists (and practioners) all over the world use Matlab.

There are however disadvantages to Matlab and advantages to the combination of Python/Numpy/OpenCV that lead me to switch to the latter:

1. The Matlab *language* is inferior to the Python language. Computer scientists tend to dislike Matlab for programming. On the other hand the *library* Matlab is far better then the Numpy mathematical libraries (especcially how it hides a lot of mathematical details from the casual user).
2. Matlab is way too expensive for the Dutch educational system. At least the University of Amsterdam cannot afford to provide student licenses for all students to enable them to work for a Matlab based course using their own computers.
3. Matlab is way way too expensive for small companies as a basis for their scientific computing needs. Where it used to be the case that students raised in Matlab could use that knowledge in commercial products they developed later on in their career, nowadays having developed in Matlab in university courses forces students to switch to other possibilities in a (small scale) commercial setting.
4. Python/Numpy is catching up very fast. I hope to prove in this course that it is viable alternative.
5. Python is the language of choice for many computer scientists. It is the language that looks most like pseudo code. It has many, many libraries available for a lot of different application aread (including image processing and computer vision).
6. The main advantage of Python is that it is open source. For education at a computer science department that is essential (you are learned to design programs, to do the things unimagined thusfar; not to use programs...)

I hope that all students in the 2011-2012 image processing and computer vision course will be able to fullfill the requirements for this course in a doable manner. Please help me build a better course by commenting on the text, the lab exercises and the programs whereever you think necessary.

Exploring Python/Numpy/Scipy/Matplotlib

The combination of Python (the language), Numpy (the numerical array lib), SciPy (scientific libs) and Matplotlib (the graphical plot lib) will serve as our computational basis to learn image processing and computer vision. Where possible we will use other libraries as well. We will use the name ‘Python’ to refer to this collection of tools.

The best way to explore Python is by using it. We assume that you have `ipython` installed on your computer. All interaction shown is from actual `ipython` sessions. Start `ipython` with the command `ipython --pylab`, this will start the Python interpreter and will load some of the libraries we will need.

You can start right away typing commands into the command line window:

```
In [2]: x = 2
```

The interpreter executes this statement and shows nothing (meaning that there were no errors). We may look at the value of variable `x` by just typing its name:

```
In [3]: x
Out[3]: 2
```

In Python there is no need to declare variables before using them.

If you start losing the overview over which variables you have already used and which data they contain, the command `whos` can help you out.

```
In [4]: whos
Variable   Type    Data/Info
----------------------------
x          int     2
```

In case you are not familiar with Python we suggest that you work your way through one of the tutorials that are available on the web (see some suggestions *here*).

In Python the list and dictionary are probably the most important datastructures. A list is a heterogeneous sequence of values (numbers, strings, lists, dictionaries, functions, really anything).

```
In [5]: a = [ 1, 2, 3, 4, 5 ]
```

TABLE OF CONTENTS

SEARCH

Go

Enter search terms or a module, class or function name.

```
In [6]: a[0]
Out[6]: 1

In [7]: a[3]
Out[7]: 4

In [8]: a[-1]
Out[8]: 5
```

Indexing is done using square brackets and indexing starts at 0. Negative indices start counting from the back to the front.

Using nested lists you may construct something that we would call a multidimensional list (or array) in C.

```
In [9]: a = [ [1,2,3], [4,5], [6] ]

In [10]: a[1]
Out[10]: [4, 5]

In [11]: len(a)
Out[11]: 3

In [12]: len(a[1])
Out[12]: 2

In [13]: a[0][1]
Out[13]: 2
```

But because of the list being heterogeneous and nested lists all can have their own length the list is not the preferred way to encode multidimensional arrays as the often are encountered in numerical computing.

The best way of learning Numpy is by reading the official tutorial: Tentative Numpy Tutorial.

Python for Image Processing

To demonstrate the use of Python for image processing we will look at two algorithms in some depth here: contrast stretching and linear filtering.

Contrast Stretching

Contrast stretching is described in *Histogram Based Operations*. Let f be a scalar image within the range $[0,1]\in\mathbb{R}$Then the contrast stretched image $g$is defined as:

$$g(\mathbf{x}) = \frac{f(\mathbf{x}) - f_{\text{min}}}{f_{\text{max}} - f_{\text{min}}}$$

Observe that this is a point-operator and we therefore do not need an explicit loop over all pixels (see *Point Operators*).

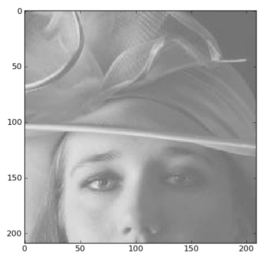
```
def cst( f ):
  fmin = amin(f) # the min value in an array
  fmax = amax(f) # the max value in an array
  return (f-fmin)/(fmax-fmin)
```

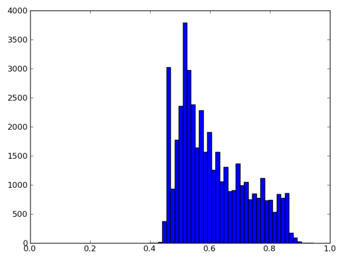
The image `lowcontrast.npy` uses only part of the range from 0 to 1. Make an histogram of this image and of the contrast stretched version of it.

The cst operator is so common when displaying images that it is even built into the matplotlib imshow function. For floating point images the values are always stretched to comprise the whole range from 0 to 1. To undo this automatic stretching you have to use something like:

```
In [14]: from pylab import *;
In [14]: a = np.load("images/lowcontrast.npy");
In [14]: clf(); imshow(a,vmin=0,vmax=1);
In [14]: h, be = histogram(a.flatten(), bins=40)

In [15]: clf(); bar( be[0:-1], h, width=diff(be)[0] ); xlim( (0,1) );
```





Exercise:

Run your version of the `cst` algorithm and plot its histogram.

Linear Filtering

With little doubt the linear filter is the most used filter in the image processing toolbox. It simply calculates a weighted summation of all pixel values in a small neighborhood (small compared to the image size). It does so for all shifted neighborhoods in an image and all weighted sums thus form a new image. In signal processing this would be called a moving (weighted) average filter.

Let F be the discrete function with all samples of the original image and let W be the weight function, the result image G then is called the correlation of F and W:

$$G(i,j) = \sum_{k=-\infty}^{+\infty} \sum_{l=-\infty}^{+\infty} F(i+k, j+l)\, W(k,l)$$

Although there are built-in functions that calculate the linear filter efficiently, here we are showing a pure Python implementation. In the above mathematical expression we assumed the image (and weight function) to be infinitely large (hence the indices that run from $-\infty$ to $+\infty$).

In practice we have to deal with the fact that the image is defined only on a finite domain. Then the indices $(i+l, j+l)$ can point *outside* the image (see the border problem) and we have to decide what value to use for $F(i+k, j+l)$.

In practice the weight function has non zero values only in small neighborhood of the origin. Assume that $W(k,l)$is non zero only in the neighborhood for $k = -K,..,+K$and $l = -L,..,+L$

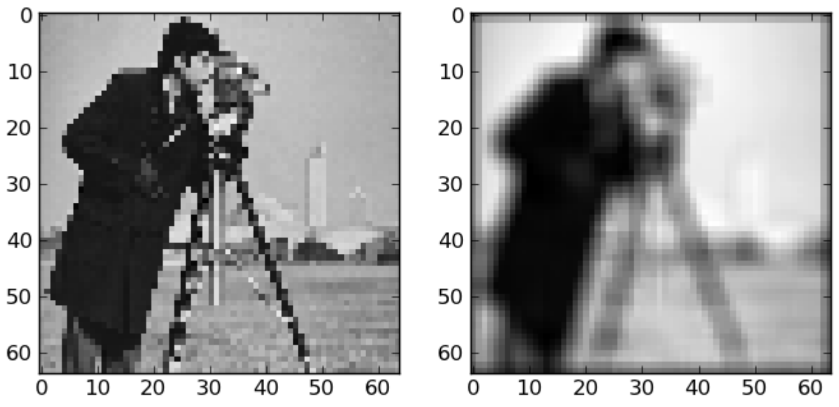
```
def linfilter1(f, w):
  g = empty(f.shape, dtype=f.dtype)  # the resulting image
  M,N = f.shape
  K,L = (array(w.shape)-1)/2

  def value(i,j):
    """The function returning the value f[i,j] in case
    (i,j) in an index 'in the image', otherwise it return 0"""
    if i<0 or i>=M or j<0 or j>=N:
      return 0
    return f[i,j]

  for j in xrange(N):
    for i in xrange(M):
      summed = 0
      for k in xrange(-K,K+1):
        for l in xrange(-L,L+1):
          summed += value(i+k,j+l) * w[k+K,l+L]
      g[i,j] = summed
  return g
```

Note that the correlation function `linfilter1` has 4 nested loops. For large images and large kernels it explains why even the simplest image processing function are so processor intensive. Because of the interpreted nature of the Python language writing a 4 levels nested loop is bound to result in a very slow program. Test this function on a small image.

```
In [15]: from linfilters import *;
In [15]: from scipy.ndimage.interpolation import zoom;
In [15]: from pylab import *;
In [15]: a = imread('images/cameraman.png');
In [15]: f = zoom(a, 0.25);
In [15]: g = linfilter1(f, ones((5,5))/25);
In [15]: subplot(1,2,1);
In [15]: imshow(f);
In [15]: subplot(1,2,2);
In [15]: imshow(g);
```



Python doesn't like explicite loops (i.e. they are slow). In many cases we can come up with other ways to implement the same function and circumvent some explicit loops (obviously the loops over all pixels and all neighbors must be there somewhere but 'hided' in Python function (written in C) we can achieve considerable speedups).

```
def linfilter2(f, w):
  """Linear Correlation based on neigborhood processing without loops"""
  g = empty(f.shape, dtype=f.dtype)
  M,N = f.shape
  K,L = (array(w.shape)-1)/2

  for j in xrange(N):
    for i in xrange(M):
      ii = minimum(M-1, maximum(0, arange(i-K, i+K+1)))
      jj = minimum(N-1, maximum(0, arange(j-L, j+L+1)))
      nbh = f[ ix_(ii,jj) ]
      g[i,j] = ( nbh * w ).sum()
  return g
```

This version circumvents the double loop over all pixels in the neighborhood. It extracts a detail image of the correct size. The correct multi-indices are build with the ix_ function.

Instead of 'hiding' the loops over all pixels in the neighborhood we can hide the loops over all pixels in the image. Observe that a correlation is equal to the addition of shifted and weighted images.

```
def linfilter3(f, w):
  """Linear Correlation using Translates of Images"""

  M,N = f.shape
  K,L = (array(w.shape)-1)/2

  di,dj = meshgrid(arange(-L,L+1), arange(-K,K+1))
  didjw = zip( di.flatten(), dj.flatten(), w.flatten() )

  def translate(di,dj):
    ii = minimum(M-1, maximum(0, di+arange(M)))
    jj = minimum(N-1, maximum(0, dj+arange(N)))
    return f[ ix_(ii, jj) ]

  r = 0*f;
  for di, dj, weight in didjw:
    r += weight*translate(di,dj)
  return r
```

And the easiest alternative is the builtin function correlate:

```
def linfilter4(f, w):
  return correlate(f, w, mode='nearest')
```

Exercise:

Time these alternatives using an image of size 64x64 for several sizes of the neighborhood (from 3x3 to 11x11 eg). For timing purposes look at the `Timer` class in module `ipcv.utils.timing`. Be careful when timing very fast operations; you better time the repitition of several calls.

Make a plot with the timings as function of the neighborhood size for all filters.