SciPy and OpenCV as an interactive computing environment for computer vision

Thiago Teixeira Santos
thiago.santos@embrapa.br
Embrapa Agricultural Informatics

SIBGRAPI 2014, Tutorials
August 27, 2014
Rio de Janeiro, Brazil

In research and development (R&D), interactive computing environments are a frequently employed alternative for data exploration, algorithm development and prototyping. In the last twelve years, a popular scientific computing environment flourished around the Python programming language. Most of this environment is part of (or built over) a software stack named the SciPy Stack. Combined to the OpenCV's Python interface, this environment becomes an alternative for current computer vision R&D. This tutorial introduces such an environment and shows how it can address different steps on computer vision research, from initial data exploration to parallel computing implementations. Several code examples are presented, addressing problems from simple image processing to inference by machine learning.

About these notebooks

These notebooks are part of the tutorial *SciPy and OpenCV as an interactive computing environment for computer vision* to be presented at SIBGRAPI 2014. The tutorial will be also available as a full paper in *Revista de Informática Teórica e Aplicada (RITA)*.

1 Introduction

1.1 What are we looking for in a computing environment?

- **Data exploration and visualization**
  - Images, video sequences, point clouds and feature vectors
- **Extensive list of tools**
  - Image processing
  - Machine learning
  - Optimization
  - Statistics
  - Linear algebra
- **High-performance computing** (HPC)
- Promote **reproducible research**

Computer vision practitioners need a computing environment that lets them explore data like images, video sequences, point clouds and feature vectors. Such an environment should help on the development and testing of new models and algorithms, and on the deployment of the results, either as a final software module or a scientific/technical publication. It should also provides a *extensive* list of tools, routines for image processing, machine learning, statistical inference, linear algebra, convex optimization and graph algorithms, just to name a few. Problems involving large sets of images can require high-performance computing (HPC)

such that the environment should provide practical ways to parallelize and distribute computation. An ideal environment should also combine documentation and computation in a single bundle, promoting results reproducibility, but preventing the research pipeline to become more cumbersome.

1.2 A Python-based environment

http://xkcd.com/353/ by Randall Munroe

- *SciPy*, a **scientific computing** environment based on Python
- Developed in the last **12 years**
- Based on the **NumPy** module for n-dimensional arrays
- Not just software, but a **community**
- This community meets at SciPy Conferences

In the last twelve years, a powerful scientific computing environment emerged from the Python programming community. This language is an attractive option for researchers: it is interpreted (a wanted property for interactive computing environments), dynamically typed, and presents a very concise and elegant syntax, resembling the pseudo-code found in computer science textbooks. But the tipping point for Python to become a major player in scientific computing was the advent of an efficient module for n-dimensional array representation and manipulation. The Numarray module was created by Greenfield *et al.* to address astronomical data analysis. In 2005, Numarray successor, NumPy, appeared and became the workhorse of the Python scientific computing. An active community composed by scientists and engineers flourished around Python and NumPy, represented today by the SciPy Stack and the SciPy Conferences.

1.3 Why is an interactive environment important to computer vision?

In computer vision, the practitioner is interested in inferring the **world state** from images, that act as **observations**. The statistical relation between the world state and the observed images is defined by **models**. A particular model is defined by **parameters**, chosen by **learning algorithms**. Finally, the world state is estimated by **inference algorithms**.

This *vision on computer vision* is properly presented by Prince in his book and translates the state-of-the-art of contemporary research in the field, which is deeply associated to machine learning nowadays.

- Environment should enable tinkering with machine learning techniques
- Visualization and exploration tools to address problems involving:
  - generalization,
  - overfitting,
  - dimensionality reduction,
  - optimization
- The SciPy environment provides these capabilities

The development of these models and subsequent problems in learning and inference require a computing environment that allows proper tinkering with machine learning techniques. Visualization and exploration tools are necessary to address problems involving generalization, overfitting, dimensionality reduction and optimization. An interactive computing environment as IPython, enriched with tools from the SciPy Stack and the OpenCV library, can address these needs.

1.4 This tutorial presents:

- interactive computing *and* HPC with the **IPython** shell;
- **OpenCV** use under Python;
- **Matplotlib** for visualization and 2D plotting;
- the SciPy **scientific library** and

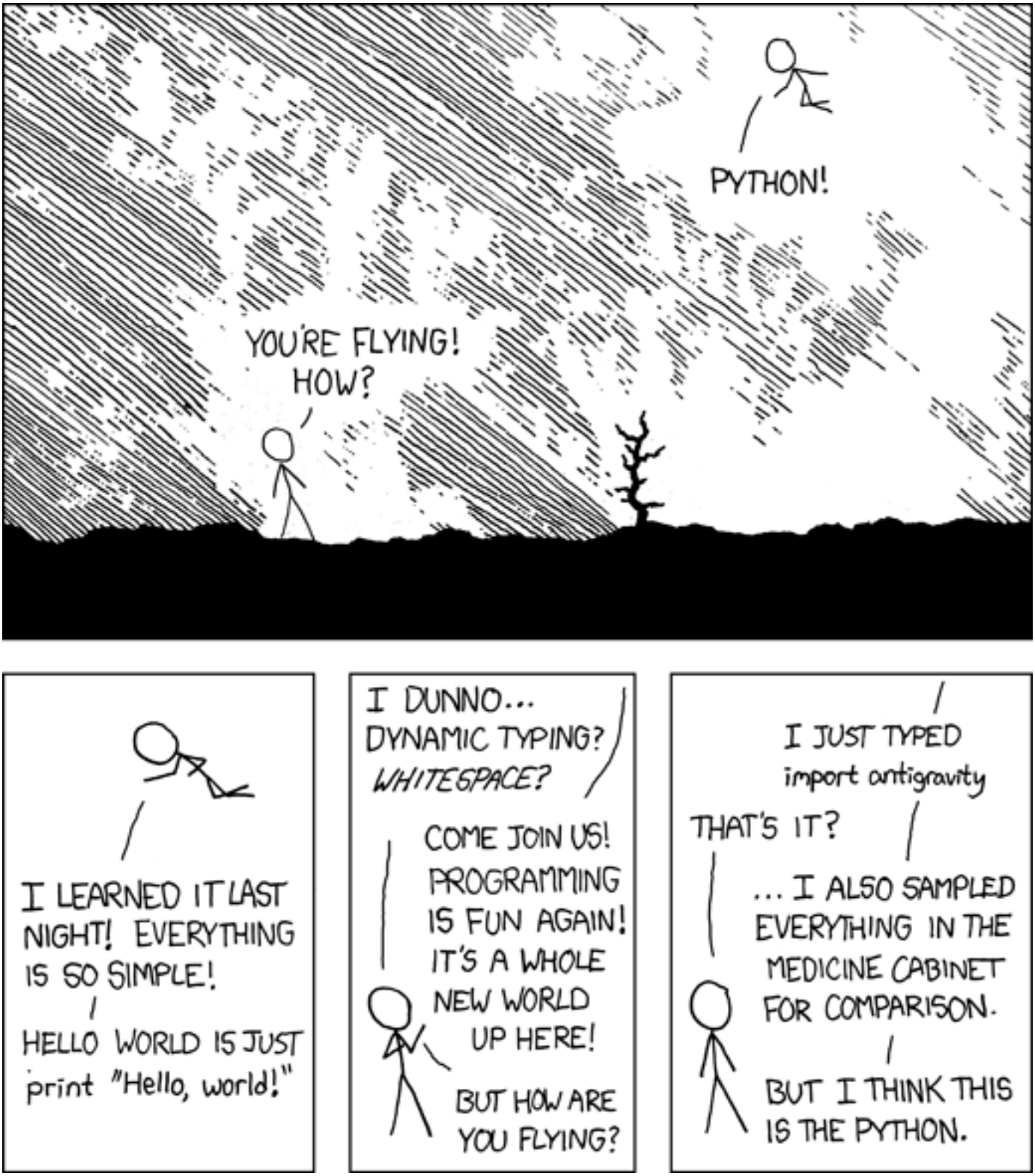


Figure 1: xkcd on Python

- machine learning with the **scikit-learn** module

- This tutorial provides a *glimpse*, not a broad and deep coverage
- Examples should provide a starting point for the interested user

The present tutorial provides a short overview on this Python-based computing environment. Considering the large set of tools available and the space constraints, this tutorial does not pretend to be a complete reference or a broad review. It provides just a glimpse of the environment's capabilities to the computer vision community, briefly presenting and discussing some problems and code examples. These examples can guide the user's first steps and the provided references help on the next ones.

1.5 Available material

IPython notebooks presenting the complete code for the examples are available on
http://nbviewer.ipython.org/github/thsant/scipy4cv
and on GitHub:
https://github.com/thsant/scipy4cv

1.6 Installation

1.6.1 Windows and Mac

Consider **Continuum Analytics's Anaconda distribution**:
http://continuum.io/downloads

1.6.2 Linux

- Consider your system's package manager
- For Debian/Ubuntu:

```
$ sudo apt-get install ipython ipython-qtconsole ipython-notebook

$ sudo apt-get install python-scipy python-sklearn
```

1.6.3 Using Python `pip`

```
$ pip install ipython scipy scikit-learn
```

1.6.4 OpenCV

OpenCV's documentation provides installation tutorials

As a general guideline, users should always consider the most recent installation instructions provided by the packages' maintainers 12 . To the date, Windows, Mac and Linux users could consider Continuum Analytics's Anaconda distribution as one of the easiest ways to get all the environment presented in this tutorial. Linux users can also use the package manager of their systems to effortless download and install the components (as the `apt-get` system in Debian/Ubuntu distributions). The pip tool is also a practical way to get the packages and keep them updated

2 The IPython shell

- IPython is an *enhanced* Python shell for **interactive** computing
- Provides facilities for distributed and parallel computing
- Allows interactive evaluation of results

```
In [1]: r = 3.
        2 * pi * r
```

```
Out[1]: 18.84955592153876
```

> This flexible style matches well **the spirit of computing in a scientific context**, in which **determining what computations must be performed next often requires significant work**. An interactive environment lets scientists look at data, test new ideas, combine algorithmic approaches, and evaluate their outcomes directly.
>
> Pérez and Granger, *IPython: A System for Interactive Scientific Computing*

IPython is an enhanced Python shell for interactive distributed and parallel computing. When users are testing new ideas and algorithms in computer vision, evaluating the results directly, in an interactive way, is more convenient than the traditional *compile-then-execute* cycle. A live computing state is convenient because previous intermediate results (images, feature vectors, parameters) are kept available for exploration. Pérez and Granger argue that, in a research context, determining which computations must be performed next often requires significant work.

2.1 Features

- Access to *all session state*
- `TAB` completion
- System shell integration
- Dynamic introspection and help
- Direct manipulation of objects in memory
- *Magic commands* (secondary control system)

2.1.1 Session state

```
In [2]: 2 * _1

Out[2]: 37.69911184307752

In [3]: _1

Out[3]: 18.84955592153876

In [4]: r

Out[4]: 3.0
```

In an interactive computing system, users should have access to all session state. In IPython, this access is not just provided by the dynamically attributed variables, but also by numbered output prompts. Previous computations can be retrieved using and underscore "_" and the number of the output.

2.1.2 System shell integration

Other useful capabilities in IPython are `TAB` completion and system shell integration. When the user types the `TAB` key, IPython tries to complete the current prompt with keywords or the names of methods, variables and files in the current directory. This is particularly useful when exploring unfamiliar or large APIs as in OpenCV or in the SciPy library. Regarding system shell integration, IPython cannot only call any system command, but also can capture the shell's output in Python variables and call system commands with values computed from variables.

Call a system's command:

```
In [5]: ls ../data/*.jpg

../data/BSD-118035.jpg  ../data/BSD-65019.jpg  ../data/girl.jpg       ../data/skin-training.jpg
../data/BSD-176035.jpg  ../data/DC_0420.jpg    ../data/skin-test.jpg  ../data/thiago.jpg
```

Store the command output in a Python variable:

```
In [6]: jpgs_list = !ls ../data/*.jpg
        jpgs_list

Out[6]: ['../data/BSD-118035.jpg',
         '../data/BSD-176035.jpg',
         '../data/BSD-65019.jpg',
         '../data/DC_0420.jpg',
         '../data/girl.jpg',
         '../data/skin-test.jpg',
         '../data/skin-training.jpg',
         '../data/thiago.jpg']
```

Call a system's command passing arguments stored in Python variables:

```
In [7]: fname = jpgs_list[0]
        !du -h $fname
        !ls -l $fname

24K        ../data/BSD-118035.jpg
-rw-rw-r-- 1 thiago thiago 18556 Ago  6 17:06 ../data/BSD-118035.jpg
```

2.1.3 *Magic functions*

- IPython presents 98 **magic functions**
- They are a secondary command system, providing extra capabilities

```
In [8]: %magic
```

Some examples

- `%load` and `%save` - load/save code into/from IPython
- `%run`- run a Python file inside IPython as a program
- `%who` - list all interactive variables
- `%timeit` - time execution of a Python statement or expression

`%who`

```
In [9]: %who

fname         jpgs_list         r

In [10]: %who str

fname

In [11]: %who float

r
```

```
%timeit
```

```
In [12]: from random import random
         %timeit random()
```

```
10000000 loops, best of 3: 72.5 ns per loop
```

```
In [13]: # Import the linear algebra module in SciPy
         from scipy import linalg
```

Create a random 100×100 matrix A and time the Singular Value Decomposition (SVD) $A = U\Sigma V^\intercal$.

```
In [14]: %%timeit A = randn(10000).reshape(100,100)
         linalg.svd(A)
```

```
1 loops, best of 3: 20.4 ms per loop
```

2.2 The IPython notebook

- Web-based version of the IPython shell
- Chunks of text and code are organized in **cells**
- Cells can be inserted, deleted, rearranged and executed as needed
- Notebooks can be **converted** to Python programs, PDF documents and slides
- These slides and the tutorial's handouts are notebooks convertions!
- IPython notebooks are **executable documents**

The IPython *Notebook* is a web-based version of the IPython shell presenting extended functionality. In the notebook, the user can organize formatted text and code blocks in a flexible way. Text and code are organized in cells that can be inserted, deleted, rearranged and executed as needed. IPython notebooks can handle plots, mathematical formulas and code output, everything organized in a single executable document. Notebooks are being used for research notes, and on the production of articles and books.

2.2.1 Markdown cells

In a notebook, a **Markdown cell** is able the render rich-formatted text using the Markdown markup convention. Mathematical notation is defined using LaTeX syntax and rendered by MathJax.

```
Let $\mu_I$ and $\sigma_II$ be the mean and the standard deviation of a grayscale
image $I$. The whitening operation is defined by:

$W_I[i,j] = \frac{I[i,j] - \mu_I}{\sigma_I}$.
```

> Let μ_I and $\sigma_I I$ be the mean and the standard deviation of a grayscale image I. The whitening operation is defined by:
> $W_I[i,j] = \frac{I[i,j]-\mu_I}{\sigma_I}$.

2.2.2 Code cells

- Code cells keep (short) Python scripts
- The browser sends the code to the server running an *IPython Kernel*
- Output is sent back to the browser for exhibition

```
In [15]: hello = "Hello, World! I'm a string in an IPython notebook code cell."
         print hello
```

```
Hello, World! I'm a string in an IPython notebook code cell.
```

```
In [16]: hello.split()

Out[16]: ['Hello,',
          'World!',
          "I'm",
          'a',
          'string',
          'in',
          'an',
          'IPython',
          'notebook',
          'code',
          'cell.']
```

Code cells contain Python code that is sent to the IPython interpreter running on the server, executed, and the resulting output sent back to the browser for exhibition. That means that IPython can be running in a powerful machine like a server while the user is able to perform her work from a leaner system as a laptop or a tablet. If the result of a code block is a plot, it can be exhibited inside the notebook in the browser (*inline plotting*).

2.2.3 Notebook files (`.ipynb`)

- IPython notebooks are stored as *JSON files*
- Can be converted to:
  - a Python script
  - a reStructuredText document
  - a HTML document
  - a LaTeX document
  - a Markdown document
  - slides
- For details, check:

```
$ ipython nbconvert --help
```

IPython notebooks are stored in JSON files that keep the cells' content. These files uses the extension `.ipynb` and can be exported as Python scripts, HTML documents or even printed. But what makes notebook files suitable to reproducible research is the fact they are executable documents, not only able to store textual and mathematical descriptions but also replicate the computations.

2.3 Starting IPython

The IPython default console (**terminal**) can be started using:

```
$ ipython
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Other alternative is **qtconsole**, a console running on a graphical window:

```
$ ipython qtconsole
```

To run IPython in notebook mode:

```
$ ipython notebook
[NotebookApp] Using existing profile dir: u'/home/thiago/.ipython/profile_default'
[NotebookApp] Using MathJax from CDN: http://cdn.mathjax.org/mathjax/latest/MathJax.js
[NotebookApp] Serving notebooks from local directory: /home/thiago
[NotebookApp] 0 active kernels
[NotebookApp] The IPython Notebook is running at: http://localhost:8888/
[NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

- The URL to access the served notebooks is `http://localhost:8888`
- System is serving the notebooks in the directory `/home/thiago`
- User can employ `Ctrl-C` to stop the server and shutdown the IPython kernels

3 OpenCV

- Popular computer vision library
  - Used in industry and academy
- Started in 1999, popularized in 2000s
- Developed in efficient C/C++ code
- Stable Python interface since 2009

OpenCV is a popular library in the computer vision community, being actively used in industry and academy. Started in 1999 and popularized in the following decade, OpenCV is covered in books and tutorials, so this text will not provide another overview of the library. The interested reader that is not familiar to OpenCV is referred to those cited texts and the library's official documentation.

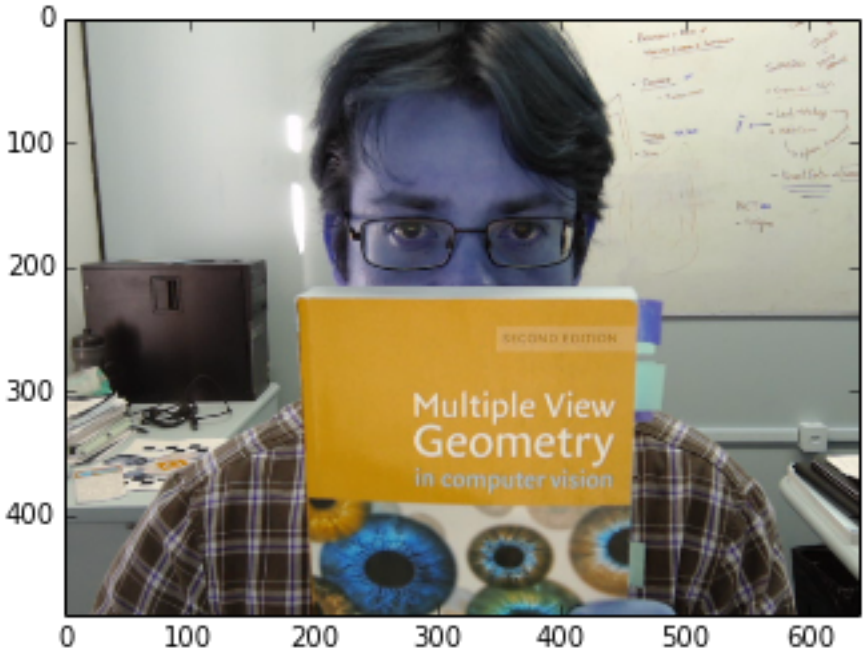
Developed in efficient C/C++ code, OpenCV presents a stable Python interface since 2009. The functions' prototypes in the Python API can differ from the C++ version, but the OpenCV documentation presents both versions for reference. IPython code completion capabilities can also help programmers used to the C++ API to quickly identify the proper Python prototypes.

3.1 Interactive OpenCV using IPython

```
In [1]: import cv2
```

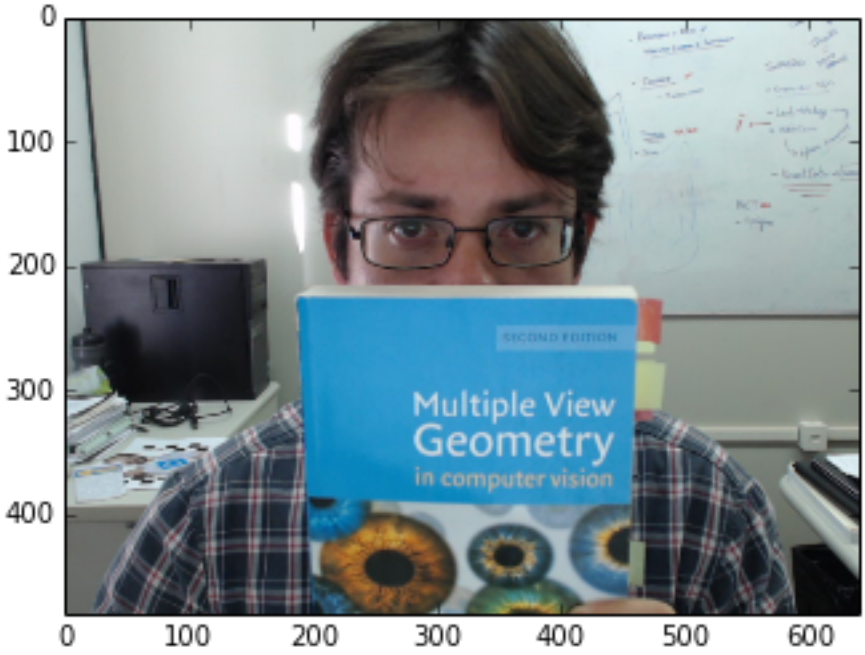
Let's play with *interactive frame grabbing* from a video camera:

```
In [2]: capture = cv2.VideoCapture(1)

In [3]: val, image = capture.read()

In [4]: val

Out[4]: True

In [5]: imshow(image)

Out[5]: <matplotlib.image.AxesImage at 0x7fa557d9f490>
```



In [6]:
```
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
imshow(image_rgb)
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x7fa54c352090>
```



3.2 What is an image in OpenCV under Python?

```
In [7]: type(image)

Out[7]: numpy.ndarray
```

- In C++, OpenCV employs its `Mat` matrix structure
- But in Python, OpenCV represents images as **NumPy n-dimensional arrays**
- Let's talk about NumPy arrays...

In OpenCV operation under Python, NumPy arrays replace the OpenCV's `Mat` type as the data structure for images. NumPy arrays will be presented in the next section.

4 NumPy

```
In [1]: import cv2
        figsize(12,8)
```

4.1 Images as NumPy arrays

- OpenCV `imread` under Python returns a **NumPy array**
- The `shape` attribute keeps the array's dimensions

In the OpenCV's Python wrapper, the `imread` function returns an image as a NumPy array. The array dimensions can be read from the `shape` attribute:

```
In [2]: lenna = cv2.imread('../data/lenna.tiff', cv2.IMREAD_GRAYSCALE)
        lenna.shape

Out[2]: (512, 512)
```

- Grayscale images can be represented as `uint8` arrays

Grayscale images are commonly represented by 2D arrays of 8 bits unsigned integers, corresponding to values from 0 ("black") to 255 ("white"). In NumPy, this **data type** (`dtype`) is named `uint8`.

```
In [3]: lenna.dtype

Out[3]: dtype('uint8')

In [4]: lenna

Out[4]: array([[162, 162, 162, ..., 170, 155, 128],
               [162, 162, 162, ..., 170, 155, 128],
               [162, 162, 162, ..., 170, 155, 128],
               ...,
               [ 43,  43,  50, ..., 104, 100,  98],
               [ 44,  44,  55, ..., 104, 105, 108],
               [ 44,  44,  55, ..., 104, 105, 108]], dtype=uint8)

In [5]: lenna[0,0]

Out[5]: 162

In [6]: imshow(lenna, cmap=cm.gray)
        colorbar()
```

```
Out[6]: <matplotlib.colorbar.Colorbar instance at 0x7ffcffb5ab00>
```

4.1.1 Color images

- A RGB color image can be represented by a $M \times N \times 3$ `uint8` array

Color images in RGB are commonly represented by 24 bits, 8 bits for each one of the three channels (red, green and blue). In NumPy, a $M \times N$ color image can be represented by a $M \times N \times 3$ `uint8` array.

```
In [7]: mandrill = cv2.imread('../data/mandrill.tiff')
        mandrill.shape

Out[7]: (512, 512, 3)

In [8]: mandrill.dtype

Out[8]: dtype('uint8')

In [9]: mandrill[2,3]

Out[9]: array([29, 46, 54], dtype=uint8)
```

- The **color triplet** can be retrieved indexing the pixel position
- Indexing can be used to retrieve a specific channel
- OpenCV `imread` returns color images in BGR order

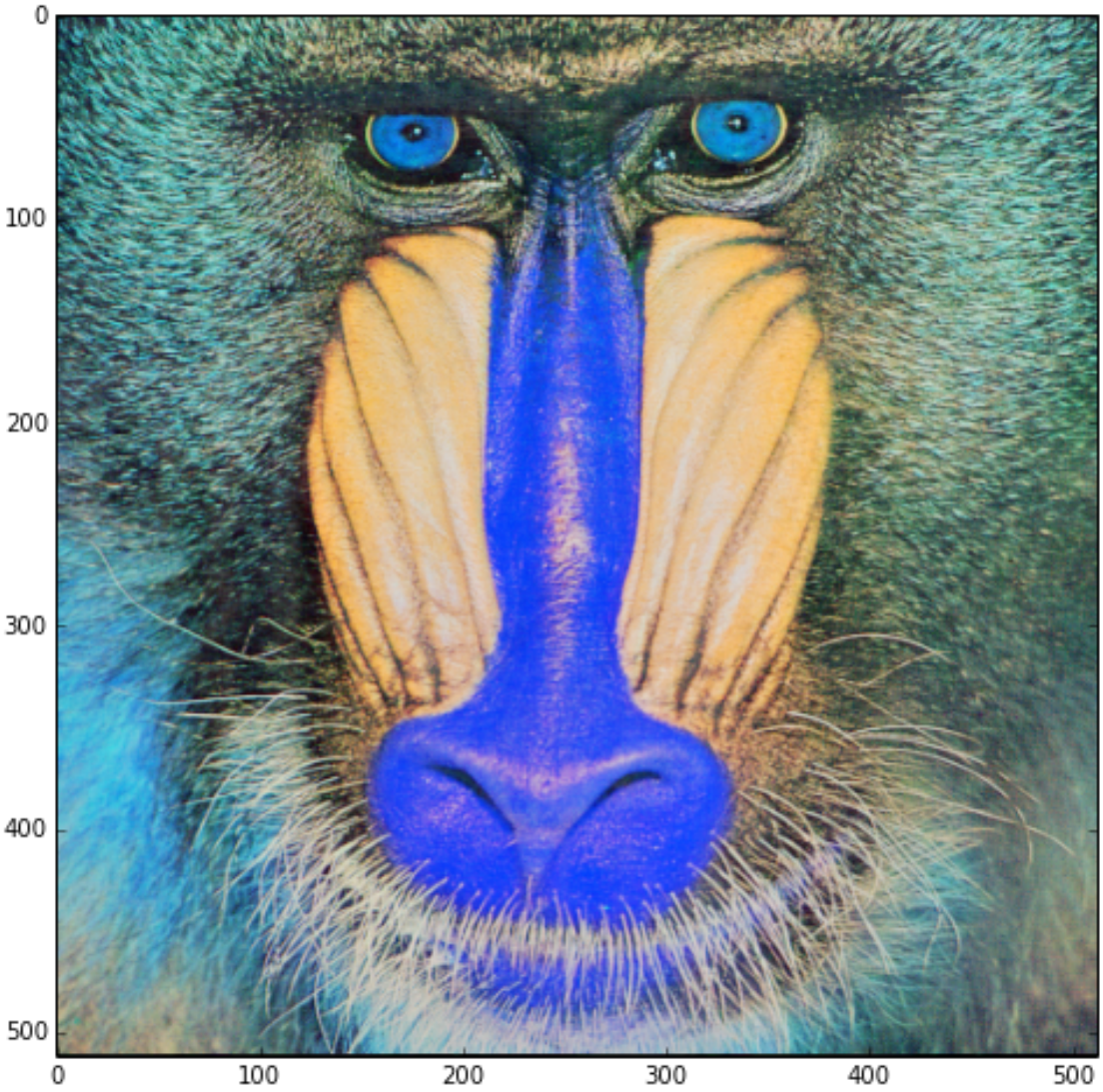
The output of the last command above shows the value at pixel $(2, 3)$ is 29, 46, 54. The image was loaded by OpenCV using the `imread` procedure. OpenCV loads color images in BGR order, so 29, 46 and 54 correspond to the values of blue, green and red respectively. The triplet is returned as a 3-d vector (a unidimensional array). Direct access to the color value can be done indexing the color dimension - `mandrill[2,3,1]` returns the green channel value, 46.

```
In [10]: mandrill[2,3,1]
```

```
Out[10]: 46
```

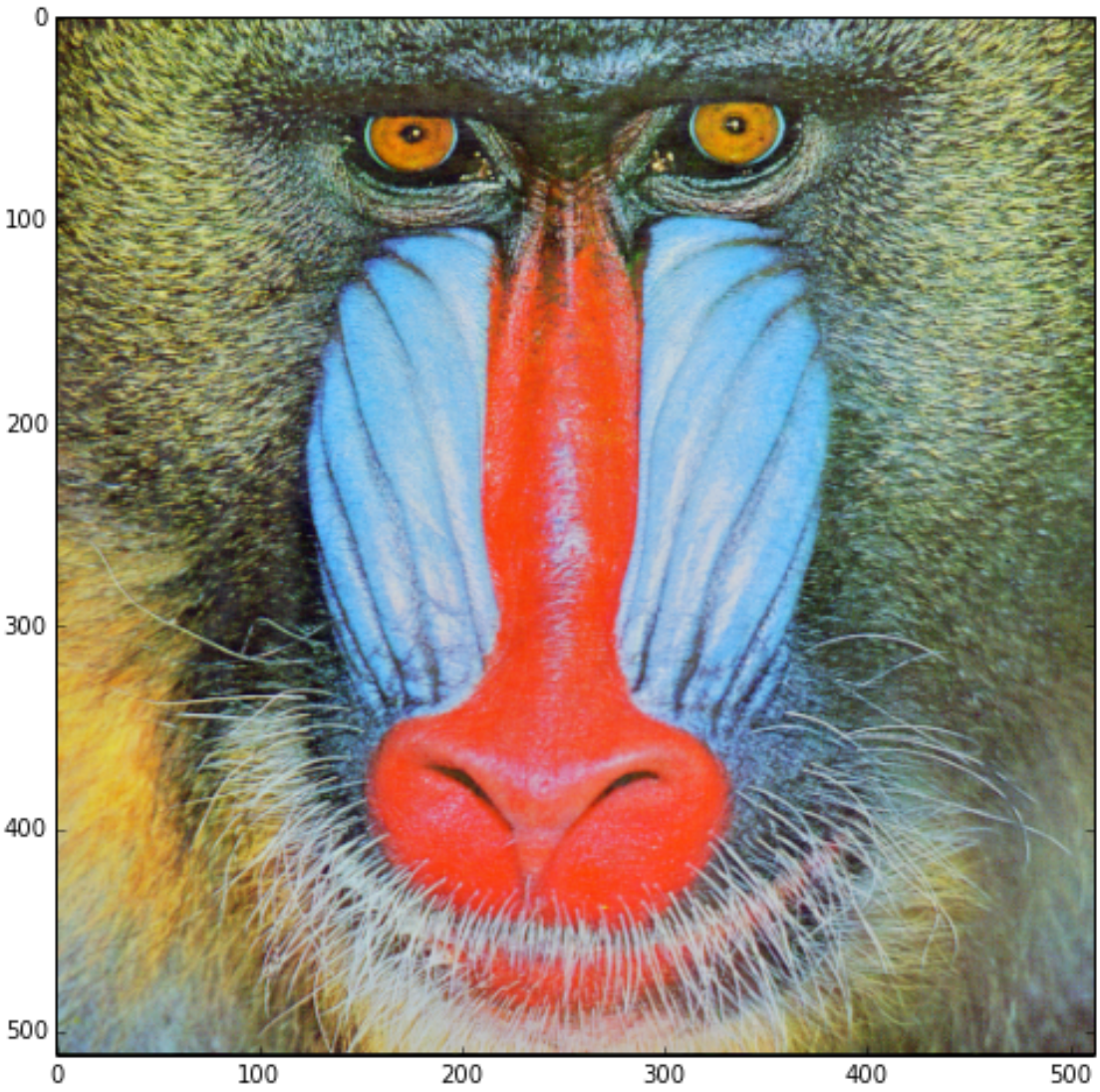
```
In [11]: imshow(mandrill)
```

```
Out[11]: <matplotlib.image.AxesImage at 0x7ffcffa6dc90>
```



```
In [12]: mandrill_rgb = cv2.cvtColor(mandrill, cv2.COLOR_BGR2RGB)
         imshow(mandrill_rgb)
```

```
Out[12]: <matplotlib.image.AxesImage at 0x7ffcfc041f10>
```



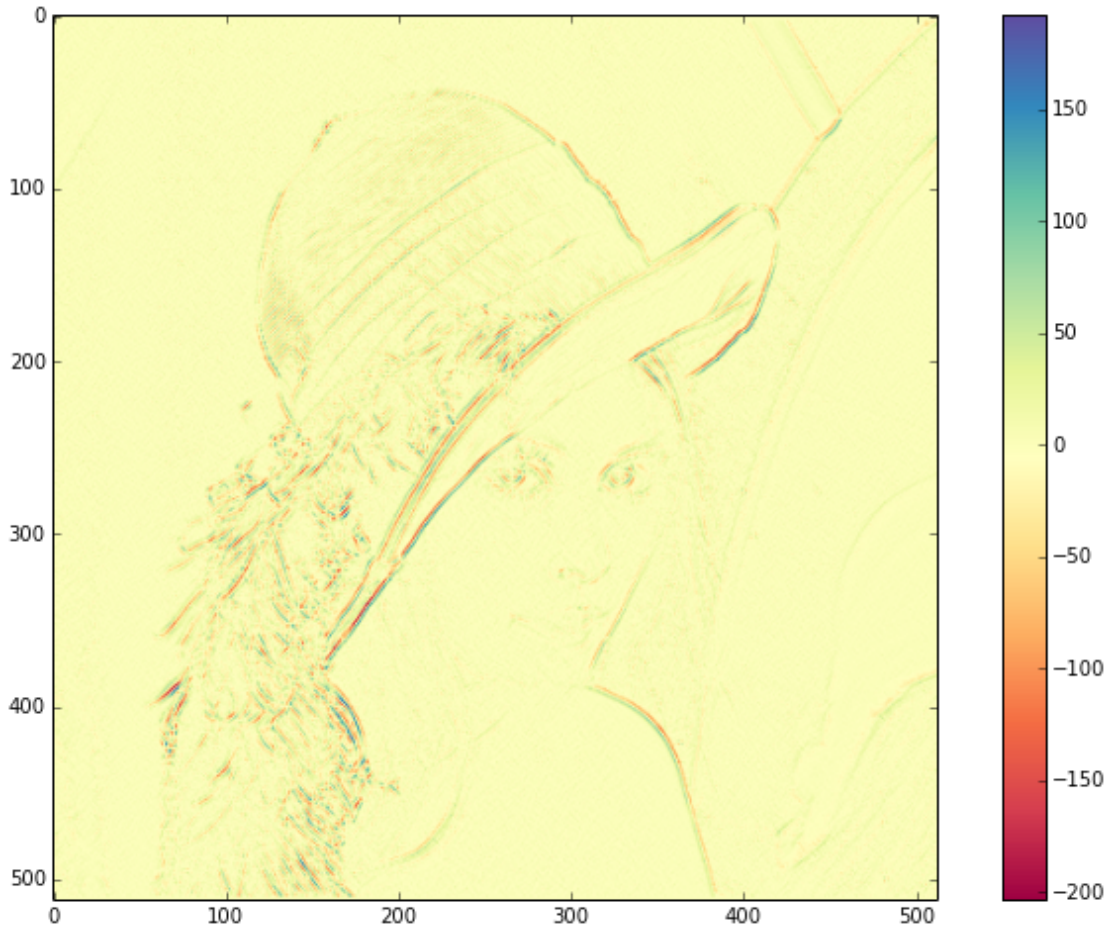
4.1.2 Other data types

- Images are not limited to the *unsigned integer* data type
- Negative integers or real values can result from operators
  - Example: Sobel's convolution
- Thermography
  - Pixels' values can be **floating point** temperatures in °C
- Depth images
  - Pixels's values are distances

Images are not limited to non-negative integer types. Image convolutions can produce negative integers or real values. For example, the Sobel convolution kernel produces negative values representing the derivatives.

```
In [13]: sobel = cv2.Sobel(lenna, cv2.CV_16SC1, 1, 1)
         imshow(sobel, cmap=cm.Spectral)
         colorbar()
```

```
Out[13]: <matplotlib.colorbar.Colorbar instance at 0x7ffcf4042638>
```

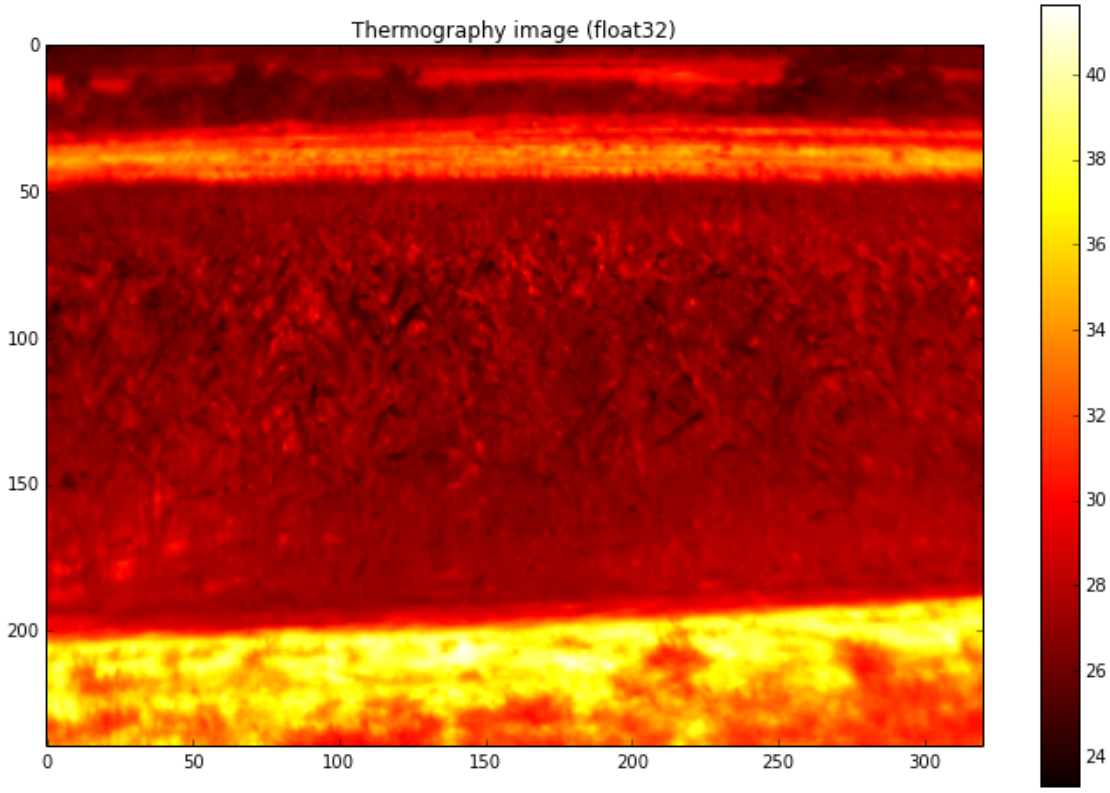



```
In [14]: sobel[100,100]
```

```
Out[14]: -8
```

```
In [15]: title(r'Thermography image (float32)')
         thermo = loadtxt('../data/thermo-maize.csv', delimiter=';')
         imshow(thermo, cmap=cm.hot)
         colorbar()
```

```
Out[15]: <matplotlib.colorbar.Colorbar instance at 0x7ffcebe8c4d0>
```



4.2 Slicing

- **Slicing** can retrieve *parts* of an array
- Employs the convention *start:stop:step*

As seen previously in the *Mandrill* example, array's elements can be indexed using the `[]` operator. Standard Python *slicing* can also be employed to retrieve parts of an array. Slicing employs the convention *start:stop:step*. For example, to retrieve the rows 3 to 9 of an bi-dimensional array A, the code `A[3:10,:]` is used (note *stop* is non-inclusive). In a similar way, to also limit the columns to the range 5 to 8, A[3:10,5:9] is employed. Consider the user is only interested in rows 3, 5, 7 and 9. She could use a step equal to 2, producing `A[3:10:2,5:9]`.

```
In [16]: A = arange(100).reshape(10,10)
         A

Out[16]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
                [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
                [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
                [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
                [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
                [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
                [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
                [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

```
In [17]: A[3:10:2,5:9]
```

```
Out[17]: array([[35, 36, 37, 38],
                [55, 56, 57, 58],
                [75, 76, 77, 78],
                [95, 96, 97, 98]])
```

4.3 Example - Thresholding and fancy indexing

A NumPy array can also be indexed by masks, defined as boolean or integer arrays. This approach is frequently called **fancy indexing**. In this example, a boolean mask is produced applying a logical operation on an array.

```
In [18]: lenna > 128
```

```
Out[18]: array([[ True,  True,  True, ...,  True,  True, False],
                [ True,  True,  True, ...,  True,  True, False],
                [ True,  True,  True, ...,  True,  True, False],
                ...,
                [False, False, False, ..., False, False, False],
                [False, False, False, ..., False, False, False],
                [False, False, False, ..., False, False, False]], dtype=bool)
```

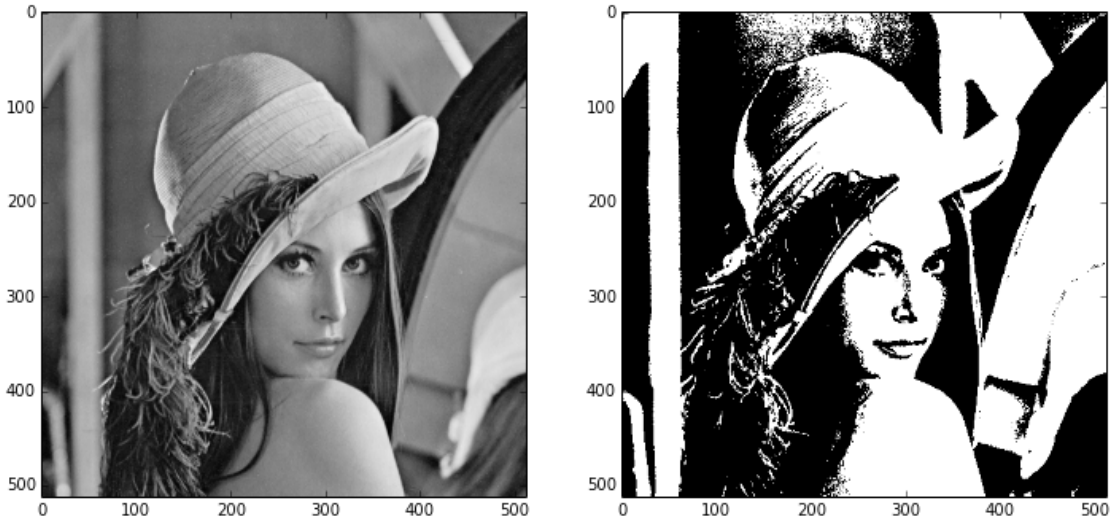
The `zeros_like` function is employed to produce an array presenting the same dimensions of the input *Lenna* image, but all pixels values set to zero. Combined to the attribution operation in the last line, the code produces the matrix `res` defined by:

$$\text{res}[i,j] = \begin{cases} 255 & \text{if lenna}[i,j] > 128, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

```
In [19]: res = zeros_like(lenna)
         res[lenna > 128] = 255
```

```
In [20]: subplot(1,2,1)
         imshow(lenna, cmap=cm.gray)
         subplot(1,2,2)
         imshow(res, cmap=cm.binary_r)
```

```
Out[20]: <matplotlib.image.AxesImage at 0x7ffcebdd1310>
```



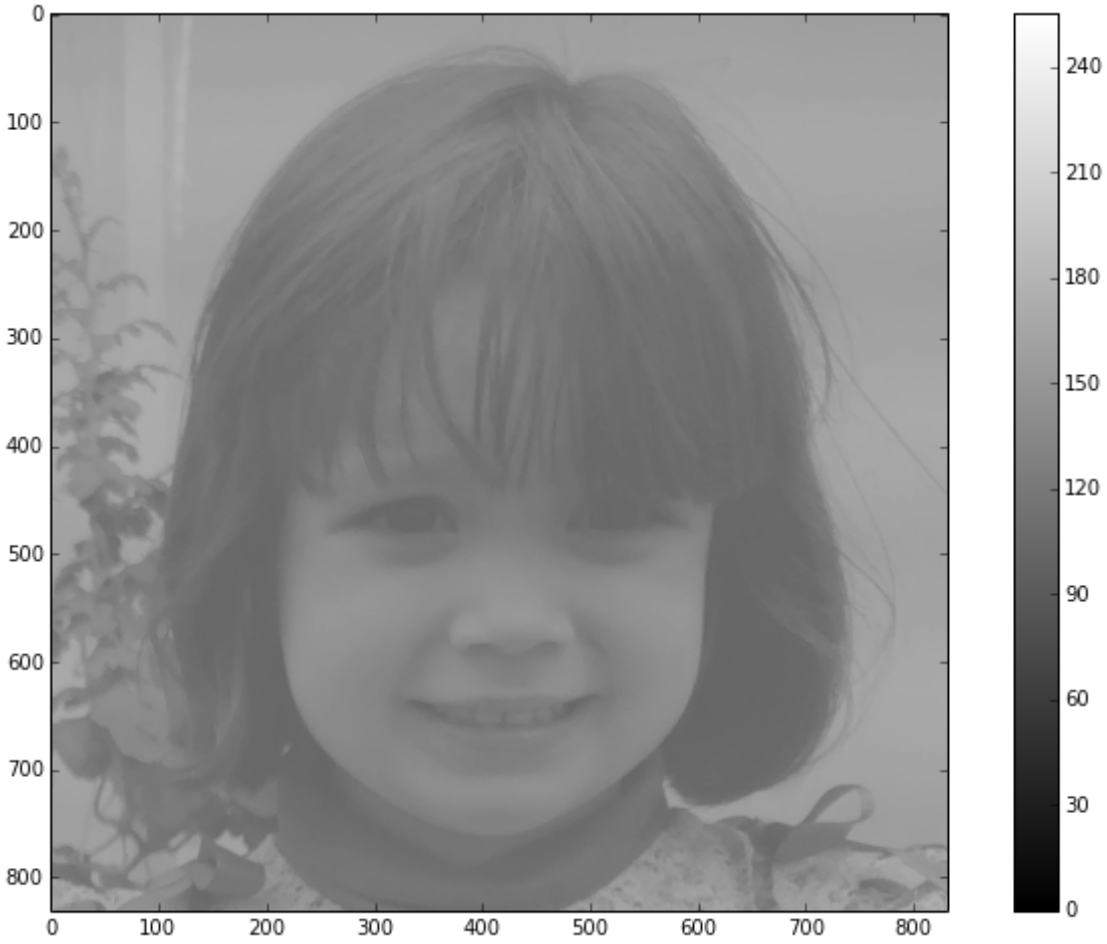
4.4 Example - Whitening

- Problem: *low contrast images*
- Ambient light intensity
- Camera gain

Factors as ambient light intensity or the camera gain produce variations in the image contrast. These factors can be compensated by *whitening*, a per-pixel operation that normalizes the intensity, producing a zero mean image that presents unit variance. This example show as whitening can be efficiently performed by vectorized operations, but keeping the same simplicity of its mathematical definition.

```
In [21]: I = cv2.imread('../data/girl.jpg', cv2.IMREAD_GRAYSCALE)
         imshow(I, cmap=cm.gray, vmin=0, vmax=255)
         colorbar()
```

```
Out[21]: <matplotlib.colorbar.Colorbar instance at 0x7ffcebc20f38>
```



- Solution: **whitening** produces a zero mean, unit variance image

$$W_I[i,j] = \frac{I[i,j] - \mu_I}{\sigma_I}. \tag{2}$$

- NumPy can perform such a *pixel-based* operation efficiently
- **Vectorized operations**

Let μ_I and σ_I be the mean and the standard deviation of a grayscale image I. The whitening operation is defined by:

$$W_I[i,j] = \frac{I[i,j] - \mu_I}{\sigma_I}. \tag{3}$$

```
In [22]: mu_I = mean(I)
         sigma_I = std(I)
         W_I = (I - mu_I)/sigma_I
```

NumPy is able to perform *scalar-array operations*. In the code example above, all elements are subtracted by a scalar, μ_I , and the resulting array is divided by another scalar, σ_I. Below, pixels' values are converted to the [0, 255] range for 8 bits representation.

```
In [23]: delta_W = W_I.max() - W_I.min()
         W_uint8 = array(255./delta_W * (W_I - W_I.min()), dtype=uint8)
         imshow(W_uint8, cmap=cm.gray, vmin=0, vmax=255)
         colorbar()
```

```
Out[23]: <matplotlib.colorbar.Colorbar instance at 0x7ffceba8ba28>
```

4.5 Image ROIs and array views

- In NumPy arrays, regions of interest are named **views**
- An array A and a *view on* A, share the same memory
- Changes in the view produces changes in A

Sometimes, procedures must to be limited to a region of interest (ROI), a rectangular part of the image. In NumPy, the ROI is equivalent to the idea of **view**, an array sharing memory with another one. In the example below, B is *a view on* array A. As expected, changes in B values produce the same change in A.

```
In [24]: A = arange(25).reshape(5,-1)
         A

Out[24]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14],
                [15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24]])

In [25]: B = A[0:3,0:3]
         B

Out[25]: array([[ 0,  1,  2],
                [ 5,  6,  7],
                [10, 11, 12]])

In [26]: B[0,0] = 255
         B

Out[26]: array([[255,   1,   2],
                [  5,   6,   7],
                [ 10,  11,  12]])

In [27]: A

Out[27]: array([[255,   1,   2,   3,   4],
                [  5,   6,   7,   8,   9],
                [ 10,  11,  12,  13,  14],
                [ 15,  16,  17,  18,  19],
                [ 20,  21,  22,  23,  24]])
```

- User can **copy** an array
- The new array will not share memory with the original one

Otherwise, if the A must be preserved of any change in B, a *copy of* A is necessary. In the example below, the `copy` method allocates more memory, data is copied and A and B do not share any memory.

```
In [28]: B = A[0:3,0:3].copy()
         B[0,0] = 128
         B

Out[28]: array([[128,   1,   2],
                [  5,   6,   7],
                [ 10,  11,  12]])

In [29]: A

Out[29]: array([[255,   1,   2,   3,   4],
                [  5,   6,   7,   8,   9],
                [ 10,  11,  12,  13,  14],
                [ 15,  16,  17,  18,  19],
                [ 20,  21,  22,  23,  24]])
```

4.5.1 Reshaping

Consider the UCI ML hand-written digits datasets

```
In [30]: from sklearn import datasets
         digits = datasets.load_digits()
         digits['data'].shape
```

```
Out[30]: (1797, 64)
```

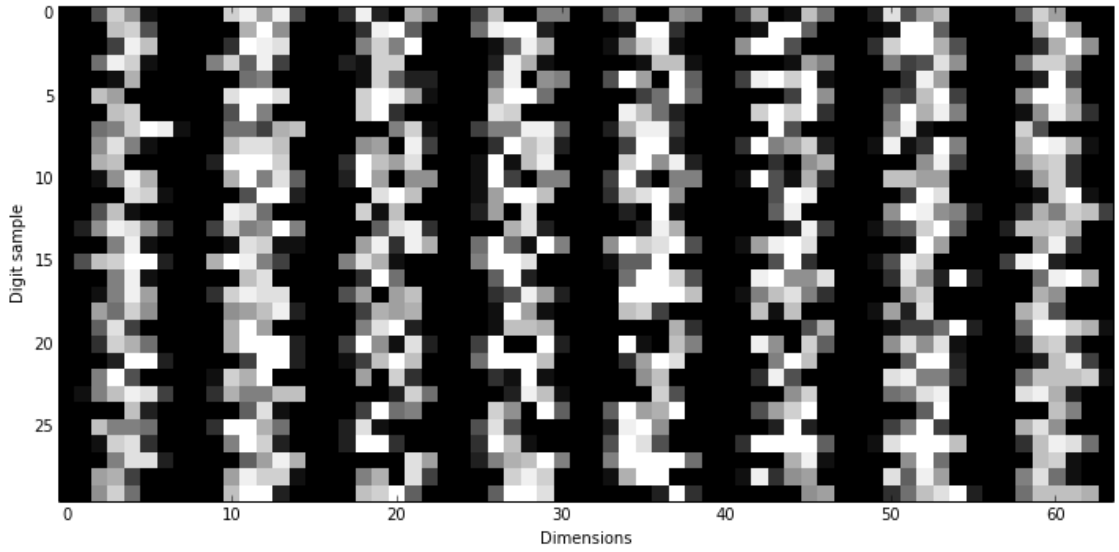
```
In [31]: print digits['DESCR']
```

```
Optical Recognition of Handwritten Digits Data Set

Notes
-----
Data Set Characteristics:
    :Number of Instances: 5620
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
1994.

References
----------
  - C. Kaynak (1995) Methods of Combining Multiple Classifiers and Their
    Applications to Handwritten Digit Recognition, MSc Thesis, Institute of
    Graduate Studies in Science and Engineering, Bogazici University.
  - E. Alpaydin, C. Kaynak (1998) Cascading Classifiers, Kybernetika.
  - Ken Tang and Ponnuthurai N. Suganthan and Xi Yao and A. Kai Qin.
    Linear dimensionalityreduction using relevance weighted LDA. School of
    Electrical and Electronic Engineering Nanyang Technological University.
    2005.
  - Claudio Gentile. A New Approximate Maximal Margin Classification
    Algorithm. NIPS. 2000.
```

```
In [32]: x = digits['data'][0]
         x

Out[32]: array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
                 15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
                  8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
                  5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
                  1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
                  0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

- Each row in the data array is a **64-d feature vector** corresponding to a **handwritten digit**

```
In [33]: imshow(digits['data'][0:30,:], interpolation='nearest', cmap=cm.gray)
         xlabel('Dimensions')
         ylabel('Digit sample')

Out[33]: <matplotlib.text.Text at 0x7ffce7fc6650>
```



- **Reshaping** produces a view on the original array
- The view presents the **same number of elements**, but different dimensions

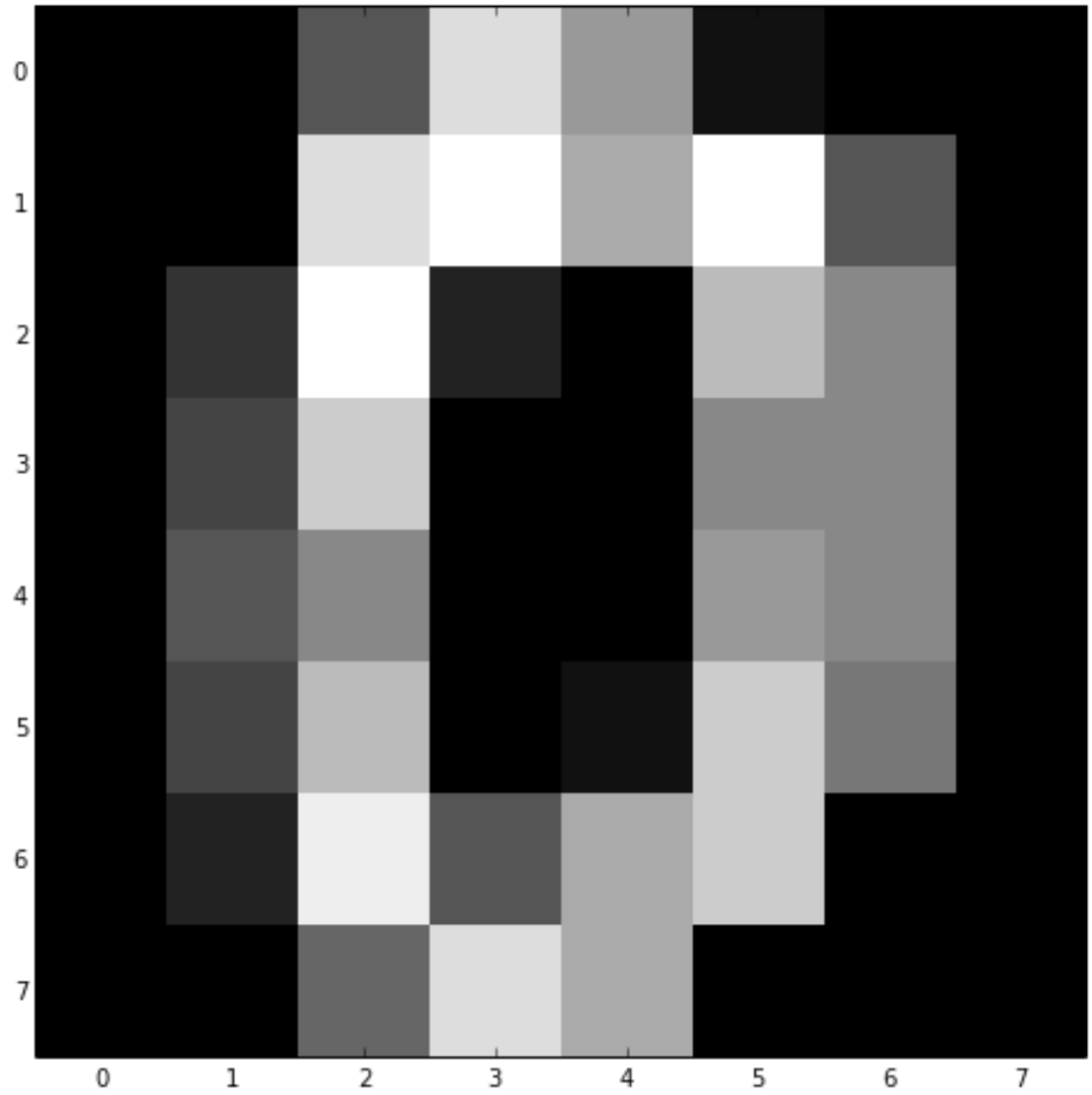
Reshaping is other operation that produces a view on an array. The reshaped array is a view presenting the same number of elements, but different dimensions. As an example, consider the Handwritten Digits Data Set in the UCI Machine Learning Repository. In the handwritten digits classification problem, the $N \times N$ images are usually transformed in N 2-d feature vectors for supervised machine learning. A 64-d feature vector $\mathbf{x}$ can be viewed as a 8×8 image by:

```
In [34]: X = x.reshape(8,8)
         X

Out[34]: array([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],
                [  0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.],
                [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],
                [  0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.],
```

```
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

In [35]:
```
imshow(X, interpolation='nearest', cmap=cm.gray)
```

Out[35]:
```
<matplotlib.image.AxesImage at 0x7ffce7edee10>
```



Similarly, a 8×8 image can be viewed as a 64-d vector using:

In [36]:
```
x = X.reshape(-1)
x
```

Out[36]:
```
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
```

```
       5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
       1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
       0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

```
In [13]: import cv2
         figsize(9,6)
```

5 Matplotlib

5.1 About this notebook

This notebook is part of the tutorial *SciPy and OpenCV as an interactive computing environment for computer vision* to be presented at SIBGRAPI 2014. The tutorial will be also available as a full paper in RITA.

- 2D and 3D publication quality plotting library
- It provides a large set of plotting tools
- Histograms
- Bar plots
- Scatter plots
- Vector fields
- ...

Matplotlib is a 2D plotting aimed to interactive computing and publication-quality image generation. It is a more powerful tool to display images than the standard OpenCV utilities. Interactive zooming, interpolation, automatic scaling and floating point array visualization are Matplotlib features that are not available in OpenCV. It also provides a large set of plotting tools, similar to R and Matlab, including line plots, scatter plots, bar plots, histograms and vector.

Matplotlib's maintainers keep a gallery of plotting examples. Users can pick the desired plot from the gallery and inspect its source code, using it as a template or starting point for their own graphics.

5.2 OpenCV `imshow` vs. Matplotlib `imshow`

- OpenCV `imshow` functions presents very limited capabilities
- It can display 8 bits unsigned integer images
- In general, it's unable to provide visualization for arrays presenting different data types
- Matplotlib `imshow` is a more capable visualization tool
- It is able to visualize arrays of different types
  - Provides different **color mapping** alternatives
- It presents **interactive zooming**
- Automatic scaling for large or small images
  - Different **interpolation methods** available

OpenCV presents an `imshow` function that lets the user display an image in a graphic window. It can show standard 8 bits color or grayscale images, but different data types need adaptions. For 16 bits and 32 bits integers, the pixels values are divided by 256, mapping a $[0, 65280]$ range to $[0, 255]$ before displaying. For 32 bits floating-point images, values are multiplied by 255, what maps the value range $[0, 1]$ to $[0, 255]$. Zooming is not supported and high-resolution images are not be fitted to the computer's screen, making the visualization of large images inconvenient. Remote sensing data or high-resolution photographies usually needs some sort of scaling before displaying.

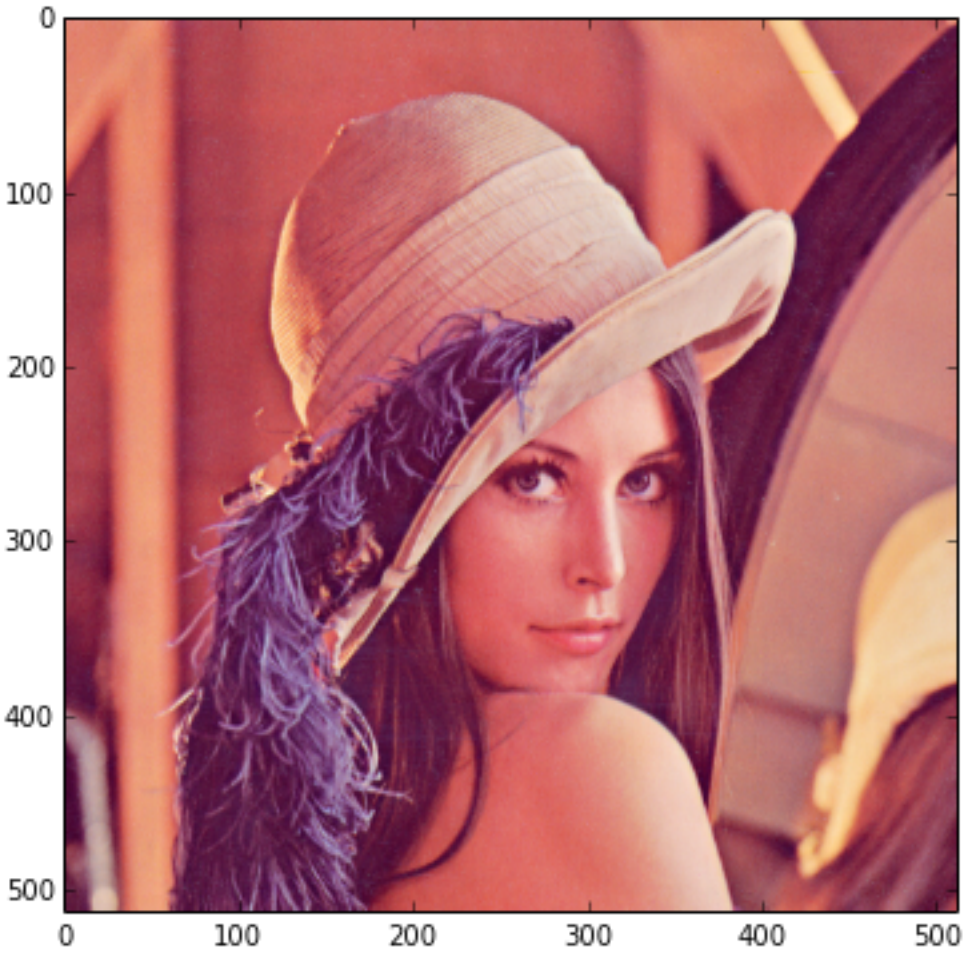
Matplotlib presents its own `imshow` function, able to display NumPy arrays as images. Differently of the OpenCV counterpart, this function is a more capable tool for scientific visualization. Images are fitted to the

plotting window according to different interpolation methods (bilinear, bicubic, sinc and Lanczos, to name just a few). Interactive zooming is available, a convenient feature when the user is inspecting the results of image processing routines. Matplotlib is able to show color images in the form of $M \times N \times 3$ RGB arrays (floating point or integer data types) or $M \times N \times 4$ RGBA arrays presenting an alpha channel. Bidimensional $M \times N$ arrays (as integer or floating point grayscale images) are exhibited using a *colormap* automatically fitted to the array range. The `colorbar` function can be used to plot a bar that illustrates the color/value mapping employed.

5.2.1 Color image visualization

```
In [2]: lenna = cv2.cvtColor(cv2.imread('../data/lenna.tiff'), cv2.COLOR_BGR2RGB)
        imshow(lenna)
```

```
Out[2]: <matplotlib.image.AxesImage at 0x7ff69c051bd0>
```

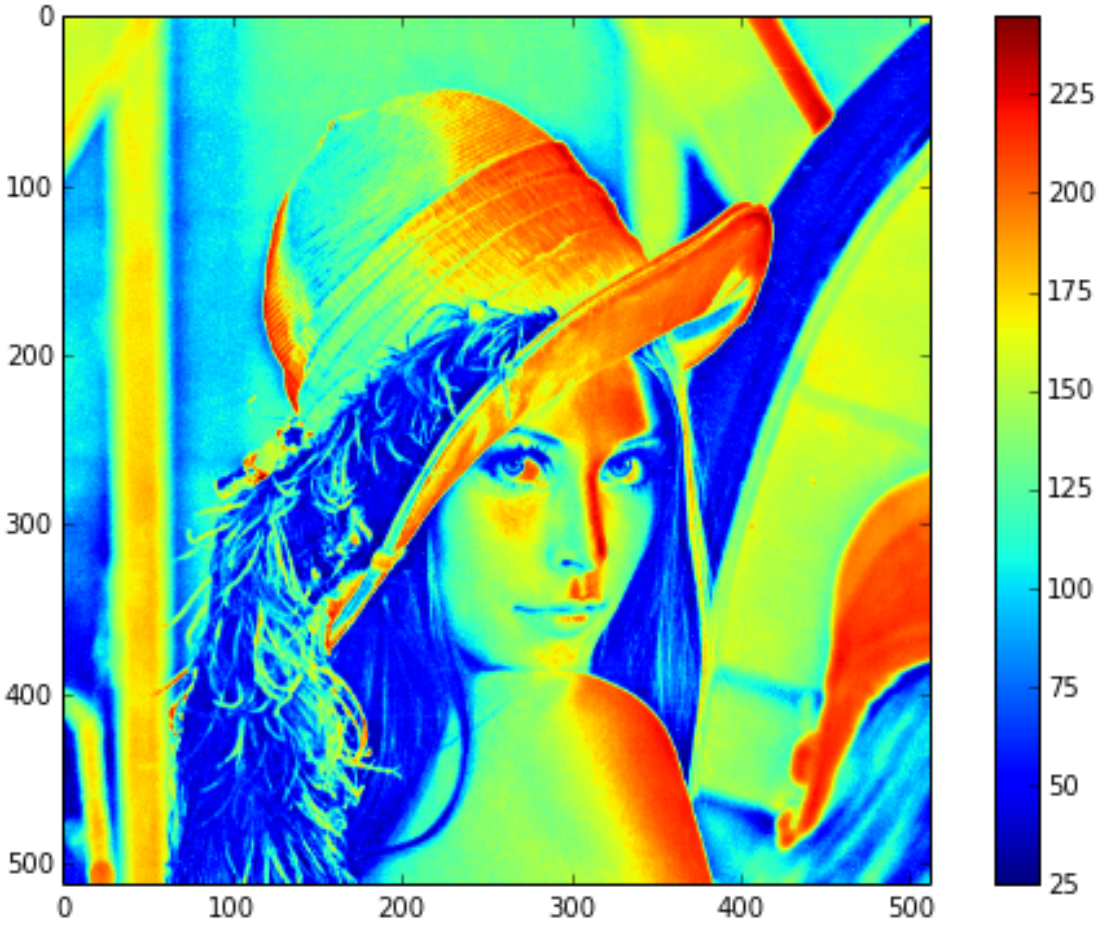


5.2.2 Grayscale or single channel images

Bilinear interpolation, `cm.jet` colormap:

```
In [3]: slenna = cv2.imread('../data/lenna.tiff', cv2.IMREAD_GRAYSCALE)
        imshow(slenna)
        colorbar()
```

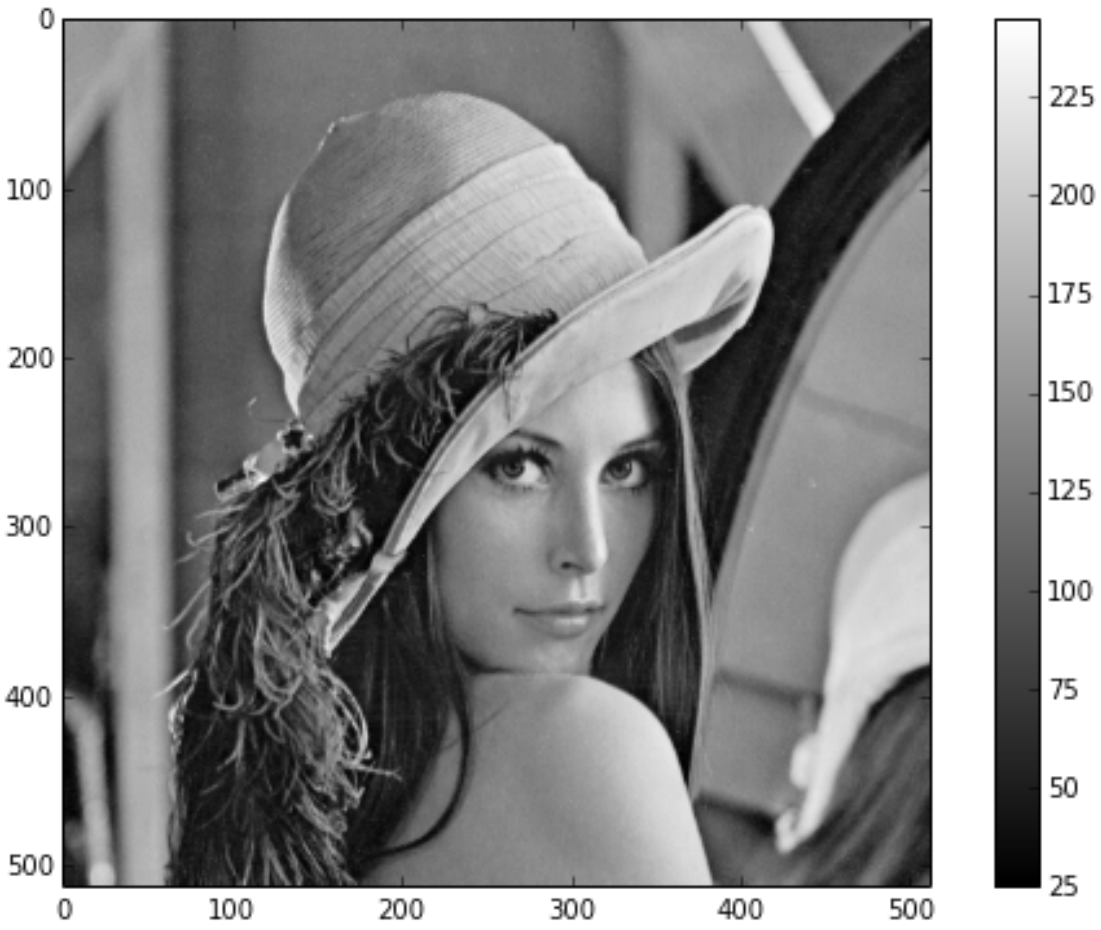
```
Out[3]: <matplotlib.colorbar.Colorbar instance at 0x7ff684068d40>
```



Bilinear interpolation, `cm.jet` colormap:

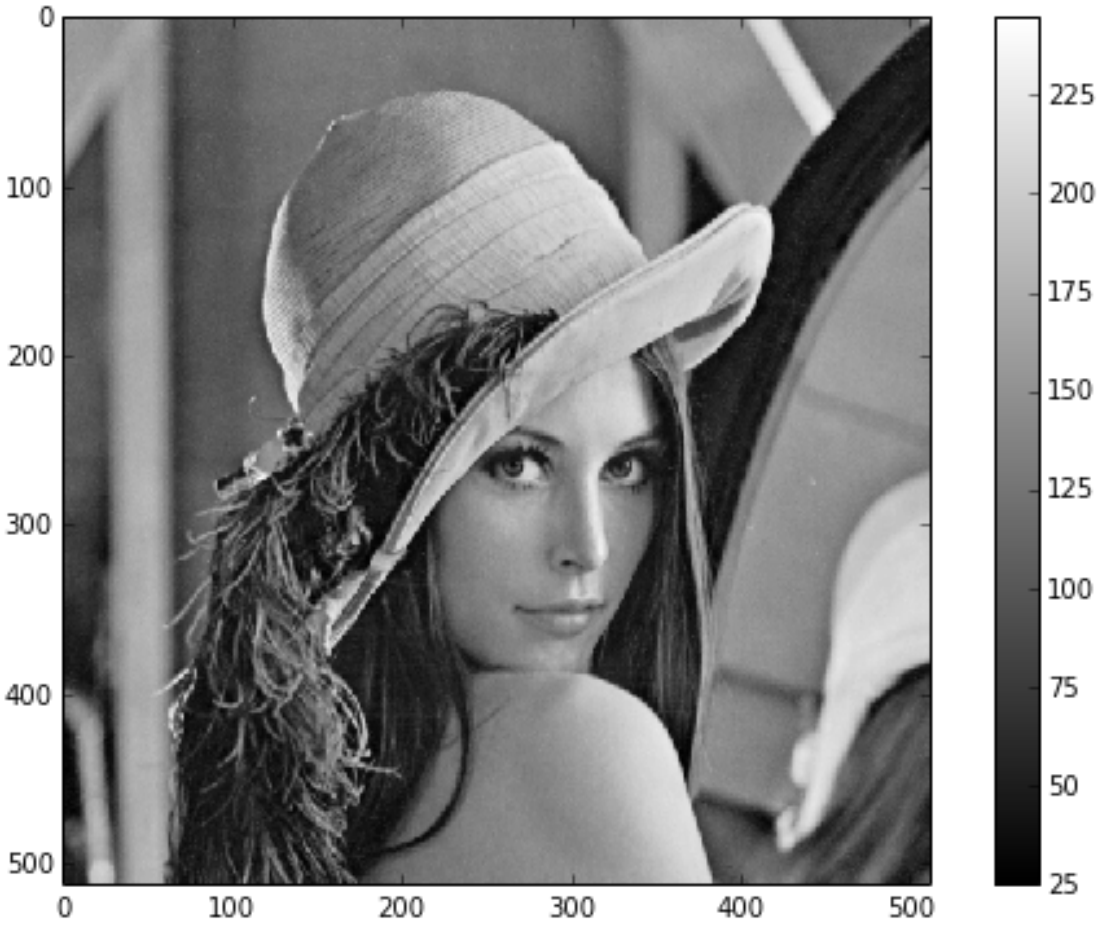
```
In [4]: imshow(slenna, cmap=cm.gray)
        colorbar()
```

```
Out[4]: <matplotlib.colorbar.Colorbar instance at 0x7ff67c48c248>
```



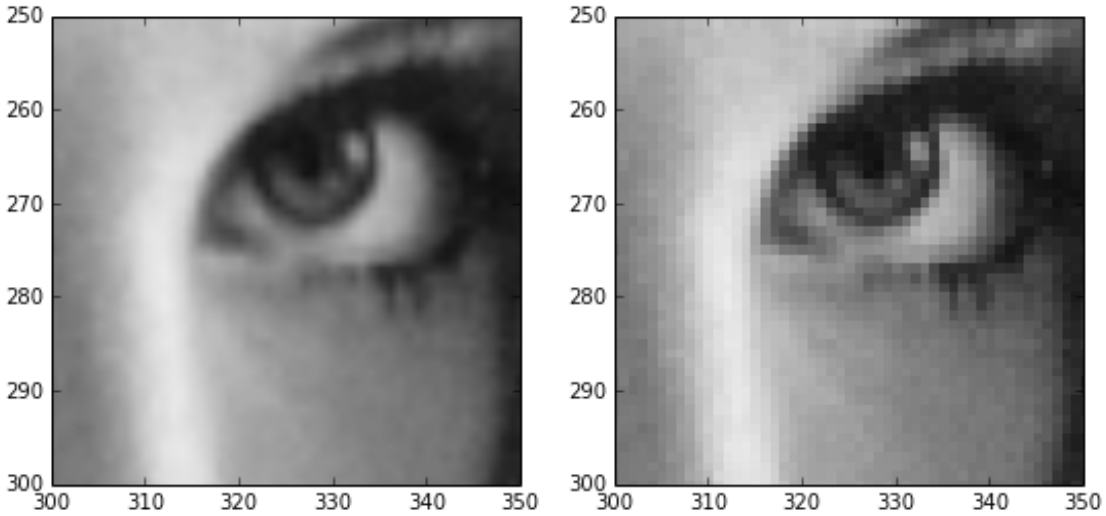
Nearest interpolation, `cm.gray` colormap:

```
In [5]: imshow(slenna, interpolation='nearest', cmap=cm.gray)
        colorbar()

Out[5]: <matplotlib.colorbar.Colorbar instance at 0x7ff67c351908>
```



Bilinear vs. Nearest

```
In [6]: subplot(1,2,1)
        imshow(slenna, cmap=cm.gray)
        xlim(300, 350)
        ylim(300, 250)

        subplot(1,2,2)
        imshow(slenna, interpolation='nearest', cmap=cm.gray)
        xlim(300, 350)
        ylim(300, 250)

Out[6]: (300, 250)
```



6 Example - Showing 2D features

In this example, GFTT, SIFT and SURF features as computed by OpenCV and the results are visualized using Matplotlib `plot` and `scatter` functions.

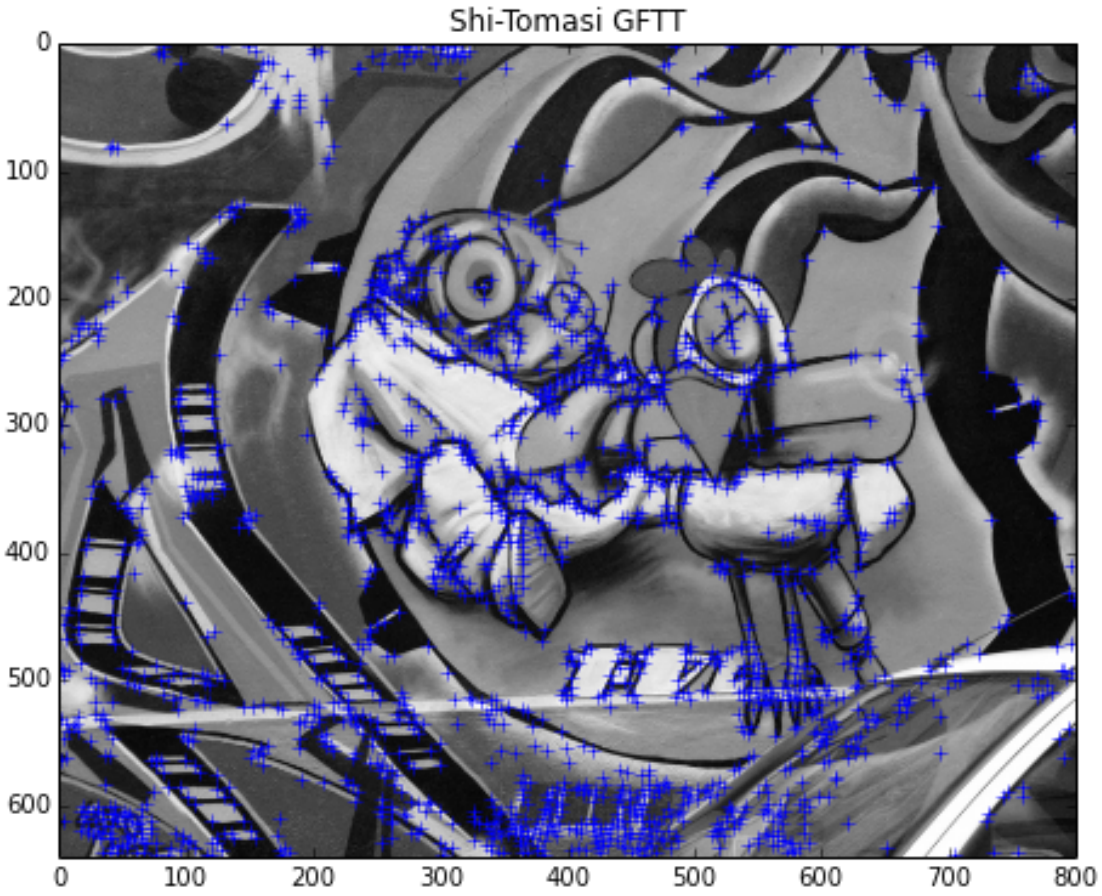
```
In [7]: graffiti = cv2.imread('../data/graffiti-Mikolajczyk.ppm', cv2.IMREAD_GRAYSCALE)
        imshow(graffiti, cmap=cm.gray)
        title('Graffiti image')
```

```
Out[7]: <matplotlib.text.Text at 0x7ff67c1ae150>
```

- GFTT will return up to 2000 features, at least 3 pixels appart
- The `plot` takes list containing the x and y coordinates

The OpenCV wrapper of the GFTT function returns 2D points as a NumPy array. The `plot` function in Matplotlib takes two lists (or arrays) containing respectively the points' x and y coordinates:

```
In [8]: kpts = cv2.goodFeaturesToTrack(graffiti, 2000, 0.01, 3)
        plot(kpts[:,:,0], kpts[:,:,1], 'b+')
        imshow(graffiti, cmap=cm.gray)
        title('Shi-Tomasi GFTT')

Out[8]: <matplotlib.text.Text at 0x7ff67c23b5d0>
```



In the code above, OpenCV's GFTT will return up to 2000 corners (features), all of them at least 3 pixels appart. The 0.01 is a quality level parameter defined over the eigen values (see OpenCV's documentation for details). The features are returned as a $2000 \times 1 \times 2$ array and slicing is used in the plotting function to recover the lists of x and y coordinates. The `b+` argument informs Matplotlib the points have to be plotted as blue crosses. Finally, imshow is employed to display the image.

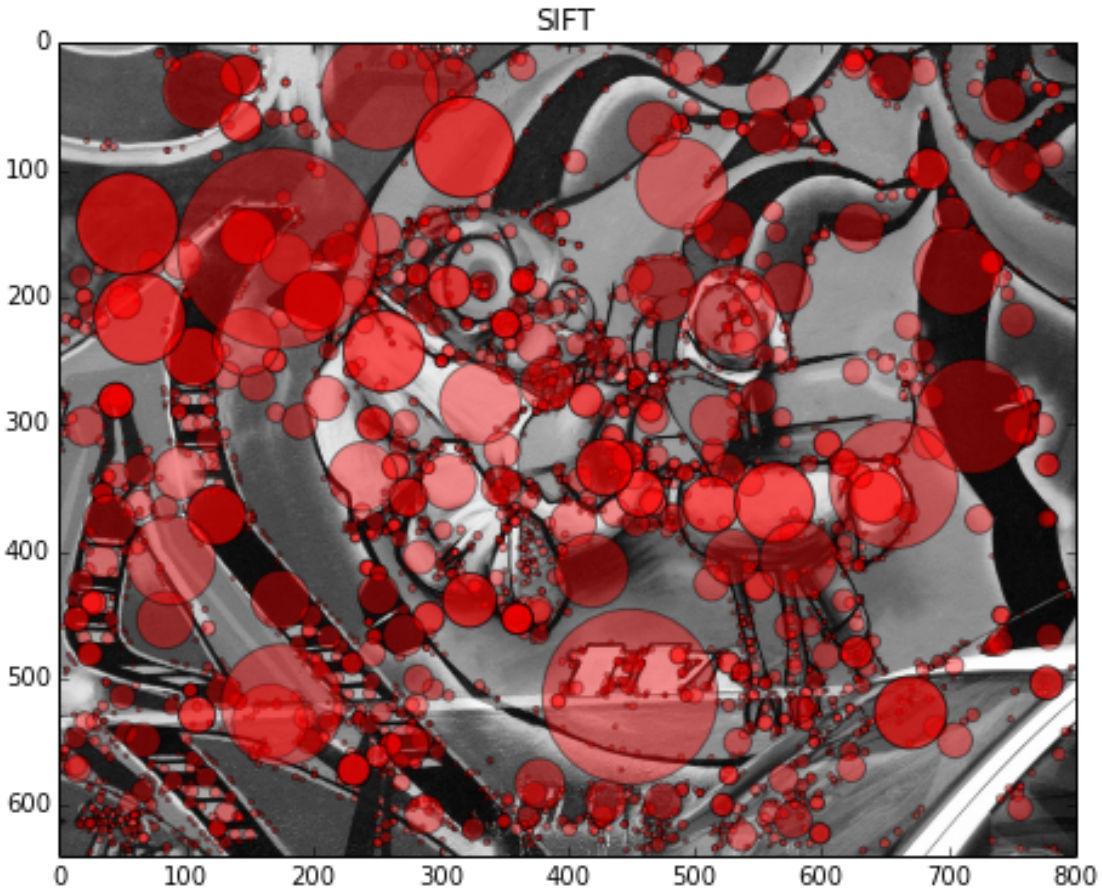
- Features like SIFT and SURF are not "dimensionless"
- They are **multi-scale features** defined over a *neighborhood*
- OpenCV's `Keypoint` structure represent these features
- `Keypoint.pt` keeps the feature location
- `Keypoint.size` keeps the diameter of the meaningful neighborhood

Differently of the GFTT features, SIFT and SURF are not dimensionless. They are multi-scale features and are defined over neighborhoods presenting different sizes. OpenCV represents these features using the `KeyPoint` data structure: the feature coordinates are stored in the `pt` variable and the neighborhood diameter in the `size` variable. The scatter function is able to take a list of sizes and plot the points as circular regions:

```
In [9]: sift = cv2.SIFT()
        kpts = sift.detect(graffiti)
        x = [k.pt[0] for k in kpts]
        y = [k.pt[1] for k in kpts]
        # s will correspond to the neighborhood area
        s = [(k.size/2)**2 * pi for k in kpts]
```

```
scatter(x, y, s, c='r', alpha=0.4)
imshow(graffiti, cmap=cm.gray)
title('SIFT')
```

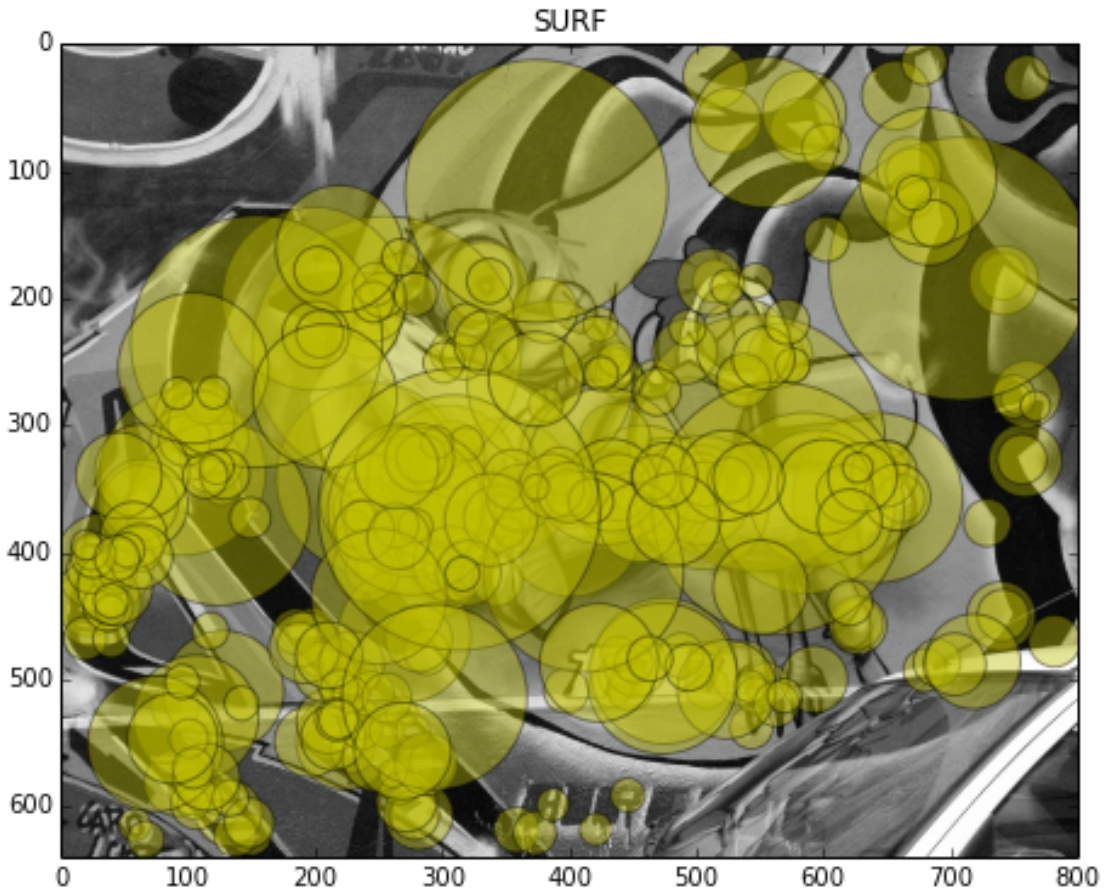
```
Out[9]: <matplotlib.text.Text at 0x7ff67c282490>
```



In the code above, the `alpha` parameter is employed to plot using a 40% transparency, allowing the viewer to see the image below the circles, and the `r` parameter asks for a plotting in *red*.

```
In [19]: surf = cv2.SURF(hessianThreshold=7000)
         kpts = surf.detect(graffiti)
         x = [k.pt[0] for k in kpts]
         y = [k.pt[1] for k in kpts]
         s = [(k.size/2)**2 * pi for k in kpts]
         scatter(x, y, s, c='y', alpha=0.4)
         imshow(graffiti, cmap=cm.gray)
         title('SURF')
```

```
Out[19]: <matplotlib.text.Text at 0x7ff67759a0d0>
```



7 The SciPy Library

- The SciPy Library includes modules for
  - optimization,
  - interpolation,
  - signal processing,
  - linear algebra,
  - sparse matrix representation,
  - KD-Trees, Delaunay triangulation, convex hull computation
  - statistics, and others

The SciPy library includes modules for optimization, interpolation, signal processing, linear algebra, statistics, sparse matrix representation and operation, among others. It also includes the `spatial` module, containing an implementations for the KD-Tree data structure (used on nearest-neighbor queries), Delaunay triangulation and convex hull computation.

7.1 Linear algebra using `scipy.linalg`

- Linear algebra is particularly important to **projective geometry**
- Consider a 3D point $\mathbf{X}_i$

- Let `P` be a **camera matrix**
- The projection of $\mathbf{X}_i$ on the camera plane is $\mathbf{x}_i = \mathtt{P}\mathbf{X}_i$

Linear algebra is an essential mathematical tool for computer vision and machine learning. It is particularly important in problems involving projective geometry as in multiple view computer vision. Consider a 3D point $\mathbf{X}_i$, represented as a *homogeneous* 4-d array, and a projective matrix `P`, represented as a 3×4 array and corresponding to a camera. The projection of $\mathbf{X}_i$ on the camera's image plane, $\mathbf{x}_i$, can be elegantly coded as:

```
In [3]: P = array([[  1.41598501e+03,  -3.53656018e+01,   6.29880366e+02,  -9.80199681e+02],
                   [ -1.66130187e+01,   1.52666051e+03,   2.41551888e+02,   2.34435820e+02],
                   [ -2.20274424e-01,   4.97475710e-03,   9.75425256e-01,   6.34871255e-01]])

        X = array([-0.11652141, -0.24440939,  3.23347243,  1.])

        x = dot(P, X)
        x

Out[3]: array([ 900.15223192,  644.29279864,    3.81333274])
```

The `dot` function, for 2-D arrays, computes the matrix multiplication, and for 1-D arrays it calculates the inner product of vectors.

The pixel inhomogeneous coordinates can be recovered using:

```
In [5]: x_coord = x[0]/x[2]
        y_coord = x[1]/x[2]
        x_coord, y_coord

Out[5]: (236.05394389057571, 168.95792816629523)
```

7.1.1 Singular value decomposition

- Several problems in multiple view geometry involves **over-determined systems of equations**
  - Homography estimation
  - 3D points triangulation
- $A\mathbf{x}$ is homogeneous systems of linear equations
- Can be solved using **Singular Value Decomposition (SVD)**

Estimation problems in projective geometry involve the solution of over-determined systems of equations. More precisely, these problems are formalized as *linear least-squares* resolutions of homogeneous systems of linear equations in the form $A\mathbf{x}$, minimizing $A\mathbf{x}$ subject to $\|\mathbf{x}\| = 1$.

The minimization problem can be solved using **singular value decomposition (SVD)**, a matrix decomposition particularly useful in numerical computations. SVD decomposes A as $A = UDV^\intercal$ and the $\mathbf{x}$ solution corresponds to the last columns of V.

```
In [8]: def dlt_triangulation(ui, Pi, uj, Pj):
            """Hartley & Zisserman, 12.2"""
            ui /= ui[2]
            xi, yi = ui[0], ui[1]

            uj /= uj[2]
            xj, yj = uj[0], uj[1]

            a0 = xi * Pi[2,:] - Pi[0,:]
            a1 = yi * Pi[2,:] - Pi[1,:]
```

```
    a2 = xj * Pj[2,:] - Pj[0,:]
    a3 = yj * Pj[2,:] - Pj[1,:]

    A = vstack((a0, a1, a2, a3))
    U, s, VT = linalg.svd(A)
    V = VT.T

    X3d = V[:,-1]

    return X3d/X3d[3]
```

Examples of `spatial.KDTree` and `linalg.svd` will be presented in the Structure from Motion example.

8 Scikit-learn

```
In [1]: import cv2
        figsize(12,8)
```

- **Machine Learning** (ML) is important for computer vision
  - Learning
  - Inference
- **Scikit-learn** is a module for machine learning algorithms
  - Supervised learning
  - Unsupervised learning
  - Dimensionality reduction
  - Parameter selection
  - Cross-validation
- Development leadership by researchers at INRIA, France

The computer vision field relies strongly on machine learning methods and Bayesian inference. Machine learning provides the learning and inference tools for fitting and predicting the world state from images in several vision problems. The scikit-learn toolbox (or sklearn) is a machine learning package built on the SciPy Stack, developed by an international community of practitioners under the leadership of a team of researchers in INRIA, France. It provides tools for regression, classification, clustering, dimensionality reduction, parameter selection and cross-validation. Gaussian mixture models, decision trees, support vector machines, and Gaussian processes are a few examples of the methods available to date.

8.1 Fit/Predict

- Scikit-learn's objects implements a **fit/predict interface**
- `fit`
  - learning step (supervised or unsupervised)
- `predict`
  - regression or classification
- The learned **model** can be **stored** using Python's built-in persistence model, `pickle`

Sklearn is able to evaluate an estimator's performance and parameters by cross-validation, optionally distributing the computation to several computer cores if necessary. The sklearn module implements machine learning algorithms as objects that provide a fit/predict interface. The fit method performs learning (supervised or unsupervised, according to the algorithm). The predict method performs regression or classification. The learned model can be saved for further usage by pickle, the Python's built-in persistence model.

8.2 Supervised learning in Sklearn

- Nearest Neighbors
- Support Vector Machines (SVM)
  - Linear Support
  - Radial Basis Function (RGB) kernel SVM
- Decision Trees
- Ensemble
  - Random Forests
  - AdaBoost
- Linear Discriminant Analysis
- Gaussian Processes

8.3 Unsupervised learning in Sklearn

- Gaussian mixture models
- Clustering
  - Affinity propagation
  - Mean-shift
  - Spectral clustering
  - Hierarchical clustering
  - DBSCAN
- Neural Networks (unsupervised)
  - Restricted Boltzmann machines

This tutorial will not provide a full view of all methods available in sklearn. Instead, the basic usage will be illustrated by three examples on Naïve Bayes classification, mean-shift clustering and Gaussian Mixture models. For a broad and in-depth view on this module, the reader is referred to the sklearn online documentation, what is rich in descriptions, tutorials and code examples. Readers interest in machine learning and its applications in vision should refer to Bishop's and Prince's books.

8.4 Example - Skin detection using Naïve Bayes

In this example, Naïve Bayes classification is employed to detect pixels corresponding to human skin in images, based just in the pixel's color measurements.

8.4.1 Training data

- A $M \times N \times 3$ array, **color a image in CIE Lab space**
- A $M \times N$ **binary mask** representing reference classification
  - Supervised learning
- L channel is discarded, avoiding lightness influence on skin detection

Let training be a $M \times N \times 3$ array representing a color training image in the CIE Lab color space, and mask a $M \times N$ binary array representing the manual classification skin/non-skin. The Gaussian fitting for Naïve Bayes classification will just use the chromaticity data (channels 1 and 2), avoiding lightness to influence on skin detection.

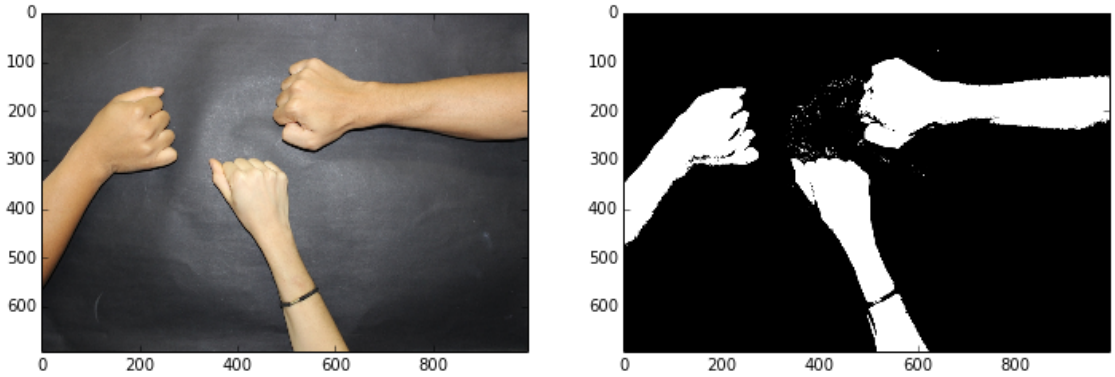
```
In [2]: training_bgr = cv2.imread('../data/skin-training.jpg')
        training_rgb = cv2.cvtColor(training_bgr, cv2.COLOR_BGR2RGB)
        training = cv2.cvtColor(training_bgr, cv2.COLOR_BGR2LAB)
        M, N, _ = training.shape
```

- The training image provides skin samples on a dark background
- Thresholding is employed to produce the binary mask

```
In [3]: mask = zeros((M,N))
        mask[training[:,:,0] > 160] = 1
```

```
In [4]: subplot(1,2,1)
        imshow(training_rgb)
        subplot(1,2,2)
        imshow(mask, cmap=cm.binary_r)
```

```
Out[4]: <matplotlib.image.AxesImage at 0x7fcd4ac194d0>
```



- Data is *reshaped* to a MN vectors
- Each vector is 2-d, containing values for a anf b channels
- *Slicing* used to skip the L channel data

The data is composed by MN 2d-vectors, easily extracted from the training image using reshaping and slicing.

```
In [5]: data = training.reshape(M*N, -1)[:,1:]
        data
```

```
Out[5]: array([[128, 129],
               [128, 129],
               [128, 129],
               ...,
               [127, 129],
               [125, 134],
               [123, 136]], dtype=uint8)
```

Similarly, the manual classification used in the learning step is represented as a binary MN vector:

```
In [6]: target = mask.reshape(M*N)
        target
```

```
Out[6]: array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

Training (fitting)

- Gaussian Naïve Bayes is implemented by the `GaussianNB` object
- It presents a `fit` method for training

Sklearn provides a `naive_bayes` module containing a `GaussianNB` object that implements the supervised learning by the Gaussian Naïve Bayes method. As previously discussed, this object presents a `fit` method that performs the learning step:

```
In [7]: from sklearn.naive_bayes import GaussianNB
        gnb = GaussianNB()
        gnb.fit(data, target)
```

```
Out[7]: GaussianNB()
```

Classification (prediction)

- Input image is converted to the CIE *Lab* color space
- Array is reshaped and sliced as the training data
- `GaussianNB.predict` is used for classification

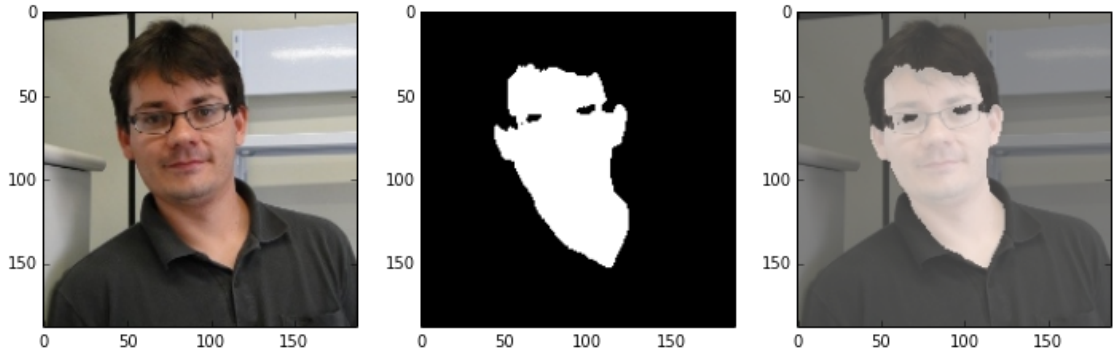
Skin detection can be performed converting the input image to the Lab color space, and then reshaping and slicing in the same way as the training image. The predict method of GaussianNB performs the classification. The resulting classification vector can be reshaped to the original image dimensions for visualization.

```
In [8]: test_bgr = cv2.imread('../data/thiago.jpg')
        test_rgb = cv2.cvtColor(test_bgr, cv2.COLOR_BGR2RGB)
        test = cv2.cvtColor(test_bgr, cv2.COLOR_BGR2LAB)
        M_tst, N_tst, _ = test.shape
```

```
In [9]: data = test.reshape(M_tst * N_tst, -1)[:,1:]
        skin_pred = gnb.predict(data)
        S = skin_pred.reshape(M_tst, N_tst)
```

```
In [10]: subplot(1,3,1)
         imshow(test_rgb)
         subplot(1,3,2)
         imshow(S, cmap=cm.binary_r)
         subplot(1,3,3)
         imshow(test_rgb, alpha=0.6)
         imshow(S, cmap=cm.binary_r, alpha=0.4)
```

```
Out[10]: <matplotlib.image.AxesImage at 0x7fcd360b3150>
```



8.5 Example - Color segmentation using mean-shift clustering

In this example, the mean-shift algorithm is employed to perform color segmentation, grouping similar colors together (*color quantization*).

8.5.1 Mean-shift

- The *feature vectors* are the pixels' **color triplets** in CIE *Lab* color space
- A $M \times N \times$ array is reshaped to MN 3-d vectors

This clustering procedure relies in the Euclidean distance between the feature vectors, in this case the pixels' color triplets. A perceptually uniform color space is more suitable to this task, once in such a space the Euclidean distances between triplets approximate the human perceptual differences. In this example, the *Lab* space is employed again. A view on the image is produced by reshaping, transforming the $M \times N$ array in a sequence of MN 3-d vectors:

```
In [11]: I = cv2.imread('../data/BSD-118035.jpg')
         I_Lab = cv2.cvtColor(I, cv2.COLOR_BGR2LAB)
         h, w, _ = I_Lab.shape
         from sklearn.cluster import MeanShift, estimate_bandwidth
         X = I_Lab.reshape(h*w, -1)
         X

Out[11]: array([[ 33, 121, 120],
                [ 33, 121, 120],
                [ 33, 121, 120],
                ...,
                [122, 122, 118],
                [125, 122, 120],
                [ 38, 122, 126]], dtype=uint8)
```

- Means-shift implementation in sklearn employs a **flat kernel**
- Such a kernel is defined by a **bandwidth** parameter
- Bandwidth can be automatically selected
  - Sampling of inter-pixels color distances
    - Euclidean distance in *Lab* approximates human perception
  - A *quantile* is selected to pick the bandwidth value

The mean-shift implementation in sklearn employs a flat kernel defined by a *bandwidth* parameter. The bandwidth can be automatically selected by sampling the color distances be tween pixels in the input image and taking an arbitrary quantile selected by the user (larger quantiles generate bandwidths that produce fewer clusters). This procedure is implemented by the `estimate_bandwidth` function. Finally, the `fit` method is employed to perform the unsupervised learning:

```
In [12]: b = estimate_bandwidth(X, quantile=0.1, n_samples=2500)
         ms = MeanShift(bandwidth=b, bin_seeding=True)
         ms.fit(X)

/usr/local/lib/python2.7/dist-packages/sklearn/cluster/mean_shift_.py:45: NeighborsWarning: kneighbors:
  d, _ = nbrs.kneighbors(X, return_distance=True)

Out[12]: MeanShift(bandwidth=12.296007814340841, bin_seeding=True, cluster_all=True,
              seeds=None)
```

`bin_seeding=True` initializes the kernel locations to discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth.

- `ms.labels_` keeps the cluster identification for each pixel
- `ms.cluster_centers_` stores the *cluster centers*
- The color quantization is performed attributing to each pixel the value the assigned cluster center

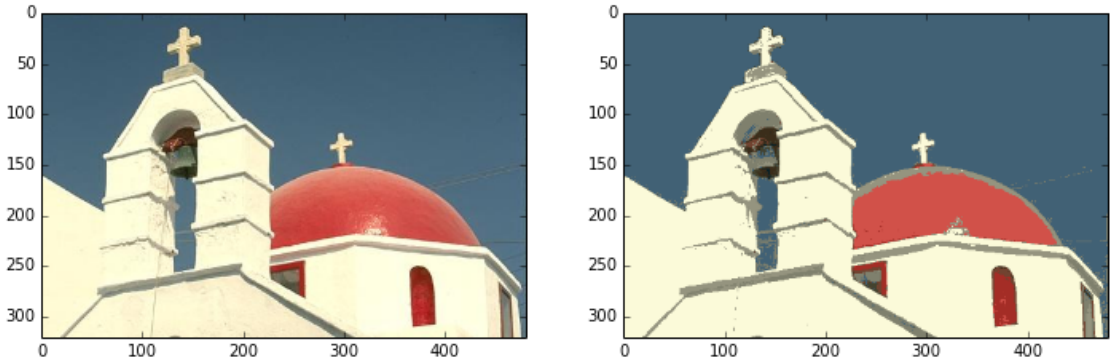
The `labels_` attribute keeps the cluster attributed to each pixel, and the `cluster_centers_` attribute stores the center value for each cluster. These centers are the quantized colors and will be employed on the visualization:

```
In [13]: S = zeros_like(I)
         L = ms.labels_.reshape(h, w)
         num_clusters = ms.cluster_centers_.shape[0]
         print num_clusters

         for c in range(num_clusters):
             S[L == c] = ms.cluster_centers_[c]
```

```
10
```

```
In [14]: subplot(1,2,1)
         imshow(cv2.cvtColor(I, cv2.COLOR_BGR2RGB))
         subplot(1,2,2)
         imshow(cv2.cvtColor(S, cv2.COLOR_LAB2RGB))
```

```
Out[14]: <matplotlib.image.AxesImage at 0x7fcd2a3a1d50>
```



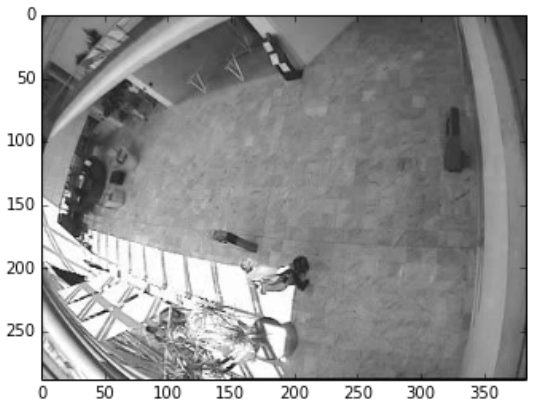
In this example, just the pixel color was employed, resulting in color quantization. To perform spatial-color segmentation as proposed by Comaniciu and Meer, a multivariate kernel is needed. To the date, multivariate kernels are not available in sklearn's mean-shift implementation.

8.6 Example - Background subtraction using Gaussian mixture models

The background of a video sequence is modeled using mixtures of Gaussians and further employed to classify people and objects as foreground.

Let V be a $T \times MN \times 3$ array representing a video sequence composed by T frames. Each frame is a $M \times N$ color image (pixels' values are color triplets). The background model is composed by M N mixtures of K multivariate Gaussians. Stauffer and Grimson proposed the use of Gaussian mixtures for background modeling because they are a simple and convenient way to represent multimodal distributions. Scenes in video sequences can present some sort of dynamic background, an issue commonly referred as the "waving trees" problem, and multimodal distributions are a better way to represent this variation.

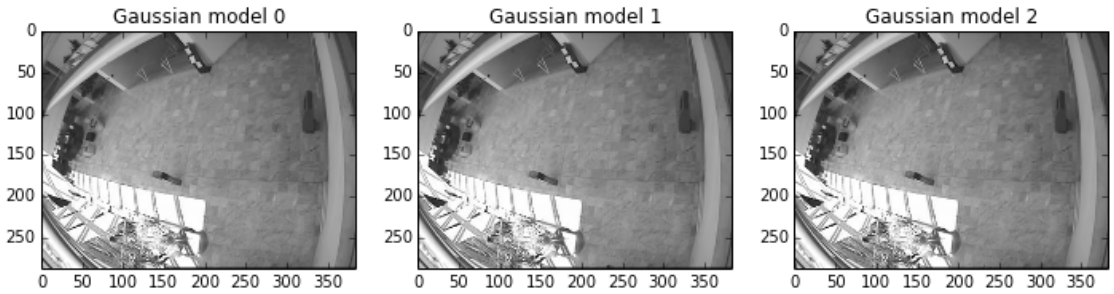
```
In [15]: frames = !ls ../data/CAVIAR_LeftBag/*.jpg
         # Let's find the frame dimensions, M x N
         F = cv2.imread(frames[0])
         M, N, _ = F.shape
         T = len(frames)
         M, N, T

Out[15]: (288, 384, 1439)

In [16]: def normalize_frame(F):
             R = array(F[:,:,0], dtype=float)
             G = array(F[:,:,1], dtype=float)
             B = array(F[:,:,2], dtype=float)
             S = sum(F, axis=2, dtype=float) + 1e-5

             r = R/S
             g = G/S
             Fmin = minimum(R, G, B)
             Fmax = maximum(R, G, B)
             L = (Fmin + Fmax)/2
             L /= 255

             return dstack((L, r, g))

In [17]: V = zeros((T, M*N, 3))
         for t, fname in enumerate(frames):
             F = cv2.cvtColor(cv2.imread(fname), cv2.COLOR_BGR2RGB)
             Fn = normalize_frame(F)
             V[t] = Fn.reshape(M*N, -1)

In [18]: subplot(1,2,1)
         L_0 = V[0].reshape(M, N, -1)[:,:,0]
         imshow(L_0, cmap=cm.gray)

         subplot(1,2,2)
         L_100 = V[100].reshape(M, N, -1)[:,:,0]
         imshow(L_100, cmap=cm.gray)

Out[18]: <matplotlib.image.AxesImage at 0x7fcd890d4b10>
```



Each one of the MN pixels is represented by a Gaussian mixture model (GMM). In the code below, Python's list comprehension is used to instantiate MN `GMM` objects and immediately perform the fitting. Such a code works because the fit method returns the calling `GMM` object. Three Gaussian are being used ($K = 3$), letting each model represent up to 3 modes in a background distribution. The learning is performed using frames V t constrained to the instants t $\in$ [600, 749], a segment of the input video presenting an empty scene without people or moving objects.

After fitting, the Gaussians' means and weights can be recovered:

```
In [19]: from sklearn.mixture import GMM
         K = 3
         fmodel = [GMM(n_components=K).fit(V[600:750,i]) for i in range(M*N)]

In [20]: bg_mean = array([gmm.means_ for gmm in fmodel])
         bg_weight = array([gmm.weights_ for gmm in fmodel])

         for k in range(K):

             subplot(1, K, k+1)
             mu = bg_mean[:,k,:].reshape(M, N, -1)
             imshow(mu[:,:,0], cmap=cm.gray)
             title('Gaussian model %d' % k)
```



For classification of a frame V_t , the predict method should be called by the GMM object of each pixel:

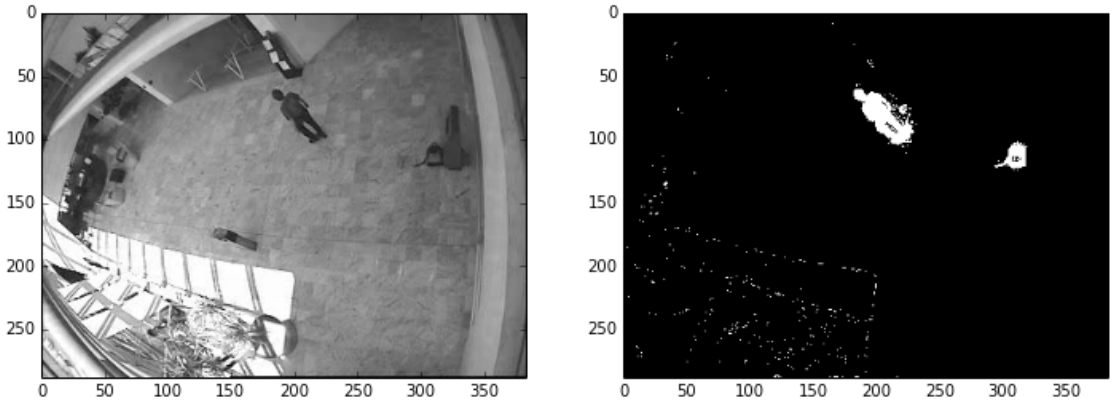
```
In [21]: t = 1000
         # Each the right Gaussian model for each pixel i
         c = array([fmodel[i].predict([V[t,i]]) for i in range(M*N)])
         pmean = array([fmodel[i].means_[c[i]] for i in range(M*N)]).reshape(M*N, 3)
         pcov = array([fmodel[i].covars_[c[i]] for i in range(M*N)]).reshape(M*N, 3)
```

Here, the prediction step just selects one of the K Gaussians, c_i , as the proper model to the i-th pixel. Note the code above also recovers the mean and the covariance matrix of c_i . The pixel is declared background if (i) the weight of of c_i is above a threshold selected by the user (or defined using supervised learning) and (ii) if the difference between the observed pixel and the mean of c_i is under a confidence interval, defined using the covariance matrix in c_i. The criteria in (i) evaluates if such a Gaussian is really modeling the background distribution or if it just captured moving objects or noise.

```
In [22]: wstmt = array([bg_weight[i, c[i]] for i in range(M*N)]) > 0.2
         wstmt = wstmt.reshape(-1)
         stmt = abs(V[t] - pmean) < 2. * sqrt(pcov)
         background = stmt[:,0] & stmt[:,1] & stmt[:,2] & wstmt
```

```
In [23]: subplot(1,2,1)
         imshow(V[t].reshape(M, N, -1)[:,:,0], cmap=cm.gray)

         subplot(1,2,2)
         imshow(background.reshape(M, N), cmap=cm.binary)
```

```
Out[23]: <matplotlib.image.AxesImage at 0x7fcc43fd32d0>
```




```
In [2]: import cv2
```

9 Performance, HPC and IPython Parallel

9.1 Performance issues

- Python looping in big arrays can be **slow**
- However
  - OpenCV operations are efficient machine code
  - NumPy operations on arrays are efficient machine code
  - SciPy Stack relies in C, C++ and Fortran implementations for numerical software

Low-level code written in Python, as looping in big arrays, can be slow, mainly because Python is dynamically typed and interpreted. However, in the scientific computing environment described above, this is rarely a problem: the OpenCV interface just access optimized C/C++ code, and most of the software in the SciPy Stack relies in a base of numerical software implemented in C, C++ and Fortran, including the efficient NumPy arrays.

9.1.1 Cython

- Cython is a **static compiler**
- It works on a **super-set of the Python language** that supports **C-like static type declarations**
- Compiles Python code to C
  - Produces a module that can be imported by the Python interpreter
- Useful to
  - speed-up low-level looping in arrays;
  - access external C/C++ libraries

- **Pareto Principle**
  - 80% of the run-time is spent in 20% of the source code

But in the few situations where a low-level looping must be implemented (if the task cannot be implemented using NumPy capabilities) or the functionality of a external library is needed, Cython raises as an alternative. Cython is a static compiler capable of working in a super-set of the Python language that supports C-like static type declarations. It compiles Python code to C, further producing a Python module that can be imported and used from the interpreter. As noted by Behnel *et al.*, the key idea behind Cython is the **Pareto Principle**, also known as the "80/20 rule": 80% of the run-time is spent in 20% of the source code. Cython's goal is to speed up the critical parts of the code while avoiding too much overhead on coding by the programmer.

9.2 IPython.parallel

- Other performance issues can be addressed by **parallelization**
- IPython.parallel allows parallel and distributed computing
  - Single Program, Multiple Data (SPMD)
  - Multiple Program, Multiple Data (MPMD).
- Parallel applications can be developed, executed and monitored from the IPython shell
- Computer vision tasks can involve large sets of images or big point clouds
- However, the parallelization of these tasks is trivial
- In IPython.parallel, those tasks can be implemented in a few lines of code

Other performance issues can be addressed by parallelization. IPython.parallel is a powerful architecture for parallel and distributed computing supporting different styles of parallelism, such as single program, multiple data (SPMD) and multiple program, multiple data (MPMD). Parallel applications can be easily developed, executed and monitored interactively from the IPython shell. Computer vision tasks can involve large sets of images or big point clouds, but many times the parallelization of these tasks is trivial and, using IPython.parallel, implemented in a few lines of code. The dynamic load balancing feature allows the use of all the available processing threads in the computer or all the processing power available in a cluster, but keeping the interactive computing environment free from large amounts of specific code for parallel computing.

9.3 Example - Process a bundle of images in parallel

In this example, SIFT descriptors of a reference image I_1 are computed. Then, descriptors are extracted for every image I_n in a list, and the matches to I_1 descriptors are computed. The processing of the list is done in parallel, using all the available cores in the user's machine.

Let D_1 be an array containing the descriptors of I_1.

```
In [37]: T1 = cv2.imread('/tmp/templeRing/templeR0001.png', cv2.IMREAD_GRAYSCALE)
         sift = cv2.SIFT(nfeatures=5000)
         _, D_1 = sift.detectAndCompute(T1, mask=None)
```

In a system shell, an IPython cluster for parallel computing is started using:

```
ipcluster start --n=8
```

Eight *nodes* are started (in this example, the number of clusters is selected based on the number of cores available in the user's machine).

Back to the IPython shell, the next step is the creation of a `Client` object. A `LoadBalancedView` object is created to provide a load-balanced parallel execution:

```
In [34]: from IPython.parallel import Client
         rc = Client()
         lview = rc.load_balanced_view()
```

- The decorator `@lview.parallel` defines a **parallel, load-balanced** function
- The arguments are:
  - The image file absolute path in the filesystem
  - The reference descriptor set, D_1
- `get_num_matches` will:
  - read the image;
  - compute SIFT features and their descriptors;
  - perform matching using OpenCV's *brute force matching* `BFMatcher` and
  - return the number of matches found

Next, a Python *decorator* is used to define a parallel function that computes the descriptors and the matches (the decorator starts with a "`@`" symbol). The function below takes a path to an image in the file system, computes the SIFT features and uses OpenCV's `BFMatcher` to get the matches to D_1, returning the number of matches found and the image's path:

```
In [35]: @lview.parallel()
         def get_num_matches(arg):
             fname, D_src = arg
             import cv2
             frame = cv2.imread(fname, cv2.IMREAD_GRAYSCALE)
             print frame.shape
             sift = cv2.SIFT(nfeatures=5000)
             _, D = sift.detectAndCompute(frame, mask=None)
             matcher = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
             matches = matcher.match(D_src, D)
             return fname, len(matches)
```

- File paths and D_1 are assembled in an **arguments list**
- The `map` function starts the parallelized call
- Load balancing is automatically performed

IPython capability to access the system's shell is employed to list all the files in a directory and store the file paths in a list of strings, `fnames`. Finally, the `map` function calls `get_num_matches` to every string in the `fnames` list, automatically performing the load balance on the nodes:

```
In [36]: fnames = !ls /tmp/templeRing/temple*.png
         args = [(fname, D_1) for fname in fnames]
         async_res = get_num_matches.map(args)

In [33]: for f, n in async_res:
             print f, n

/tmp/templeRing/templeR0001.png 802
/tmp/templeRing/templeR0002.png 549
/tmp/templeRing/templeR0003.png 491
/tmp/templeRing/templeR0004.png 482
/tmp/templeRing/templeR0005.png 470
/tmp/templeRing/templeR0006.png 441
/tmp/templeRing/templeR0007.png 401
/tmp/templeRing/templeR0008.png 358
```

```
/tmp/templeRing/templeR0009.png 393
/tmp/templeRing/templeR0010.png 438
/tmp/templeRing/templeR0011.png 456
/tmp/templeRing/templeR0012.png 455
/tmp/templeRing/templeR0013.png 444
/tmp/templeRing/templeR0014.png 405
/tmp/templeRing/templeR0015.png 436
/tmp/templeRing/templeR0016.png 421
/tmp/templeRing/templeR0017.png 410
/tmp/templeRing/templeR0018.png 398
/tmp/templeRing/templeR0019.png 408
/tmp/templeRing/templeR0020.png 451
/tmp/templeRing/templeR0021.png 430
/tmp/templeRing/templeR0022.png 444
/tmp/templeRing/templeR0023.png 454
/tmp/templeRing/templeR0024.png 476
/tmp/templeRing/templeR0025.png 475
/tmp/templeRing/templeR0026.png 509
/tmp/templeRing/templeR0027.png 509
/tmp/templeRing/templeR0028.png 546
/tmp/templeRing/templeR0029.png 558
/tmp/templeRing/templeR0030.png 662
/tmp/templeRing/templeR0031.png 577
/tmp/templeRing/templeR0032.png 465
/tmp/templeRing/templeR0033.png 477
/tmp/templeRing/templeR0034.png 479
/tmp/templeRing/templeR0035.png 457
/tmp/templeRing/templeR0036.png 456
/tmp/templeRing/templeR0037.png 473
/tmp/templeRing/templeR0038.png 477
/tmp/templeRing/templeR0039.png 473
/tmp/templeRing/templeR0040.png 454
/tmp/templeRing/templeR0041.png 461
/tmp/templeRing/templeR0042.png 431
/tmp/templeRing/templeR0043.png 442
/tmp/templeRing/templeR0044.png 454
/tmp/templeRing/templeR0045.png 448
/tmp/templeRing/templeR0046.png 454
/tmp/templeRing/templeR0047.png 459
```

This simple example is able to explore all the available cores in the local machine, just asking for a few extra lines of code. But the parallel computing capabilities in IPython go far beyond, supporting SPMD and MPMD parallelism and the use of StarCluster for execution in Amazon's Elastic Compute Cloud (EC_2). The interested reader is referred to the section Using IPython for parallel computing in the IPython documentation.

10 A structure from motion example

```
In [1]: import cv2
        figsize(12,8)
```

The last example in this tutorial combines different packages to solve a structure from motion problem on 2 images. It starts using OpenCV to detect SIFT features and their descriptors. Scikit-learn's implementation of PCA is employed to reduce the dimensionality of the descriptors and the KD-Trees in the SciPy library are

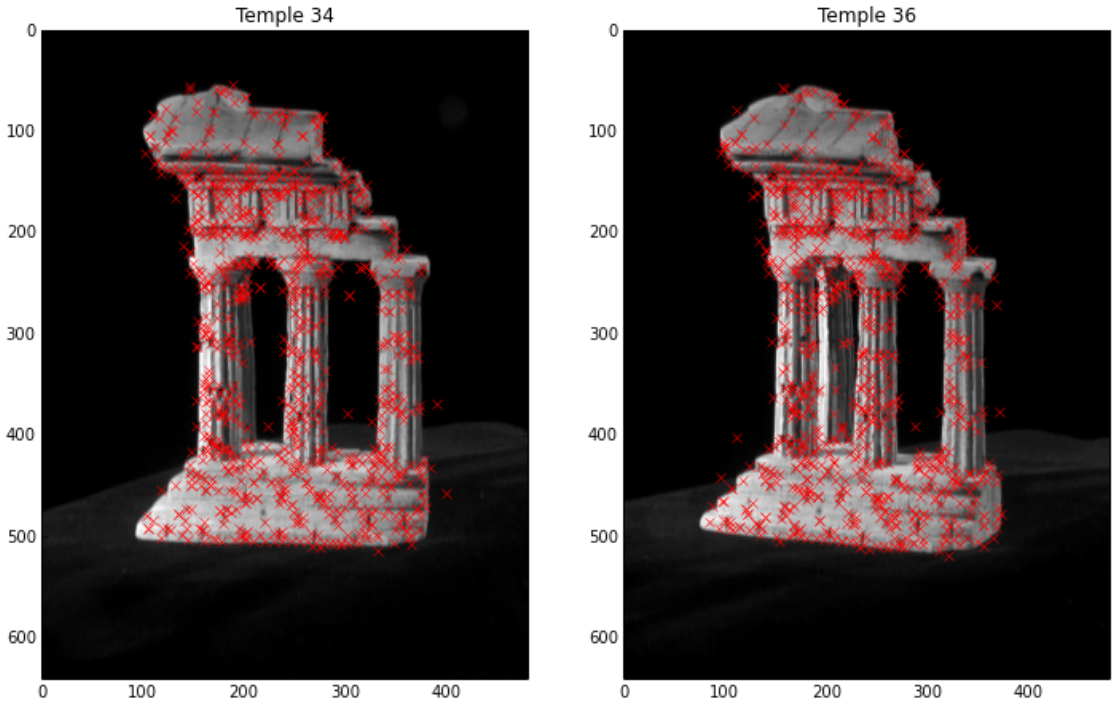
then used to efficiently find features matches. From the found matches, the fundamental matrix between the pair of images is computed using OpenCV, and linear algebra procedures from SciPy are employed to compute the essential matrix and retrieve valid projection matrices. Finally, SVD decomposition, implemented in scipy.linalg, is used to perform triangulations and estimate a 3D point for each pair of matching features. The code and comments are available online 11 as an IPython notebook.

- Take two images input images from a static object
- Compute features and matches between the images
- Recover the fundamental and the essential matrices `F` and `E`
- Extract a pair of projective matrices from `E`
- Triangulate to get the 3D points $\mathbf{X}_i$

10.1 Input: the *Temple Ring* dataset

- The Temple Ring dataset was created by Seitz *et al.*
- We will load **two images**
- And detect SIFT features and their descriptors

```
In [2]: T1 = cv2.imread('../data/templeRing/templeR0034.png', cv2.IMREAD_GRAYSCALE).T
        sift = cv2.SIFT(nfeatures=5000)
        kpts1, D_i = sift.detectAndCompute(T1, mask=None)
        K1 = array([[k.pt[0], k.pt[1]] for k in kpts1])

In [3]: T2 = cv2.imread('../data/templeRing/templeR0036.png', cv2.IMREAD_GRAYSCALE).T
        sift = cv2.SIFT(nfeatures=5000)
        kpts2, D_j = sift.detectAndCompute(T2, mask=None)
        K2 = array([[k.pt[0], k.pt[1]] for k in kpts2])

In [4]: subplot(1,2,1)
        plot(K1[:,0], K1[:,1], 'rx')
        imshow(T1, cmap=cm.gray)
        title('Temple 34')

        subplot(1,2,2)
        plot(K2[:,0], K2[:,1], 'rx')
        imshow(T2, cmap=cm.gray)
        title('Temple 36')

Out[4]: <matplotlib.text.Text at 0x2a719d0>
```



10.2 Computing features matches

- Matching
  - $d_i \in \mathbf{D}_i \rightarrow d_j \in \mathbf{D}_j$
  - Matching is a **nearest neighbor problem**
- In a **low dimensional space**, can be efficiently performed using **KD-Trees**
- However, SIFT descriptors are **128-d** vectors
- Alternatives
  - Approximate Nearest Neighbors (as FLANN library)
  - Brute force matching
  - **Dimensionality reduction**

Consider the two sets of descriptors, $\mathbf{D}_i$ and $\mathbf{D}_j$, computed using a method as SIFT or SURF, for two images I_i and I_j. Matching can be performed using nearest neighbors, just selecting for each $\mathbf{d}_i \in \mathbf{D}_i$ the closest vector $\mathbf{d}_j \in \mathbf{D}_j$. Nearest neighbor queries can be efficiently done representing $\mathbf{D}_i$ in a KD-Tree.

KD-Tree performance is close to brute force for vectors presenting large dimensions. SIFT vectors are 128-d and SURF ones are 64-d. Before matching using KD-Trees, a dimensionality reduction procedure, as PCA, is recommended. Sklearn and OpenCV provide PCA implementations.

```
In [5]: from sklearn.decomposition import PCA
        pca = PCA(n_components=10)

        pca.fit(D_i)
        D_i = pca.transform(D_i)
        D_j = pca.transform(D_j)
```

- Lowe recommends to compare the **two nearest neighbors**

- In a **good match**, there is a contrast between the two distances
  - Descriptors with no proper match present similar distances between their closest neighbors
- Filtering
  - $d_j \in \mathbf{D}_j$ should be assigned to just **one** $d_i \in \mathbf{D}_i$

Lowe observes that many features will not have any correct match so just picking the closest neighbor would produce many incorrect matches. The recommended method is to compare the distance of the nearest neighbor to that of the second-closest one. If the ratio between the two distances is below a threshold, the matching is accepted. The rationale behind this procedure is that features with no proper matching would present similar distances between to their closest neighbors.

A KD-Tree is created from the N_j vectors in the array $\mathbf{D}_j$. The `query` procedure takes all vectors in $\mathbf{D}_j$ and the number of closest neighbors to be retrieved, just two in this case ($k = 2$). The function return two $N_i \times 2$ arrays, `d` and `nn`, keeping the distances and the neighbors indexes respectively. That means `d[n][0]` keeps the distance between the n-th vector in $\mathbf{D}_i$ and its closest neighbor, that is the element `nn[n][0]` in $\mathbf{D}_j$. Fancy indexing is used to select the matches that obey the ratio test proposed by Lowe. Finally, a $N_i \times 2$ array is created, keeping in each row the indexes n_i and n_j such that n_i-th descriptor in $\mathbf{D}_i$ matches the n_j-th descriptor in $\mathbf{D}_j$. This array `m` is produced using the `vstack`, a function that "stack" arrays vertically, in this case a row of indexes in $\mathbf{D}_i$ followed by the corresponding matching indexes in $\mathbf{D}_j$.

Other procedure to reject bad matches is to ensure that the same feature $\mathbf{d}_j \in \mathbf{D}_j$ is the proper match of a single feature $\mathbf{d}_i \in \mathbf{D}_i$.

In the code below, `h` is a Python dictionary acting as an histogram, counting the number of times the n_j-th descriptor of $\mathbf{D}_j$ was matched to a descriptor in $\mathbf{D}_i$. The last line just keeps the matches that are unique.

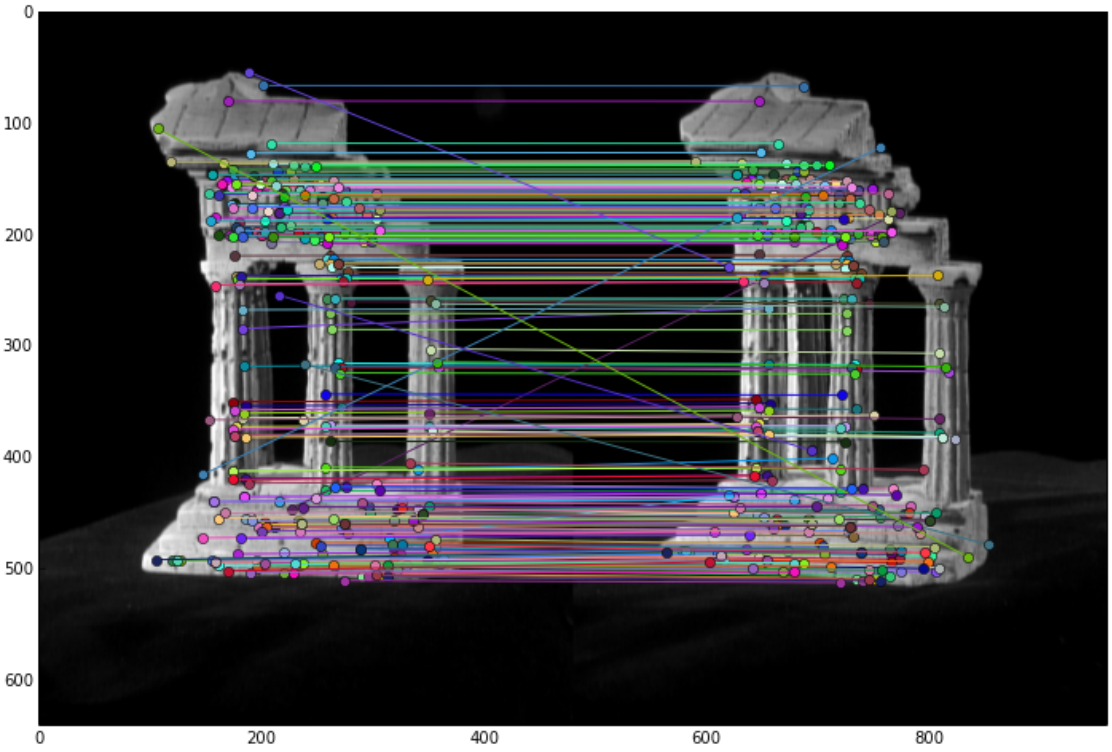
```
In [6]: from scipy.spatial import cKDTree
```

```
In [7]: kdtree_j = cKDTree(D_j)
        N_i = D_i.shape[0]
        d, nn = kdtree_j.query(D_i, k=2)
        ratio_mask = d[:,0]/d[:,1] < 0.6
        m = vstack((arange(N_i), nn[:,0])).T
        m = m[ratio_mask]

        # Filtering: If more than one feature in I matches the same feature in J,
        # we remove all of these matches
        h = {nj:0 for nj in m[:,1]}
        for nj in m[:,1]:
            h[nj] += 1

        m = array([(ni, nj) for ni, nj in m if h[nj] == 1])
```

```
In [8]: def rcolor():
            return (random.rand(), random.rand(), random.rand())

        def show_matches(matches):

            n_rows, n_cols = T1.shape
            display = zeros( (n_rows, 2 * n_cols), dtype=uint8 )
            display[:,0:n_cols] = T1
            display[:,n_cols:] = T2

            for pi, pj in matches:
                plot([K1[pi][0], K2[pj][0] + n_cols],
                     [K1[pi][1], K2[pj][1]],
```

```
                    marker='o', linestyle='-', color=rcolor())

        imshow(display, cmap=cm.gray)
```

```
In [9]: show_matches(m)
```



10.3 Recovering structure

Find the fundamental matrix `F`

```
In [10]: xi = K1[m[:,0],:]
         xj = K2[m[:,1],:]
```

```
In [11]: F, status = cv2.findFundamentalMat(xi, xj, cv2.FM_RANSAC, 0.5, 0.9)
         assert(det(F) < 1.e-7)
         is_inlier = array(status == 1).reshape(-1)

         inlier_i = xi[is_inlier]
         inlier_j = xj[is_inlier]
```

```
In [12]: hg = lambda x : array([x[0], x[1], 1])
```

10.3.1 Find epipolar matrix `E`

- The epipolar matrix `E` is defined as $\mathtt{E} = \mathtt{K}^\intercal\mathtt{F}\mathtt{K}$,
- `K` is the *calibration matrix* (*camera intrinsics*)
- The values used here are provided by the Temple Ring dataset

```
In [13]: K = array([[1520.4, 0., 302.32],
                    [0, 1525.9, 246.87],
                    [0, 0, 1]])
         K

Out[13]: array([[  1.52040000e+03,   0.00000000e+00,   3.02320000e+02],
                [  0.00000000e+00,   1.52590000e+03,   2.46870000e+02],
                [  0.00000000e+00,   0.00000000e+00,   1.00000000e+00]])

In [14]: E = dot(K.T, dot(F, K))
         U, s, VT = linalg.svd(E)

         if det(dot(U, VT)) < 0:
             VT = -VT
         E = dot(U, dot(diag([1,1,0]), VT))
         V = VT.T

         # Let's check Nistér (2004) Theorem 3 constraint:
         assert(det(U) > 0)
         assert(det(V) > 0)
         # Nistér (2004) Theorem 2 ("Essential Condition")
         assert sum(dot(E, dot(E.T, E)) - 0.5 * trace(dot(E, E.T)) * E) < 1.0e-10
```

10.3.2 Direct Linear Transform-based triangulation

Given two corresponding homogeneous points $\mathbf{x}_i$ and $\mathbf{x}_j$, observed in images I_i and I_j respectively, and the projection matrices $\mathtt{P}_i$ and $\mathtt{P}_j$, estimate the 3D points $\mathbf{X}$ in the scene associated to the pair $\mathbf{x}_i \leftrightarrow \mathbf{x}_j$.

- Rays back-projection
  - Imperfect measures $\mathbf{x}_i$ and $\mathbf{x}_j$ avoids rays intersection
- Alternative
  - Linear least-square formulation
  - Build a system $\mathtt{A}\mathbf{X}$
  - Solve using SVD

The point $\mathbf{X}$ could be estimated by rays back-projection in the 3D space. However, in general $\mathbf{x}_i$ and $\mathbf{x}_j$ are imperfect measures and the back-projected rays will not perfectly intersect in $\mathbf{X}$. Such a problem can be formalized as a linear least-square problem in the system of equations $\mathtt{A}\mathbf{X}$, where $\mathtt{A}$ is defined over $\mathbf{x}_i$, $\mathbf{x}_j$, $\mathtt{P}_i$ and $\mathtt{P}_j$ (see Section˜12.2 of Hartley and Zisserman's book). The following code uses the SVD implementation in SciPy to perform the minimization, finding the best estimation of $\mathbf{X}$. Note the last column of $\mathtt{V}$ can be indexed by -1.

```
In [15]: def dlt_triangulation(ui, Pi, uj, Pj):
             """Hartley & Zisserman, 12.2"""
             ui /= ui[2]
             xi, yi = ui[0], ui[1]

             uj /= uj[2]
             xj, yj = uj[0], uj[1]

             a0 = xi * Pi[2,:] - Pi[0,:]
             a1 = yi * Pi[2,:] - Pi[1,:]
             a2 = xj * Pj[2,:] - Pj[0,:]
             a3 = yj * Pj[2,:] - Pj[1,:]
```

```
    A = vstack((a0, a1, a2, a3))
    U, s, VT = linalg.svd(A)
    V = VT.T

    X3d = V[:,-1]

    return X3d/X3d[3]
```

10.3.3 Depth of points

Result 6.1 in Hartley and Zisserman's book) says:

> Let $\mathbf{X} = (X, Y, Z, T)^\intercal$ be a 3D point and $\mathtt{P} = [\mathtt{M}|\mathbf{p}_4]$ be a camera matrix for a finite camera. Suppose $\mathtt{P}(X, Y, Z, T)^\intercal = w(x, y, 1)^\intercal$. Then
>
> $\text{depth}(\mathbf{X}, \mathtt{P}) = \frac{\text{sign}(\det \mathtt{M})w}{Tvert\mathbf{m}^3vert}$
>
> is the depth of the point $\mathbf{X}$ in front of the principal plane of the camera.

In the result above, `M` corresponds to a view on the three first columns of `P` and $\mathbf{m}^3$ is the last row of `M`. The function below takes $\mathbf{X}$ and `P` and applies this result to compute the depth of $\mathbf{X}$.

```
In [16]: def depth(X, P):
             T = X[3]
             M = P[:,0:3]
             p4 = P[:,3]
             m3 = M[2,:]

             x = dot(P, X)
             w = x[2]
             X = X/w
             return (sign(det(M)) * w) / (T*norm(m3))
```

10.3.4 Getting the projection matrices from `E`

The code below implements the method described by Nistér (see *An efficient solution to the five-point relative pose problem*, section 3.1).

```
In [17]: def get_proj_matrices(E, K, xi, xj):
             hg = lambda x : array([x[0], x[1], 1])
             W = array([[0., -1., 0.],
                        [1.,  0., 0.],
                        [0.,  0., 1.]])

             Pi = dot(K, hstack( (identity(3), zeros((3,1))) ))

             U, s, VT = linalg.svd(E)
             u3 = U[:,2].reshape(3,1)

             # Candidates
             Pa = dot(K, hstack((dot(U, dot(W ,VT)), u3)))
             Pb = dot(K, hstack((dot(U, dot(W ,VT)), -u3)))
             Pc = dot(K, hstack((dot(U, dot(W.T ,VT)), u3)))
             Pd = dot(K, hstack((dot(U, dot(W.T ,VT)), -u3)))

             # Find the camera for which the 3D points are *in front*
```

```
            xxi, xxj = hg(xi[0]), hg(xj[0])

            Pj = None
            for Pk in [Pa, Pb, Pc, Pd]:
                Q = dlt_triangulation(xxi, Pi, xxj, Pk)
                if depth(Q, Pi) > 0 and depth(Q, Pk) > 0:
                    Pj = Pk
                    break

            assert(Pj is not None)

            return Pi, Pj
```

```
In [18]: P1, P2 = get_proj_matrices(E, K, inlier_i, inlier_j)
```

```
In [19]: print P1
```

```
[[  1.52040000e+03   0.00000000e+00   3.02320000e+02   0.00000000e+00]
 [  0.00000000e+00   1.52590000e+03   2.46870000e+02   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00]]
```

```
In [20]: print P2
```

```
[[ -1.42505476e+03   4.70752545e+01  -6.08289727e+02   1.52545806e+03]
 [  7.78012848e+00  -1.52389286e+03  -2.58854432e+02   1.99731458e+02]
 [  2.05743507e-01   5.27826740e-03  -9.78591717e-01   3.50422051e-01]]
```

10.3.5 Recovering the 3D points using DLT triangulation

```
In [21]: X = []

         for xxi, xxj in zip(inlier_i, inlier_j):
             X_k = dlt_triangulation(hg(xxi), P1, hg(xxj), P2)
             X.append(X_k)
         X = array(X)
```

10.3.6 3D plotting with Matplotlib

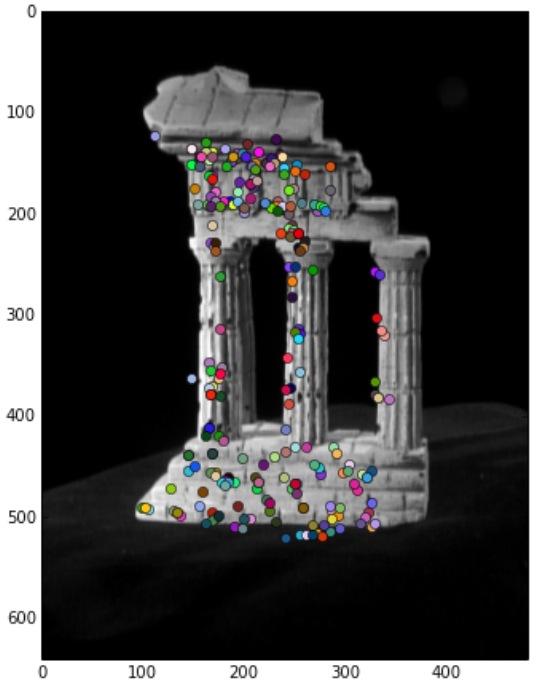
```
In [22]: num_pix = X.shape[0]
         pix_color = [rcolor() for k in range(num_pix)]
```

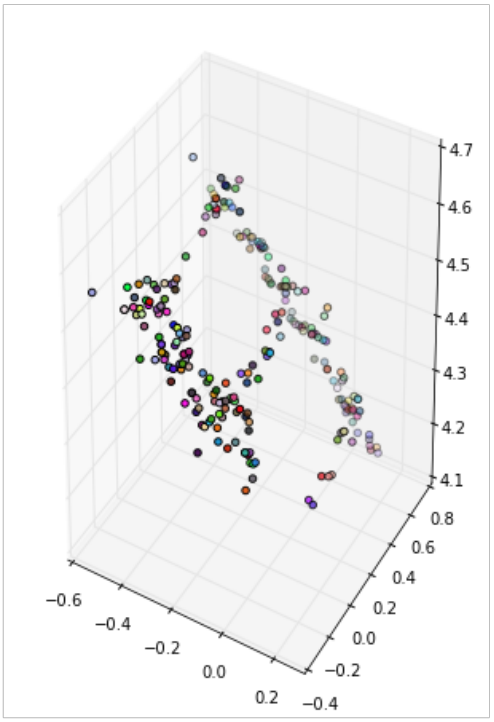
```
In [23]: pix = dot(P2, X.T).T
         pix = divide(pix, pix[:,2].reshape(num_pix, -1))
```

The 3D plot below is *static* because of the inline plotting. Other Matplotlib's backends allow us to rotate the 3D plot, letting us to inspect the results.

```
In [24]: from mpl_toolkits.mplot3d import Axes3D
         fig = figure()

         subplot(1,2,1)
         for k in range(num_pix):
             plot(pix[k,0], pix[k,1], color=pix_color[k], marker='o')
         imshow(T1, cmap=cm.gray)
         ax = fig.add_subplot(1, 2, 2, projection='3d')
         ax.scatter(X[:,0], X[:,1], X[:,2], zdir='z', c=pix_color)
```

```
Out[24]: <mpl_toolkits.mplot3d.art3d.Patch3DCollection at 0x5798750>
```





11 Conclusion

- The Python scientific computing environment keeps evolving
  - IPython development (Data Science team at Berkeley)
  - Scikit-learn (INRIA team)
  - Lots of contributions every year, spotted at SciPy Conference
- New arrivals: **deep learning**
  - Theano and Pylearn2 (LISA Lab, University of Montreal)
  - Theano
    - Expressions involving NumPy arrays efficiently computed in GPUs
  - Pylearn2
    - ConvNets

In his book, Prince argues that computer vision should be understood in terms of measurements (images), the world state, a model (defining the statistical relationships between the observations and the world), parameters, and learning and inference algorithms. The presented environment can address all these elements in modern computer vision R&D. Also, such a environment keeps evolving. For example, in the raising field of deep learning vision, packages as Pylearn2 and Theano are promising tools that could become an important part of the Python ecosystem for scientific computing in a near future.