

SciPy and OpenCV as an interactive computing environment for computer vision

Thiago T. Santos ¹

Data submissão: 12.08.2014

Data aceitação: 25.03.2015

Seção melhores tutoriais XXVII SIBGRAPI, Rio de Janeiro, Brasil, 2014.

Abstract: 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 SciPy Stack. Combined with 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 of computer vision research, from initial data exploration to parallel computing implementations. Several code examples are presented. They deal with problems from simple image processing to inference by machine learning. All examples are also available as IPython notebooks.

1 Introduction

Computer vision practitioners need a computing environment that lets them explore a variety of data like images, video sequences, point clouds and feature vectors. Such an environment should help the development and testing of new models and algorithms, and the deployment of the results, either as a final software module or a scientific/technical publication. It should also provide an *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 such high-performance computing (HPC) 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, allowing results reproducibility [1], but preventing the research pipeline to become more cumbersome.

In the last twelve years, a powerful scientific computing environment emerged from the Python programming community. This language is an attractive option for researchers:

¹Embrapa Agricultural Informatics, POBOX 6041
thiago.santos@embrapa.br

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 advent of an efficient module for n -dimensional array representation and manipulation was the tipping point for Python, which has become a major player in scientific computing. The Numarray module was created by Greenfield *et al.* [2] to address astronomical data analysis. In 2005, Numarray successor, NumPy [3, 4], appeared and became the workhorse of the Python scientific computing. An active community composed of scientists and engineers flourished around Python and NumPy, represented today by the SciPy Stack and the SciPy Conferences².

1.1 Why is an interactive environment important to computer vision?

In computer vision, the practitioner is interested in inferring the *world state* from images, which 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 [5] and translates the state-of-the-art of contemporary research in the field, which is deeply associated to machine learning nowadays.

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 [6], enriched with tools from the SciPy Stack and the OpenCV library, can address these needs.

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 intend 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.

This tutorial is organized as follows. Section 2 presents the IPython interactive Python shell. Section 3 briefly presents how OpenCV can be used under Python. Section 4 presents the core components of SciPy Stack: the NumPy n -dimensional arrays, the Matplotlib plotting library and the scientific library. Machine learning receives attention in Section 5 where the scikit-learn module is presented. Section 6 introduces *IPython parallel* high performance computing capabilities. A longer example on structure from motion illustrates various fea-

²<http://conference.scipy.org>

tures of the environment combined in Section 7. Finally, Section 8 presents some concluding remarks. Orientations regarding the environment’s installment and initialization can be found in the appendices. *IPython notebooks* presenting the complete code for the examples are available at GitHub³.

An important clarification to the reader: along this tutorial, code examples will show the numbered inputs, presented by blocks of code preceded by “In [n]:”, and the numbered outputs, the result produced by the inputted code block, preceded by “Out [n]:”. The purpose is to illustrate the experience on working in an interactive computing environment. However, in some cases the code block of a numbered input produces no output at all, so the IPython shell will omit the corresponding “Out [n]:” text.

2 The IPython shell

IPython is an enhanced Python shell for interactive distributed and parallel computing [6]. 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 [6] argue that, in a research context, determining which computations must be performed next often requires significant work. This is particularly true for computer vision by the reasons presented in Section 1.1.

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 an underscore “_” and the number of the output, as shown in the following example:

```
In [1]: r = 3.
```

```
In [2]: 2 * pi * r
Out[2]: 18.84955592153876
```

```
In [3]: _2
Out[3]: 18.84955592153876
```

```
In [4]: 2 * _2
Out[4]: 37.69911184307752
```

³ <http://nbviewer.ipython.org/github/thasant/scipy4cv>

IPython has other useful features: `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 SciPy library. Regarding system shell integration, not only can IPython call any system command, but it also can capture the shell's output in Python variables and call system commands with values computed from variables.

These simple features are useful, but they alone would not explain IPython popularity in scientific computing. The three features that make IPython an outstanding scientific computing environment are (i) the support for data visualization, (ii) the *notebook* environment and (iii) the facilities for interactive distributed and parallel computation. Visualization features will be presented in Section 4.2, in the context of Matplotlib. The notebook environment is presented next.

2.1 The IPython Notebook

The IPython Notebook is a web-based version of the IPython shell presenting extended functionality⁴. In a 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 is organized in a single executable document. Notebooks are being used for research notes and the production of articles and books [7].

A *notebook server* is a Web server that will read/write notebook files from/to the current directory. Users can employ any standard Web browser to edit code, with the help of automatic syntax highlighting, and tab completion (Figure 1).

In a notebook, a *Markdown cell* is able to render rich-formatted text using the Markdown markup language. Mathematical notation is defined using \LaTeX syntax and rendered by MathJax. Figure 1 (c) shows a Markdown cell produced by the following plain text:

```
# Whitening

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


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

```

⁴Appendix B presents how to execute IPython in console, qtconsole and notebook modes.

Figure 1 (b) shows a *code cell*. It contains Python code⁵ that is sent to the IPython interpreter running on the server and executed, then the resulting output is 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*), as seen in Figure 1 (d).

IPython notebooks are stored in JSON files that keep the cells' content. These files use 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 that they are *executable documents*: they are not only able to store textual and mathematical descriptions but also to replicate the computations.

3 OpenCV

OpenCV is a popular library in the computer vision community. It has been used widely in industry and academy. Started in 1999 and popularized in the following decade, OpenCV is covered in books [8, 9, 10] and tutorials [11, 12], so this text will not provide another overview of the library. The interested reader who is not familiar with OpenCV should refer to those cited texts and the library's official documentation [13].

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.

The OpenCV's Python module can be easily imported using:

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

In this tutorial every code that starts with the "`cv2.`" prefix is using OpenCV's functions. Several examples of the OpenCV usage under Python are found in the next sections.

4 SciPy Stack

SciPy maintainers define it as an ecosystem of open-source software for mathematics, science, and engineering. This section introduces three of the core packages: NumPy (*n-*

⁵IPython notebooks are being extended to support other programming languages. The Julia language is supported to date.



Figure 1. An IPython notebook. (a) IPython creates a Web server and users edit and manipulates their notebooks using a Web browser. (b) A *code cell* executes Python scripts. (c) A *Markdown cell* is able to present rich-formatted text and mathematical formulas using \LaTeX . (d) An *inline plot* using Matplotlib.

dimensional arrays), Matplotlib (visualization and plotting) and the SciPy Library (numerical algorithms and domain-specific toolboxes). IPython, presented previously in Section 2, is also considered a core package of the stack. The scikit-learn package for machine learning is not part of the core, but it is considered a major component of the stack and will be introduced in Section 5. The next section will present the base that supports all the stack: NumPy.

4.1 NumPy

A NumPy *array* is a multidimensional and uniform collection of elements (all elements occupy the same number of bytes in memory) [4]. It is the standard n -dimensional array representation in the SciPy Stack and, in OpenCV operation under Python, it replaces the OpenCV's `Mat` type as the data structure for images. A NumPy array is commonly used to represent images, matrices, vectors and data points. It is also able to represent general n -d arrays up to 32 dimensions. NumPy avoids expensive `for` loops executed by the Python interpreter when operating on arrays. Instead, efficient *vectorized* operations are internally executed by machine code, resulting in improved performance.

In Python environment, the NumPy module can be imported as follows:

```
In [1]: import numpy as np
```

Then, the module's functions and variables can be accessed by the declared name:

```
In [2]: np.arange(9)
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

Alternatively, names from a module can be imported directly into the symbol table as follows:

```
In [1]: from numpy import *
In [2]: arange(9)
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

In IPython, the `-pylab` option pre-load NumPy for interactive use, producing the same effects as the import operation above (see Appendix B).

4.1.1 Images as NumPy arrays 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 [18]: lenna = cv2.imread('../data/lenna.tiff',
```

```
cv2.IMREAD_GRAYSCALE)
```

```
In [20]: lenna.shape
Out[20]: (512, 512)
```

Grayscale images are commonly represented by 2-D 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 [21]: lenna.dtype
Out[21]: dtype('uint8')
```

```
In [22]: lenna
Out[22]:
```

```
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 [23]: lenna[0,0]
Out[24]: 162
```

The `uint8` data type in NumPy corresponds to the `CV_8U` type in OpenCV. Similarly, 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 an $M \times N \times 3$ `uint8` array:

```
In [11]: mandrill = cv2.imread('../data/mandrill.tiff')
```

```
In [12]: mandrill.shape
Out[12]: (512, 512, 3)
```

```
In [14]: mandrill.dtype
Out[14]: dtype('uint8')
```

```
In [15]: mandrill[2,3]
```

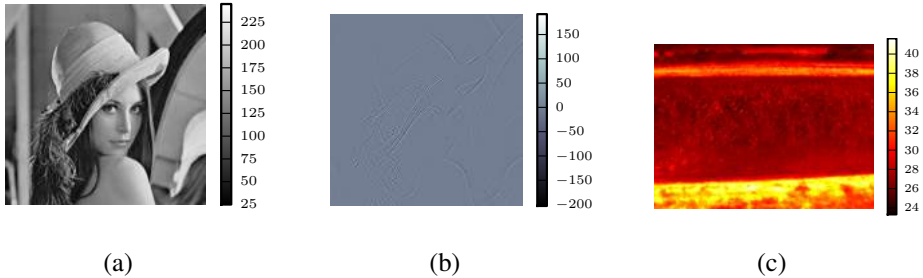



Figure 2. Images as NumPy arrays. (a) A grayscale image represented as an `uint8` array. (b) The convolution by a Sobel’s kernel produced an image presenting negative values, represented by an `int32` array. (c) A thermogram of a maize field represented by a `float32` array, where each pixel’s value corresponds to a temperature in Celsius degrees.

```
Out [15]: array([29, 46, 54], dtype=uint8)
```

The output of the last command above shows that 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 obtained by indexing the color dimension: `mandrill[2, 3, 1]`, which returns the green channel value, 46.

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. Thermography images and depth images present real numbers that are better represented by floating-point values. Figure 2 shows some examples of images represented with different data types.

4.1.2 Indexing and slicing As seen previously in the *Mandrill* example, array’s elements can be indexed with 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 a 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. For example, if the user is only interested in rows 3, 5, 7 and 9, she could use a step of 2, producing `A[3:10:2, 5:9]`. This slicing example is illustrated in the code below:

```
In [173]: A
Out [173]:
```

```
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 [178]: A[3:10:2, 5:9]
```

```
Out[178]:
```

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

Example 1 (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 by applying a logical operation on an array.*

```
In [1]: lenna > 128
```

```
Out[1]:
```

```
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)
```

```
In [2]: res = zeros_like(lenna)
```

```
res[lenna > 128] = 255
```

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

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

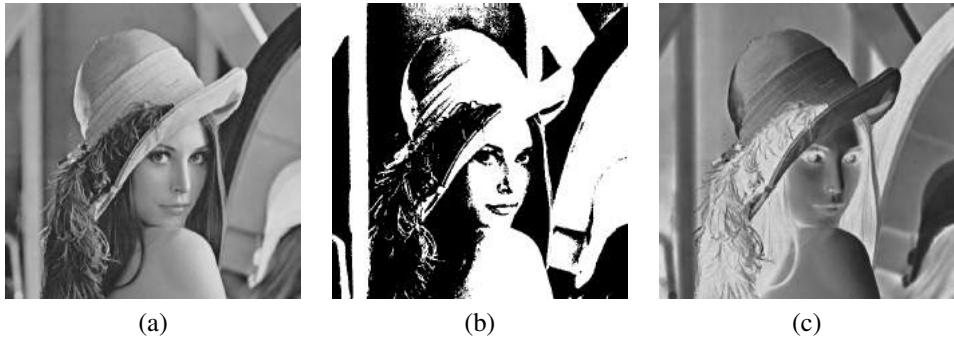


Figure 3. Thresholding by indexing a NumPy array properly. (a) Original *Lenna* image. (b) Thresholded image produced with *fancy indexing*, i.e. by employing a boolean mask to perform the maximum value attribution. (c) Negative produced by a vectorized subtraction between a scalar and an array.

Example 2 (Whitening) *Factors like ambient light intensity or 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 shows how whitening can be efficiently performed by vectorized operations, without losing the simplicity of its mathematical definition.*

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

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

This operation is easily implemented by using NumPy as follows:

```
In [38]: mu = mean(I)
         sigma = std(I)
         W = (I - mu)/sigma
```

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 . Figure 4 shows the result of whitening on a low-contrast picture.

4.1.3 Image ROIs and array views Sometimes, procedures must be limited to a *region of interest* (ROI), a rectangular part of the image. In NumPy, the ROI is equivalent to the idea

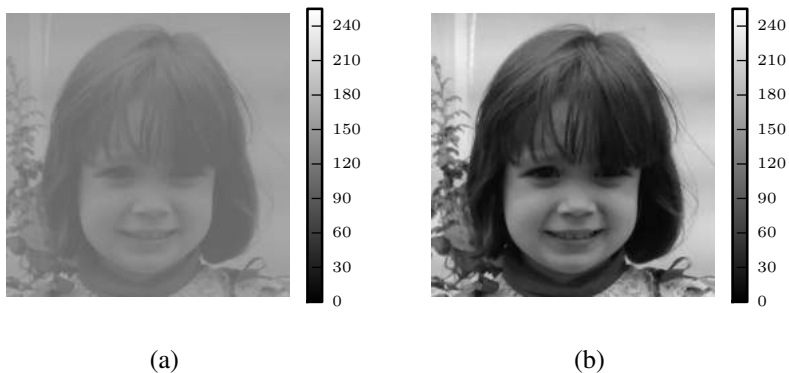


Figure 4. Whitening. (a) Original low-contrast image. (b) Result after whitening through efficient vectorized operations in NumPy (values are converted to the [0, 255] range for 8 bits representation).

of *view*, an array that shares memory with another one. In the example below, *B* is a view on array *A*. As expected, changes in *B* values produce the same changes in *A*.

```
In [61]: A
Out [61]:
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 [62]: B = A[0:3,0:3]
         B
```

```
Out [62]:
array([[ 0,  1,  2],
       [ 5,  6,  7],
       [10, 11, 12]])
```

```
In [63]: B[0,0] = 255
         B
```

```
Out [63]:
array([[255,  1,  2],
       [  5,  6,  7],
```

```
[ 10, 11, 12]])
```

```
In [64]: A
```

```
Out [64]:
```

```
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]])
```

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, so A and B do not share any memory.

```
In [65]: B = A[0:3,0:3].copy()
```

Reshaping is other operation that produces a view on an array. The reshaped array is a view that presents the same number of elements, but different dimensions. As an example, consider the Handwritten Digits Data Set in the UCI Machine Learning Repository [14] (Figure 5). 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 8×8 image as follows:

```
In [80]: X = x.reshape(8,8)
```

Similarly, a 8×8 image can be viewed as a 64-D vector as follows:

```
In [81]: x = X.reshape(-1)
```

4.2 Matplotlib

Matplotlib [15] is a 2-D 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 fields⁶.

⁶Matplotlib's maintainers keep a *gallery* of plotting examples at <http://matplotlib.org/gallery.html>. 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.

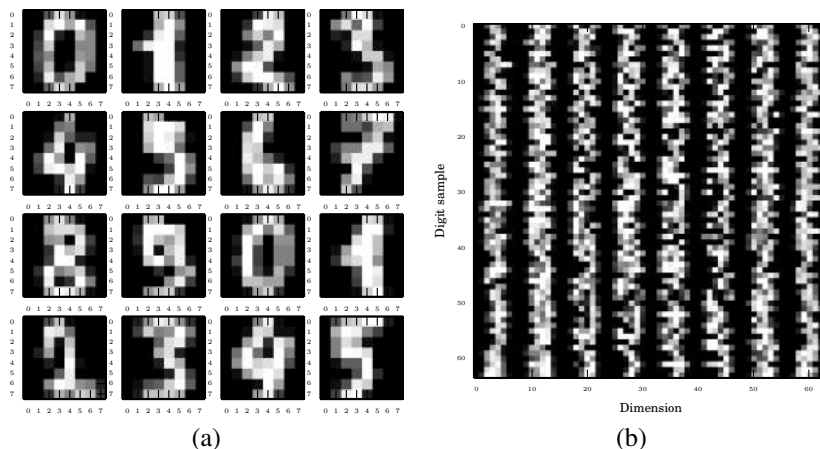


Figure 5. Reshaping. (a) The handwritten digits dataset composed by 8×8 images [14]. (b) Each image is reshaped in a 64-D feature vector for classification.

4.2.1 OpenCV `imshow` vs. Matplotlib `imshow` 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 adaptations. 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, which maps the value range $[0, 1]$ to $[0, 255]$. Zooming is not supported and high-resolution images do not benefit the computer's screen, making the visualization of large images inconvenient. Remote sensing data or high-resolution photographs usually needs some sort of scaling before displaying.

Matplotlib presents its own `imshow` function, which is 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. Exhibiting bidimensional $M \times N$ arrays (as integer or floating-point grayscale images) can be done with 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, as shown in Figures 2 and 4.

The simplest call to `imshow` just takes the array to be visualized:

```
In [64]: imshow(slenna)
         colorbar()
```

The standard setting employs bilinear interpolation and the `cm.jet` colormap, as seen in Figure 6 (a). Other types of interpolation and colormaps can be specified by using the arguments `interpolation` and `cmap`. The code:

```
In [65]: imshow(slenna, cmap=cm.gray, interpolation='nearest')
         colorbar()
```

produces the result shown in Figure 6 (e).

Example 3 (Showing 2-D features) *In this example, GFTT [16], SIFT [17] and SURF [18] features as computed by OpenCV and the results are visualized by using Matplotlib `plot` and `scatter` functions.*

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

```
In [18]: kpts = cv2.goodFeaturesToTrack(graffiti, 2000, 0.01, 3)
         plot(kpts[:, :, 0], kpts[:, :, 1], 'b+')
         imshow(graffiti, cmap=cm.gray)
         title(r'Shi-Tomasi GFTT')
```

In the code above, OpenCV's GFTT will return up to 2000 corners (features), all of them at least 3 pixels apart. The 0.01 is a quality level parameter defined over the eigenvalues (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 (see Section 4.1.2). The "b+" argument informs Matplotlib the points that have to be plotted as blue crosses. Finally, `imshow` is employed to display the image, which results in the plotting shown in Figure 7 (b).

Differently of the GFTT features, SIFT and SURF are not dimensionless. They are multiscale features [19] and are defined over neighborhoods presenting different sizes. OpenCV represents these features by 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:

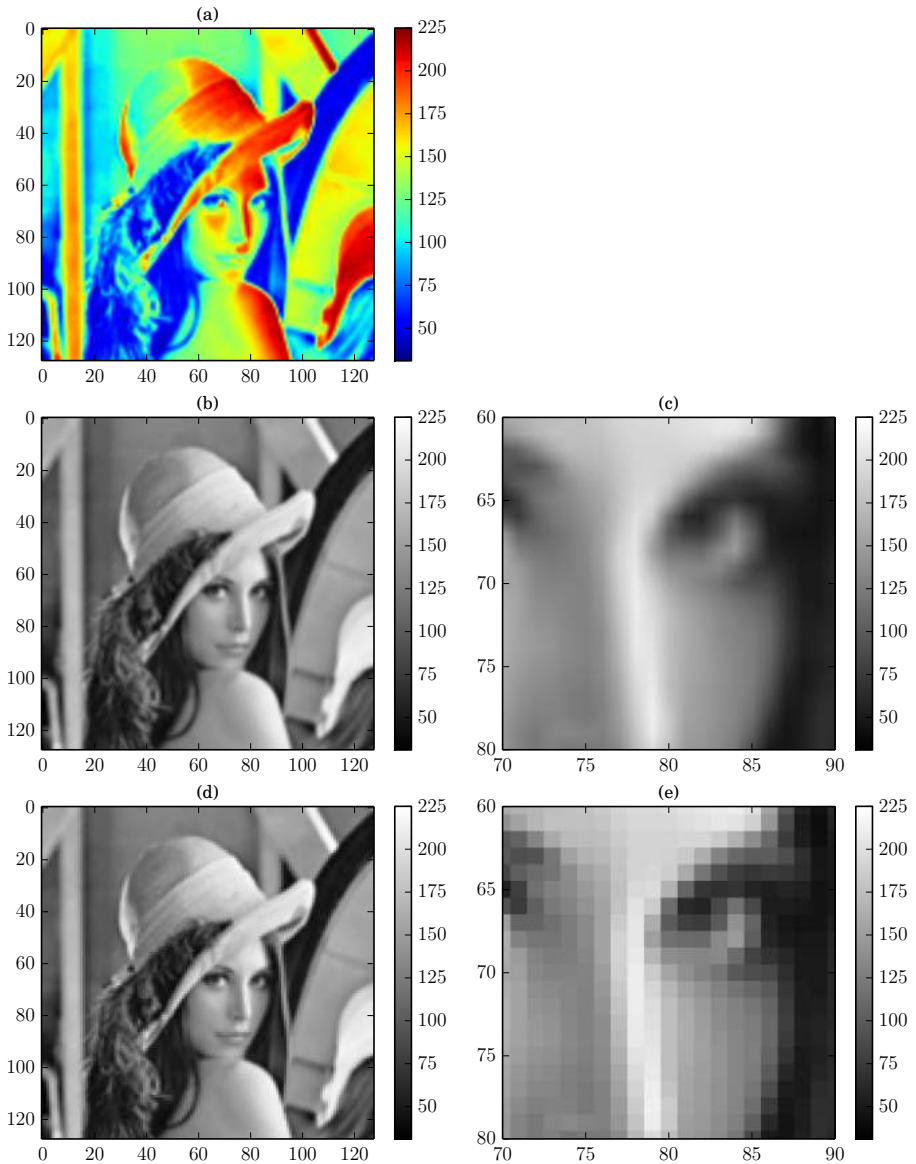


Figure 6. Visualizing images with Matplotlib’s `imshow`. In this example, a 128×128 grayscale version of *Lenna* is employed. (a) Default setting with bilinear interpolation and the `cm.jet` colormap. (b) Bilinear interpolation and the `cm.gray` colormap. (c) Example of zooming in on the bilinear interpolated image. (d) The “nearest” interpolation and `cm.gray` colormap. (e) Example of zooming in on the nearest interpolated image.


```
In [19]: sift = cv2.SIFT()
        kpts = sift.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='r', alpha=0.4)
        imshow(graffiti, cmap=cm.gray)
        title(r'SIFT')
```

In the code above, the `alpha` parameter is employed to plot with a 40% transparency, allowing the viewer to see the image under the circles, and the “`r`” parameter asks for a plotting in red. The resulting plot can be seen in Figure 7 (c). A similar plot, but displaying SURF features, is shown in Figure 7 (d).

4.3 The SciPy Library

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, which contains implementations for the KD-Tree data structure (used on nearest-neighbor queries), Delaunay triangulation and convex hull computation.

Example 4 (Descriptors matching with KD-Trees) *In this example, a KD-Tree is employed to perform features descriptors matching between two images using the criteria proposed by Lowe [17].*

Consider two sets of descriptors, \mathbf{D}_i and \mathbf{D}_j , computed by SIFT or SURF, for two images, I_i and I_j . Matching can be performed by 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 performed by representing \mathbf{D}_j in a KD-Tree⁷.

Lowe [17] 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 comparing the distances to the nearest neighbor and to 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 to their closest neighbors:

⁷KD-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 with KD-Trees, a dimensionality reduction procedure, as PCA, is recommended. Sklearn and OpenCV provide PCA implementations.

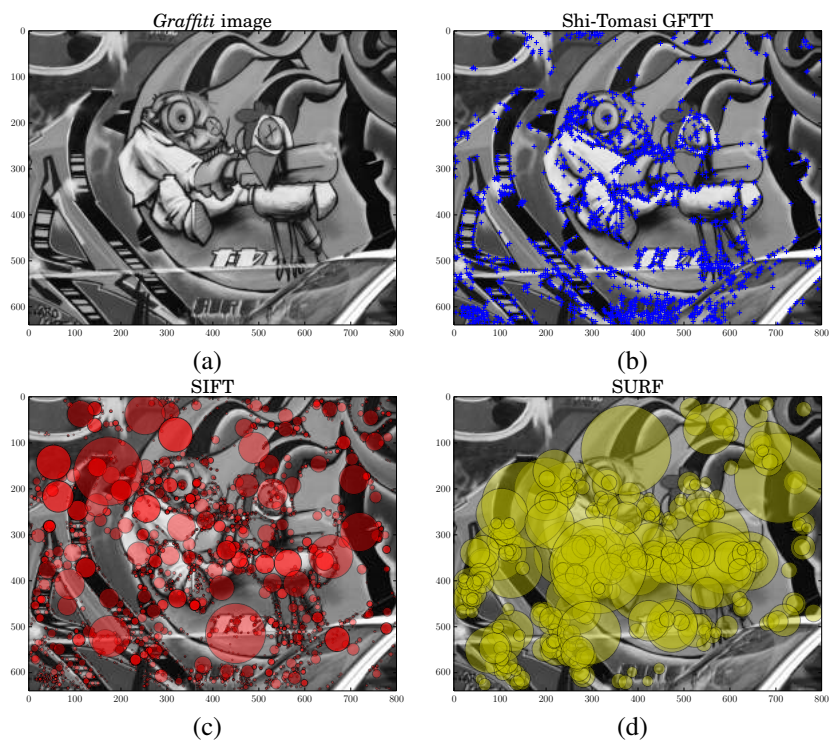


Figure 7. (a) The *Graffiti* image [19]. (b) Features found by the Shi-Tomasi's method [16], plotted as points marked by crosses. (c) Features found by SIFT [17]. (d) Features found by SURF [18]. The `scatter` function in Matplotlib was used to plot the features according to their neighborhood (size).

```
In [26]: from scipy.spatial import cKDTree
In [27]: 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]
```

A KD-Tree is created from the N_j vectors in the array \mathbf{D}_j . The `query` method in the `KDTree` object 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 returns 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 . A boolean mask, `ratio_mask` is produced to select just the entries that obey the ratio constraint. A $N_i \times 2$ array is created, keeping in each row the indexes n_i and n_j , and n_i -th descriptor in \mathbf{D}_i matches the n_j -th descriptor in \mathbf{D}_j . This array `m` is produced by `vstack`, a function that “stacks” 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 ensuring that a matched feature $\mathbf{d}_j \in \mathbf{D}_j$ corresponds to a single feature $\mathbf{d}_i \in \mathbf{D}_i$ [20]. The code below implements this filtering:

```
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 the code above, `h` is a Python dictionary acting as a 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. Figure 8 shows the produced results for two images from the *Temple Ring* dataset [21].

4.4 Linear algebra using `scipy.linalg`

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 [22]. Consider a point \mathbf{X}_i in 3-D, represented as a *homogeneous* 4-D array, and a projective matrix \mathbf{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:

```
x = dot(P, X)
```

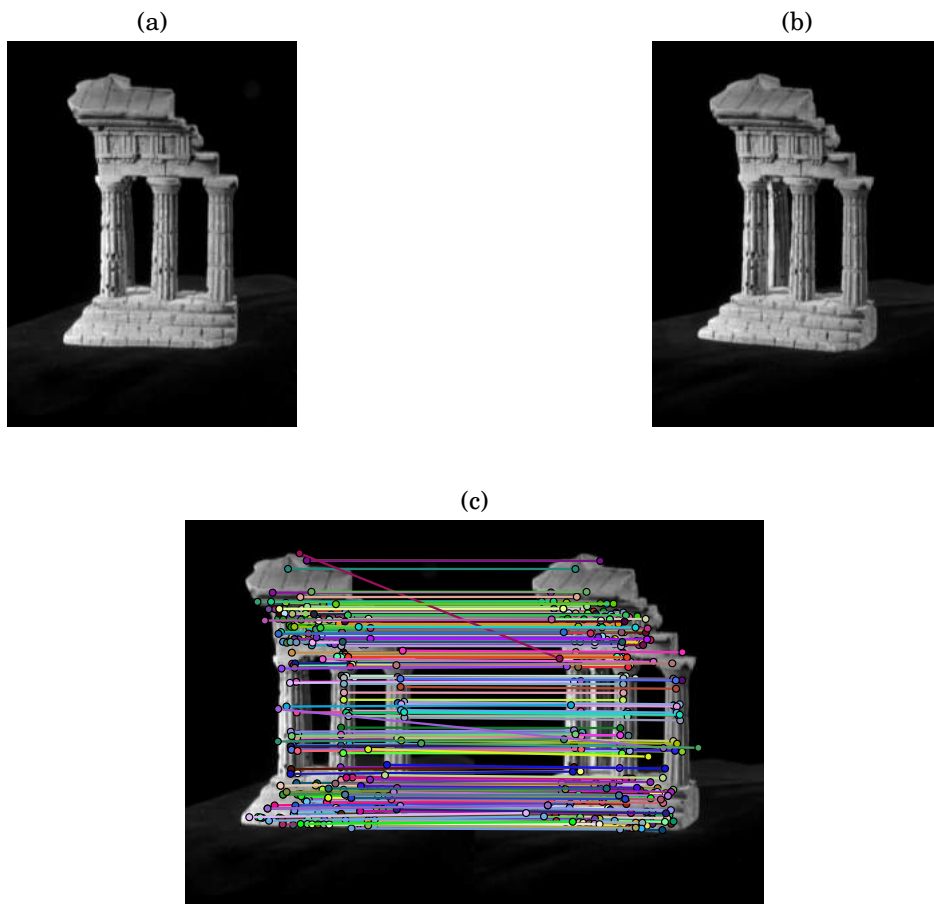


Figure 8. Features matches for 3-D reconstruction. (a) and (b) Frames 34 and 36 from the *Temple Ring* dataset [21]. (c) SIFT features detection and matching using the matching criteria proposed by Lowe [17].

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:

```
x_coord = x[0]/x[2]
y_coord = x[1]/x[2]
```

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 \mathbf{Ax} , minimizing $\|\mathbf{Ax}\|$ subject to $\|\mathbf{x}\| = 1$ [22].

The minimization problem can be solved using *singular value decomposition* (SVD), a matrix decomposition particularly useful in numerical computations. SVD decomposes \mathbf{A} as $\mathbf{A} = \mathbf{UDV}^T$ and the \mathbf{x} solution⁸ corresponds to the last columns of \mathbf{V} .

Example 5 (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 \mathbf{P}_i and \mathbf{P}_j , we can estimate the 3-D point \mathbf{X} in the scene associated to the pair $\mathbf{x}_i \leftrightarrow \mathbf{x}_j$.*

The point \mathbf{X} could be estimated by back-projection of rays in the 3-D space. However, \mathbf{x}_i and \mathbf{x}_j are imperfect measures and the back-projected rays would not perfectly intersect in \mathbf{X} . Such a problem can be formalized as a linear least-square problem in the system of equations \mathbf{AX} , where \mathbf{A} is defined over \mathbf{x}_i , \mathbf{x}_j , \mathbf{P}_i and \mathbf{P}_j (see Section 12.2 of [22] for details). The following code uses the SVD implementation in SciPy to perform the minimization, finding an estimation for \mathbf{X} :

```
In [18]: a0 = xi[0] * Pi[2, :] - Pi[0, :]
         a1 = xi[1] * Pi[2, :] - Pi[1, :]
         a2 = xj[0] * Pj[2, :] - Pj[0, :]
         a3 = xj[1] * Pj[2, :] - Pj[1, :]

         A = vstack((a0, a1, a2, a3))
         U, s, VT = linalg.svd(A)
         V = VT.T
         X = V[:, -1]
```

Note that the last column of \mathbf{V} can be indexed by -1 . This procedure will be used again in the structure from motion example in Section 7.

⁸The interested reader should refer to Chapter 4 and Appendix 5 of Hartley and Zimmerman's book [22].

5 Machine learning with scikit-learn

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 [23], 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. 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.

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 should refer to the sklearn on-line documentation [24], which is rich in descriptions, tutorials and code examples. Readers interest in machine learning and its applications in vision should refer to Bishop's [25] and Prince's [5] books.

Example 6 (Skin detection using Naïve Bayes) *In this example, Naïve Bayes classification is employed to detect pixels corresponding to human skin through pixel color measurement.*

Let `training` be a $M \times N \times 3$ array representing a color training image in the CIE $L^* a^* b^*$ 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.

The data is composed by MN 2-D vectors, which are easily extracted from the training image using reshaping and slicing [see Figure 9 (a)]:

```
In [4]: data = training.reshape(M*N, -1)[: ,1:]
        data
Out [4]:
array([[128, 129],
```

```
[128, 129],
[128, 129],
...,
[127, 129],
[125, 134],
[123, 136]], dtype=uint8)
```

Similarly, the manual classification [Figure 9 (b)] used in the learning step is represented as a binary MN vector:

```
In [5]: target = mask.reshape(M*N)
        target
Out [5]:
array([ 0.,  0.,  0., ...,  0.,  0.,  0.]
```

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

Skin detection can be performed converting the input image [Figure 9 (c)] to the $L^* a^* b^*$ 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 [Figure 9 (d) and (e)]:

```
In [7]: test_bgr = cv2.imread('../data/thiago.jpg')
        test = cv2.cvtColor(test_bgr, cv2.COLOR_BGR2LAB)
        M_tst, N_tst, _ = test.shape

In [8]: data = test.reshape(M_tst * N_tst, -1)[: ,1:]
        skin_pred = gnb.predict(data)
        S = skin_pred.reshape(M_tst, N_tst)
```

Example 7 (Color segmentation using mean-shift clustering) *In this example, the mean-shift algorithm [26] is employed to perform color segmentation, grouping similar colors together (color quantization).*

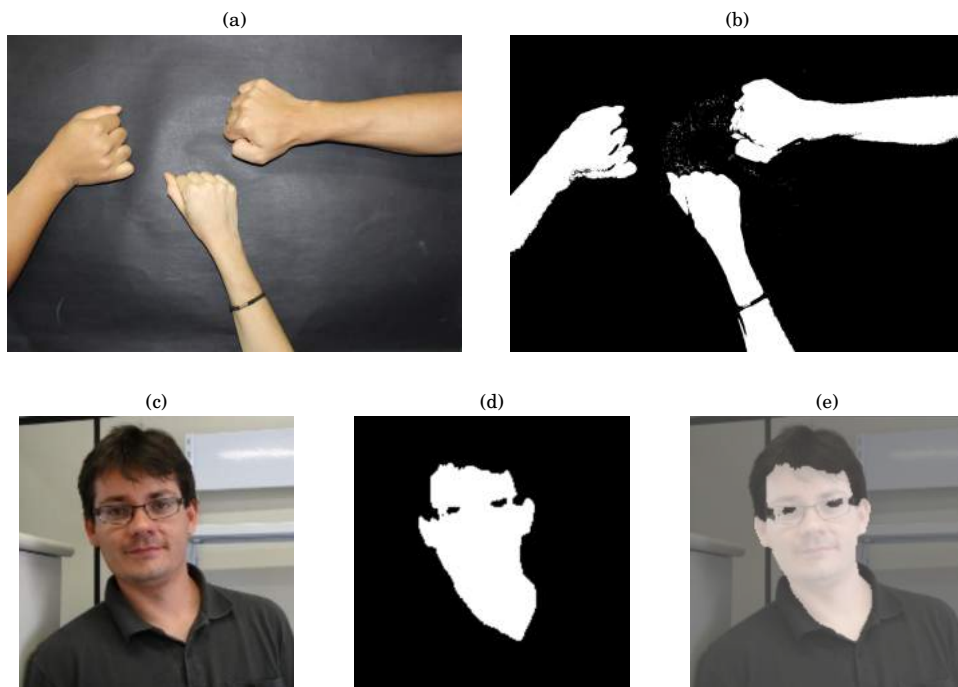


Figure 9. Skin classification using Naïve Bayes. (a) Training image. (b) Binary classification used in supervised learning. (c) Input image to skin detection. (d) Naïve Bayes classification results. (e) Classification results and the input image shown together.

This clustering procedure relies on 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 L*a*b 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 [2]: I = cv2.imread('../data/BSD-118035.jpg')
        I_Lab = cv2.cvtColor(I, cv2.COLOR_BGR2LAB)
        M, N, _ = I_Lab.shape
```

```
In [3]: X = I_Lab.reshape(M*N, -1)
        X
```

```
Out[3]:
array([[ 33, 121, 120],
       [ 33, 121, 120],
       [ 33, 121, 120],
       ...,
       [122, 122, 118],
       [125, 122, 120],
       [ 38, 122, 126]], dtype=uint8)
```

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 between 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 [4]: from sklearn.cluster import MeanShift
        from sklearn.cluster import estimate_bandwidth
```

```
In [5]: b = estimate_bandwidth(X, quantile=0.1, n_samples=2500)
        ms = MeanShift(bandwidth=b, bin_seeding=True)
        ms.fit(X)
```

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 [6]: S = zeros_like(I)
        L = ms.labels_.reshape(h, w)
        num_clusters = ms.cluster_centers_.shape[0]
        for c in range(num_clusters):
            S[L == c] = ms.cluster_centers_[c]
        imshow(cv2.cvtColor(S, cv2.COLOR_LAB2RGB))
```

Figure 10 shows clustering results⁹ for three images from the Berkeley Segmentation Dataset [28].

Example 8 (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 MN mixtures of K multivariate Gaussians. Stauffer and Grimson [29] 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 to as the “waving trees” problem [30], and multimodal distributions are a better way to represent this variation.

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 Gaussians are being used ($K = 3$), letting each model represents 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.

```
In [15]: K = 3
        MN = M * N
        bgmodel = [GMM(n_components=K).fit(V[600:750,i])
                    for i in range(MN)]
```

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

⁹In this example, just the pixel color was employed, resulting in color quantization. To perform spatial-color segmentation as proposed by Comaniciu and Meer [27], a multivariate kernel is needed. To the date, multivariate kernels are not available in sklearn's mean-shift implementation.

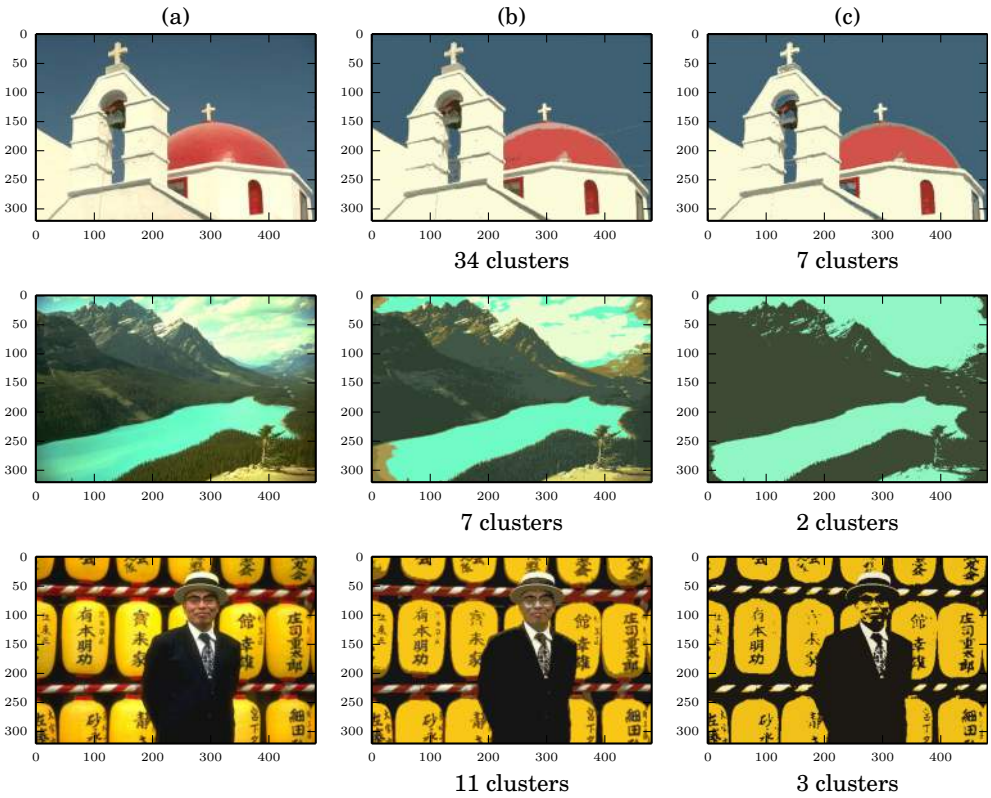


Figure 10. Mean-shift based color segmentation. (a) Input images from the Berkeley Segmentation Dataset [28]. The bandwidth parameter is selected evaluating the distances among 2,500 sample pixels and taking a quantile. (b) Mean-shift clustering results using a 5% quantile. (c) Results employing a 20% quantile.

```
In [21]: bg_mean = array([gmm.means_ for gmm in bgmodel])
         bg_weight = array([gmm.weights_ for gmm in bgmodel])
```

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

```
In [25]: c = array([bgmodel[i].predict([V[t,i]])
                  for i in range(N)])
```

```
In [26]: pmean = array([bgmodel[i].means_[c[i]]
                       for i in range(MN)])
         pcov = array([bgmodel[i].covars_[c[i]]
                      for i in range(MN)])
```

Here, the prediction step just selects one of the K Gaussians, c_i , as the proper model to the i -th pixel. Note that the code above also recovers the mean and the covariance matrix of c_i . The pixel is declared *background* if (i) the weight 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. The criteria in (ii) can be easily implemented as:

```
In [40] stmt = abs(V[t] - pmean) < 2. * sqrt(pcov)
```

Figure 11 presents the result for background classification in 1000-th frame of the *Left Bag* video sequence in the CAVIAR dataset [31].

6 Performance and HPC

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 the software in SciPy Stack relies on a base of numerical software implemented in C/C++ and Fortran, including the efficient NumPy arrays.

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 [32] raises as an alternative. Cython is a static compiler capable of working

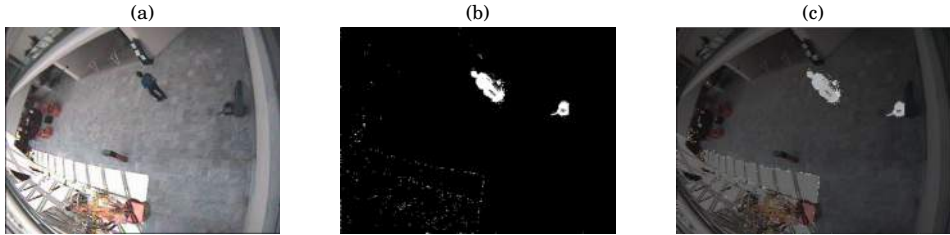


Figure 11. Background subtraction using Gaussian Mixture Models. (a) Input frame from the *Left Bag* sequence in the CAVIAR dataset. (b) Result of the background subtraction. (c) Background subtraction result and the input frame shown together.

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.* [32], 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.

Other performance issues can be addressed by parallelization. IPython.parallel is a powerful architecture for parallel and distributed computing which supports 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: it can be implemented in a few lines of code with IPython.parallel. 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.

Example 9 (Processing 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 a system shell, an IPython cluster for parallel computing is started as follows:

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

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 [4]: @lview.parallel()
        def get_num_matches(arg):
            fname, D_src = arg
            import cv2
            frame = cv2.imread(fname, cv2.IMREAD_GRAYSCALE)
            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)
```

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

```
In [6]: for f, n in async_res:
        print f, n
```

```
Out [6]:
/tmp/templeRing/templeR0001.png 802
/tmp/templeRing/templeR0002.png 549
```

```
/tmp/templeRing/templeR0003.png 491
...
/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 Elastic Compute Cloud (Amazon EC2)¹⁰.

7 A complete case: structure from motion

The last example in this tutorial combines different packages to solve a structure from motion problem [22, 33] in two images. It starts using OpenCV to detect SIFT features and their descriptors [17]. 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 [22, 34]. Finally, SVD decomposition, implemented in `scipy.linalg`, is used to perform triangulations and estimate a 3-D point for each pair of matching features. The code and comments are available online¹¹ as an IPython notebook. Figure 12 shows the result for a pair of frames in the *Temple Ring* dataset [21].

8 Conclusion

In his book, Prince [5] argues that computer vision should be understood in terms of measurements (images), world state, 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.

Other interactive environments also provides image processing and computer vision capabilities. MATLAB[®] is a famous comercial alternative that also provides a notebook-like functionality. But the Python environment has attracted users because it is free, open-source

¹⁰The interested reader is referred to the section *Using IPython for parallel computing* in the IPython documentation – <http://ipython.org/documentation.html>.

¹¹<http://nbviewer.ipython.org/github/thsant/scipy4cv>

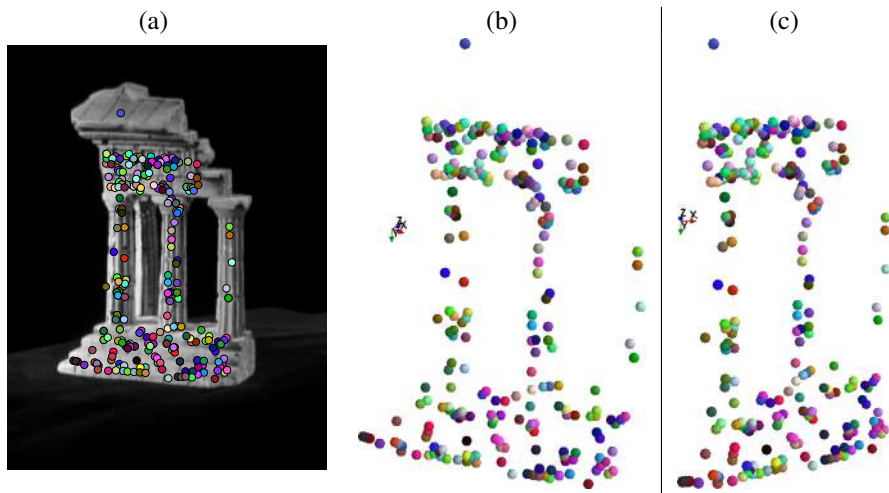


Figure 12. Results for a 3-D reconstruction from 2-D features matches. (a) Frame from the *Temple Ring* dataset [21] – the estimated three-dimensional points \mathbf{X}_i are projected using the estimated projection matrix \mathcal{P} . (b) and (c) Recovered 3-D model showing different views of the point cloud \mathbf{X}_i . Each point is plotted with the same color in the three images.

and based in a general-purpose language. It probably has the best-designed of the digital notebooks to date [35, 36].

The large Python community keeps the environment evolving. For example, in the raising field of deep learning vision, packages as Pylearn2 [37] and Theano [38] are promising new tools that could become an important part of the Python ecosystem for computer vision and scientific computing in a near future.

A Installation

As a general guideline, users should always consider the most recent installation instructions provided by the packages' maintainers¹². Currently, Windows, Mac and Linux users could consider *Anaconda*, the Continuum Analytics distribution¹³, 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

¹²See <http://www.scipy.org/install.html>

¹³Available at <http://continuum.io/downloads>

apt-get system in Debian/Ubuntu distributions). The `pip` tool¹⁴ is also a practical way to get the packages and keep them updated:

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

B IPython initialization

The IPython default console can be started using:

```
$ ipython
```

Other options are the Qt based console and the notebook mode:

```
$ ipython qtconsole
$ ipython notebook
```

The `-pylab` option loads Matplotlib and NumPy for interactive use:

```
$ ipython qtconsole --pylab
$ ipython notebook --pylab
```

To enable plotting in the browser for notebooks or inside the graphical window for the Qt console (inline plotting), users can execute:

```
$ ipython notebook --pylab=inline
$ ipython qtconsole --pylab=inline
```

References

- [1] P. Vandewalle, J. Kovacevic, and M. Vetterli, “Reproducible research in signal processing,” *Signal Processing Magazine, IEEE*, vol. 26, no. 3, pp. 37–47, May 2009.
- [2] P. Greenfield, T. Miller, J.-C. Hsu, and R. L. White, “An Array Module for Python,” in *Astronomical Data Analysis Software and Systems XI, ASP Conference Proceedings*, D. A. Bohlender, D. Durand, and T. H. Handley, Eds., vol. 281, San Francisco, 2002, pp. 140–143.

¹⁴See <https://pypi.python.org/pypi/pip>

- [3] T. E. Oliphant, “Python for Scientific Computing,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.
- [4] S. van der Walt, S. C. Colbert, and G. Varoquaux, “The NumPy Array: A Structure for Efficient Numerical Computation,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, Mar. 2011.
- [5] S. J. D. Prince, *Computer Vision: Models, Learning and Inference*. Cambridge University Press, 2012.
- [6] F. Pérez and B. E. Granger, “IPython: A System for Interactive Scientific Computing,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.
- [7] C. Davidson-Pilon, *Probabilistic Programming and Bayesian Methods for Hackers*, 2013.
- [8] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st ed. O’Reilly Media, Oct. 2008.
- [9] R. Laganière, *OpenCV 2 Computer Vision Application Programming Cookbook: Over 50 Recipes to Master this Library of Programming Functions for Real-Time Computer Vision*. Packt Publishing, 2011.
- [10] D. L. Baggio, S. Emami, D. M. Escrivá, K. Ievgenm, N. Mahmood, J. Saragih, and R. Shilkrot, *Mastering OpenCV with Practical Computer Vision Projects*. Packt Publishing, 2012.
- [11] M. Marengoni and D. Stringhini, “Tutorial: Introdução à Visão Computacional usando OpenCV,” *Revista de Informática Teórica e Aplicada - RITA*, vol. 16, no. 1, pp. 125–160, 2009.
- [12] K. Pulli, A. Baksheev, K. Korniyakov, and V. Eruhimov, “Real-Time Computer Vision with OpenCV,” *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, Jun. 2012.
- [13] *OpenCV 2.4.9.0 documentation*. [Online]. Available: <http://docs.opencv.org>
- [14] K. Bache and M. Lichman, “UCI machine learning repository,” 2013.
- [15] J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.
- [16] J. Shi and C. Tomasi, “Good-Features to Track,” in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*. IEEE Comput. Soc. Press, 1994, pp. 593–600.

- [17] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.
- [18] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-Up Robust Features (SURF),” *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346–359, Jun. 2008.
- [19] T. Tuytelaars and K. Mikolajczyk, “Local Invariant Feature Detectors: A Survey,” *Foundations and Trends in Computer Graphics and Vision*, vol. 3, no. 3, pp. 177–280, 2007.
- [20] N. Snavely, S. Seitz, and R. Szeliski, “Modeling the World from Internet Photo Collections,” *International Journal of Computer Vision*, vol. 80, no. 2, pp. 189–210, Nov. 2008.
- [21] S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, “A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms,” in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 1, June 2006, pp. 519–528.
- [22] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, Apr. 2004.
- [23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [24] *Documentation of scikit-learn 0.15*. [Online]. Available: <http://scikit-learn.org/stable/documentation.html>
- [25] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer New York, 2006, vol. 1.
- [26] Y. Cheng, “Mean shift, mode seeking, and clustering,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 790–799, 1995.
- [27] D. Comaniciu and P. Meer, “Mean shift: a robust approach toward feature space analysis,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 5, pp. 603–619, May 2002.
- [28] D. Martin, C. Fowlkes, D. Tal, and J. Malik, “A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics,” in *Proc. 8th Int’l Conf. Computer Vision*, vol. 2, July 2001, pp. 416–423.

- [29] C. Stauffer and W. Grimson, “Adaptive background mixture models for real-time tracking,” in *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, vol. 2. Fort Collins, CO, USA: IEEE Computer Society, Aug. 1999, Conference proceedings (article), pp. 246–252.
- [30] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers, “Wallflower: principles and practice of background maintenance,” in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 1, 1999, Conference proceedings (whole), pp. 255–261 vol.1.
- [31] R. C. Fisher, “CAVIAR: Context Aware Vision using Image-Based Active Recognition.” [Online]. Available: <http://homepages.inf.ed.ac.uk/rbf/CAVIAR/>
- [32] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, “Cython: The Best of Both Worlds,” *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, Mar. 2011.
- [33] D. Scaramuzza and F. Fraundorfer, “Visual Odometry [Tutorial],” *IEEE Robotics & Automation Magazine*, vol. 18, no. 4, pp. 80–92, Dec. 2011.
- [34] D. Nister, “An efficient solution to the five-point relative pose problem,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 6, pp. 756–770, Jun. 2004.
- [35] H. Shen, “Interactive Notebooks: Sharing the Code,” *Nature*, vol. 515, no. 7525, pp. 151–152, Nov. 2014.
- [36] J. M. Perkel, “Programming: Pick up Python,” *Nature*, vol. 518, pp. 125–126, 2015.
- [37] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio, “Pylearn2: a machine learning research library,” p. 9, Aug. 2013.
- [38] J. Bergstra, O. Breuleux, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-farley, and Y. Bengio, “Theano: A CPU and GPU Math Compiler in Python,” in *Proceedings of the 9th Python in Science Conference (SCIPY 2010)*, 2010, pp. 1–7.