A Tutorial Introduction to the Gamera Framework*

Christoph Dalitz
Hochschule Niederrhein, Fachbereich Elektrotechnik und Informatik
Reinarzstr. 49, 47805 Krefeld, Germany

Version 1.5, 20. Nov 2012

Abstract

The *Gamera framework* is a Python library for building custom applications for document analysis and recognition. Additionally, it allows for custom extensions. While its online documentation is an indispensable reference manual when working with Gamera, a beginner usually has trouble finding his or her way through it. This tutorial hopes to bridge the gap by providing a kind of terse text book on Gamera including exercises explaining the most common tasks.

Contents

1 Overview

Gamera [1] can be used for a wide variety of tasks, from building complete image recognition systems down to implementing and evaluating particular algorithms for image processing or document layout analysis. Depending on your goal, you will typically do one of the following:

- *use* the Gamera library. This typically means to write Python scripts or -to a lesser extent- to use the interactive Gamera GUI.
- *extend* the Gamera library. This typically means to write a "toolkit", which can include custom "plugins" and other stuff.

The Gamera framework uses the following terms in a specific meaning:

Plugin Image processing methods are called *plugins* because Gamera uses a general interface for adding custom image methods. This interface is also used by the built in image methods, so that even these methods are technically "plugins".

Toolkit A *toolkit* is an optionally installable addon library for Gamera. This can be useful for distributing your code or for separating the code of your self written plugins from the code of the Gamera core distribution.

Classifier The recognition of individual symbols is done by a *classifier*. The term "classifier" stems from the fact that it takes a symbol and assigns it to a "class" (like "lower case a").

1.1 Using Gamera

To use Gamera interactively, start it from the command line with the command *gamera_gui &* (the optional final ampersand starts the program in the background so that the current terminal is not blocked for further input). You can then load an image with "File/Open image..." and operate image processing routines on the image by right clicking on its icon. Moreover, you can directly enter Python code in the Python shell on the right. As all equivalent commands invoked by the right click menu items are echoed in the right subwindow, this is a simple way to learn how particular methods are called in a Python script. The most important use case for the GUI is the training of symbols before classification.

In most cases, you may want to write a script that does the processing steps automatically, rather than doing them all one by one in the interactive GUI. To use the functions provided by Gamera, you must first import its library in your python script:

```
from gamera.core import *
init_gamera()
```

Make sure that you do not name your script "gamera.py"! This is an as common pitfall, like the error almost every C programming novice runs into by naming his first program *test*[1]. An introduction to working with images in a Python script is given in section 2.

1.2 Extending Gamera

The most common need to extend Gamera is the implementation of additional *plugins*. As pixel access is quite slow from the Python side, this typically requires the implementation of the plugins in C++.

[1] *test* is a shell builtin, so the command "test" might do anything but running the program.

Moreover, to keep your own code separate from the Gamera core, it is generally a good idea to collect all of your custom plugins in a *toolkit*. Both aspects are described in section 3.

2 Image Processing on the Python Side

2.1 Image creation

The image constructor

```
Image(Point ul, Point lr, pixeltype)
```

allocates memory and initializes all pixel values to white. *ul* means the "upper left" (usually $(0, 0)$) and *lr* the "lower right" point. *pixeltype* can be one of RGB, GREYSCALE or ONEBIT (default). Example:

```
# create an 11x11 color image
Image(Point(0,0), Point(10,10), RGB)
```

Note that the alternative constructor *Image(otherimage)* creates an image of the same size and pixel type as *otherimage*, but does not copy its content. To copy an image use the method *image_copy*, e.g.

```
img2 = img1.image_copy()
```

Important image properties are[2]

- *ncols* and *nrows* for the number of columns and rows, respectively. This means that $0 \leq x \leq ncols - 1$ and $0 \leq y \leq nrows - 1$.
- *data.pixel_type* for the pixel type (RGB, GREYSCALE or ONEBIT)

In most cases, images are not created from scratch, but are loaded from files with the *load_image* function, e.g.

```
img = load_image("file1.png")
```

The *load_image* function currently supports PNG and TIFF images. For writing images to files, use the *save_PNG* and *save_tiff* image method, e.g.

```
img.save_PNG("file2.png")
```

2.2 Pixel access and image methods

The value of individual pixels is obtained with the method *get(Point(x,y))* or *get([x,y])*, as in the following example:

```
# count the number of black pixels in a Onebit image
n = 0
for x in range(img.ncols):
    for y in range(img.nrows):
        n += img.get([x,y])
```

Individual pixels can be set with the method *set(Point(x,y), pixelvalue)* or *set([x,y], pixelvalue)*. Depending on the pixel type of the image, *pixelvalue* is

[2] On the Python side, these are indeed *properties* (and not methods), which means that they are to be used without parentheses.

- 0 or 1 for onebit images (0 = white, 1 = black)
- 0 to 255 for greyscale images (0 = black, 255 = white)
- `RGBPixel(r,g,b)` with $0 \leq r, g, b \leq 255$ for RGB color images (r = red value, g = green value, b = blue value)

Here is an example:

```
# write an 11x11 image with a red point in its center
img = Image(Point(0,0), Point(10,10), RGB)
img.set([5,5], RGBPixel(255,0,0))
img.save_PNG("out.png")
```

All image methods are documented under "Reference/Plugins" in the online documentation. Of particular interest are the plugins for conversion between the different image types: *to_greyscale*, *to_rgb*, and *to_onebit*[3]. The following code reads an image file and converts it to onebit, if necessary:

```
img = load_image("file.png")
if img.data.pixel_type != ONEBIT:
   img = img.to_onebit()
```

2.3 Image views

Gamera uses a "shared data" model where the same data can be accessed through different "views". This means that the data type *Image* is actually a *view* where the underlying data can be accessed through its property *data* (like the property *data.pixel_type* in the previous section). This has a number of advantages:

- images are light weight objects that can even be passed by value
- the same data can be represented differently (e.g., as CC or onebit image)
- subimages can be created and accessed without new memory allocation and copying

Subimages containing a subregion of image *img* are created with

```
SubImage(Image img, Point ul, Point lr)
```

where *ul* means the "upper left" and *lr* the "lower right" point of the subimage. Important properties of image views are

- *data* = the underlying image data
- *offset_x*, *offset_y* = displacement of origin with respect to the underlying data

How the values of the view and its data can differ is demonstrated in the following snapshot from the Python shell in the Gamera GUI:

```
>>> img1 = Image(Point(0,0), Point(50,50))
>>> img2 = SubImage(img1, Point(5,5), Point(10,10))
>>> img2.offset_x
5
>>> img2.ncols
```

[3]Conversion to onebit is a nontrivial task for which a wide variety of algorithms can be used. The *to_onebit* method uses global Otsu thresholding [3]. If this does not work for your image, try one of the other plugins in the categories "Binarization" and "Thresholding". A decent and robust solution for varying illumination is *shading_subtraction*.

```
6
>>> img2.data.ncols
51
```

2.4 Special operations for onebit images

While there are a great number of plugin functions for greyscale and color images, Gamera is particularly suited for dealing with onebit images. This does not mean that the input images need to be onebit images, but in document analysis the input images are typically binarized at one point and subsequent operations all work on the resulting onebit images. This section explains a number of important concepts and functions.

2.4.1 Combining onebit images

Images of the same size can be combined pixelwise:

$$h(x, y) = f(x, y) \otimes g(x, y) \quad \text{for all } x, y$$

where $\otimes$ denotes a logical or arithmetic operation. The corresponding plugin functions in Gamera are

- logical operations: *and_image*, *or_image*, and *xor_image*
- arithmetic operations: *add_images*, *subtract_images*, and *multiply_images*

The result on two sample images is shown in Fig. 1. Obviously, we have for onebit images that *or* $\equiv$ *add* and *and* $\equiv$ *multiply*.

When the images are of different size, it is generally undefined how these images should be combined. It is nevertheless possible to combine such images with the following simple trick:

- create a subimage of the larger image at the position that shall be combined with the smaller image
- combine the subimage with the smaller image while setting the optional second parameter *in_place = True*

```
# let a be a 5x5 image and b a 3x3 image
c = a.subimage(Point(2,2), Point(4,4))
c.xor_image(b, in_place=True)
```

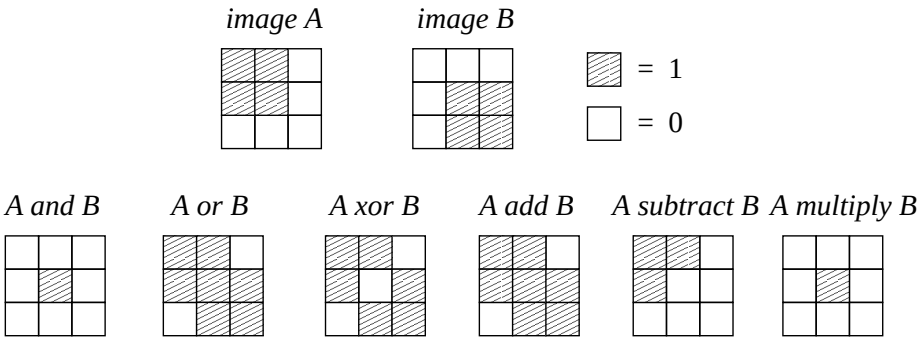

Figure 1: Demonstration of pixelwise operations.

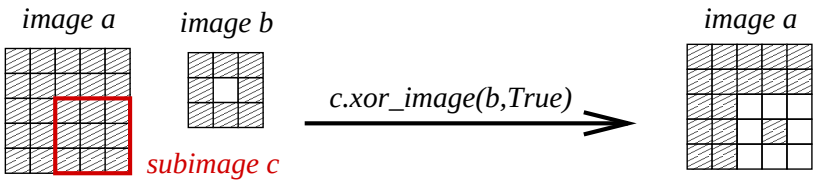

Figure 2: “In place” combination of differently sized images.

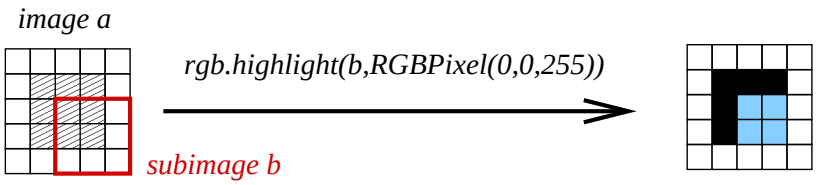

Figure 3: Highlighting the black pixels of only a subregion.

When the parameter *in_place* is *True*, the resulting image is not returned, but is written in the example above to *c*, which shares its data with *a*, so that the original image *a* is changed (see Fig. 2). If this is not what you want, use *image_copy* beforehand.

2.4.2 Color highlighting

For visualization purposes, it is often useful to mark by color all pixels of a given onebit image in a second different image. This can be done with the RGB image method *highlight(onebitimage, pixelvalue)*, as in the following example:

```
# let a and b be of the same size
# mark all pixels red that are black in a, but not in b
c = a.subtract_images(b)
rgb = a.to_rgb()
rgb.highlight(c, RGBPixel(255,0,0))
```

highlight also works with subimages, as in the following example (see Fig. 3):

```
# mark all black pixels in a subregion of image a blue
b = a.subimage(Point(2,2),Point(4,4))
rgb = a.to_rgb()
rgb.highlight(b, RGBPixel(0,0,255))
```

The most important use case of this feature is the highlighting of particular connected components (see section 2.4.4).

2.4.3 Projections and runlegths

An important tool in document analysis are *projections*, that is simply the count of black pixels per row or column. This “projects” the two dimensional image $f(x, y)$ onto a one dimensional list of projection values:

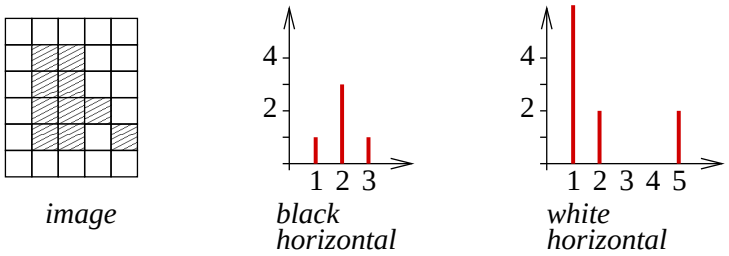

Figure 4: An example image and two of its runlength histograms.

- the image method *projection_rows* computes the sum over each row, or the *horizontal* projection

$$p_{hor}(y) = \sum_{x=0}^{ncols-1} f(x,y)$$

- *projection_cols* computes the sum over each column, or the *vertical* projection

$$p_{ver}(x) = \sum_{y=0}^{nrows-1} f(x,y)$$

Projections can be useful for page segmentation, e.g. to detect the gaps between adjacent text lines.

Another important concept are *runlengths*, that is the number of subsequent pixels of the same color. "Subsequent" means that they are adjacent either in the horizontal or vertical direction. For onebit images, we have two colors and two directions, resulting in four different type of runlengths: black horizontal, etc.

When we count the frequency of each runlength in the image, we obtain its *runlength histogram*. Examples for runlength histograms can be seen in Fig. 4 (make sure you understand this example!). In Gamera, the code

```
p = img.run_histogram(color, direction)
```

returns the runlength histogram as a list where $p[n]$ is the frequency of the runlength of n pixels. *color* can be "black" or "white", and *direction* can be "vertical" or "horizontal".

There are also methods for removing runlengths below or above a given threshold:

```
img.filter_xxx_runs(length, color)
```

where *color* can be "white" or "black", *length* is the threshold, and *xxx* specifies which runlengths are to be removed:

- *xxx* = narrow: remove all horizontal runlength less than *length*
- *xxx* = short: remove all vertical runlength less than *length*
- *xxx* = wide: remove all horizontal runlength greater than *length*
- *xxx* = tall: remove all vertical runlength greater than *length*

Note that all these plugins do not return the result image, but operate directly on the input image.

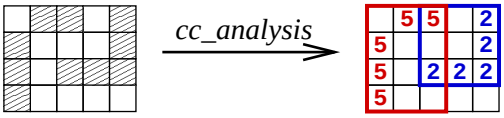

Figure 5: *cc_analysis* replaces the black pixel values with unique labels for each CC.

2.4.4 Connected components

A *connected components* (CC) is a connected[4] set of black pixels. Fig. 5 shows an image with two CCs. CCs are very important in document analysis because they roughly correspond to characters. The image method *cc_analysis* returns a list of images, each of which is a subimage containing only the individual CC. Here is a usage example:

```
# remove all CCs from "img" that are smaller than 2x2
# additionally create an image "rgb" with the removed CCs marked red
rgb = img.to_rgb()
ccs = img.cc_analysis()
for c in ccs:
    if c.nrows < 3 and c.ncols < 3:
        rgb.highlight(c, RGBPixel(255,0,0))
        c.fill_white() # removes the CC on "img"
```

The method *cc_analysis* does not only return a list of CCs, but changes the input image by setting the values of all pixels belonging to the same CC to a unique *label*. This means that "onebit images" actually can have other pixel values than 0 and 1. Methods working on onebit images therefore consider all non zero pixel values as "black".

The example in Fig. 5 shows why this labeling is necessary. In Gamera, CCs are rectangular subimage views (the rectangle is the closest *bounding box* around the CC) which poses problems when the rectangles of different CCs overlap. The labeling helps to distinguish the pixels belonging to the actual CC within the bounding box from pixels belonging to other CCs. Therefore, the subimages returned by *cc_analysis* are not simply of data type *Subimage* but of data type *Cc*. The type *Cc* is derived from *Subimage* and has an additional property *label*. When a onebit image method is applied to a *Cc*, it only affects the pixels with the same value as *Cc.label*.

Exercises for Section 2

Exercise 2.1 Write a script that creates a 20×20 RGB image and writes it to a file *out.png*. Draw two crossing green diagonals into the image,

a) by using only the methods *get* and *set* in a loop.

b) by using the plugin function *draw_line* (see section "Draw" in the plugin online reference).

Exercise 2.2 Write a script that computes the runlength histogram of an image of your choice and writes it into a control file *runs.dat* for the plotting program *gnuplot*[5]. The control files must have the following form:

[4] Gamera assumes 8-connectivity, that is, each pixel has eight neighbors.

[5] *gnuplot* is shipped with all Linux distributions and is also freely available for MacOS X and Windows [4].

```
set xrange [0:30]   # optional for setting xrange
plot '-' with impulses title 'black horizontal runs'
0 0
1 40
...
e
```

where the first column is the runlength and the second its frequency. You can then display the plot with *gnuplot -persist runs.dat*. Hints:

- Have a look at the method *runlength_histogram* in the plugin reference. The parameters are passed as strings.
- To iterate simultaneously over an index and its list value, you can use the Python iterator *enumerate* [2].

Extend your script such that it accepts command line parameters determining whether black/white or horizontal/vertical runs shall be counted (look for the *argv* variable in [2]).

Exercise 2.3 Write a script that measures the most frequent black vertical runlength of an image (see the section "Runlength" in the plugin reference of the Gamera online doc) and creates an RGB image with all black vertical runs of the most frequent runlength marked in red.

Hints: You must try to create an image that only contains the runlengths that are to be marked, so that you can use *highlight* to mark them. All black vertical runlengths of length n are obtained by subtracting all runlengths smaller than n (*filter_tall_runs*) and all runlegths greater than n (*filter_short_runs*). Beware that these methods work in place on the image. This means that you must copy the image beforehand (*image_copy*).

Apply your script to a music score. What do you observe?

Exercise 2.4 Do a *cc_analysis* on an image and list all used labels in sorted order. Are the labels consecutive or are there gaps?

Hint: You can read out all labels elegantly via a Python "list comprehension" [2]:

```
labels = [c.label for c in ccs]
```

3 Image Processing on the C++ Side

When you write custom image methods that access individual pixels, it is a good idea to implement them in C++ rather than Python for performance reasons. While it is theoretically possible to directly add your code to the core code of Gamera, it is much more reasonable to collect your own plugins in a *toolkit*. This does not only reduce the compilation time for your plugins considerably, but it also lets you keep your code independent from the Gamera core code.

3.1 Organizing your code in a toolkit

A first introduction to Gamera toolkits is the HowTo "Writing Gamera toolkits" in the Gamera online documentation. To get started with writing plugins, do the following:

- Download the skeleton toolkit, unpack it and rename it to a name of your choice (let us assume that this name be *myplugins*):

```
tar xzf skeleton-version.tar.gz
mv skeleton-version myplugins
cd myplugins
python rename.py myplugins
```

Here *version* stands for the version number of the skeleton toolkit. If you choose a different name for your toolkit, make sure that your name is a valid Python identifier; in particular it may not contain hyphens or dots!

- Change the category of the demo plugin *clear* in the file

  ```
  gamera/toolkits/myplugins/plugins/clear.py
  ```

  from "Draw" to something different, e.g. "My Plugins". This defines the section of the plugin in the image right click menu of the Gamera GUI.

- Compile and install the toolkit with

  ```
  python setup.py build && sudo python setup.py install
  ```

  When compiling the toolkit under Windows, make sure that you use the same compiler that was used for building Gamera. To this end, it is generally necessary to compile and install Gamera from the sources and not to rely on a Gamera binary install!

To have access to plugins defined in your toolkit in the Gamera GUI, you must import it over the "Toolkits" main menu in *gamera_gui*. To have access to your plugins in a script, import your toolkit with[6]

```
from gamera.toolkits.myplugins import *
```

In Python lingo, a Gamera toolkit is not a module, but a *package*, i.e. a collection of python modules. This means that the above statement does nothing but to execute the file *__init__.py* in the directory *gamera.toolkits.myplugins*. There are different ways to let this actually load the plugins defined in your toolkit (see [2]). The skeleton toolkit does this by directly importing the module *clear.py* with the following line in *gamera/toolkits/myplugins/__init__.py*

```
from gamera.toolkits.myplugins.plugins import clear
```

An alternative method is as follows. Let us assume you have written some plugins in the file *gamera.toolkits.myplugins/plugins/bla.py*. To import them all automatically with the import of your toolkit, do the following:

- Replace the above line in *gamera/toolkits/myplugins/__init__.py* with

  ```
  import plugins
  ```

  This loads the file *__init__.py* in the subdirectory *plugins*.

- In the latter file *plugins/__init__.py*, a plugin module named *bla.py* can be loaded with

  ```
  import bla
  ```

  This will load all image methods defined in the module, but no free functions that are not image methods. If you also want to load them by default, use the following line instead:

  ```
  from bla import *
  ```

[6] Alternatively, you can of course directly import only specific plugin functions from your toolkit with the usual Python ways for importing modules. This includes the possibility to import plugins into a different than the public namespace. See your Python documentation for details, e.g. the chapter "Modules and Packages" in [2].

3.2 Writing C++ plugins

The definitive guide for writing your own C++ plugins are the HowTos “Writing Gamera plugins”, “Specifying arguments for plugin generation” and “The Gamera C++ Image API” in the Gamera online documentation. The best way to start learning writing plugins is by reading these HowTos!

To get a safe base from which to start, try to copy the example plugin *volume* from the HowTo “Writing Gamera plugins” into your toolkit:

- Create the files *example.py* and *example.hpp* at the appropriate locations in your toolkit and copy the code from the HowTo therein.
  Beware: To avoid unnecessary trouble, rename the plugin function from “volume” to “countvolume” in both files, because otherwise it conflicts with the *volume* plugin of the Gamera core!
- Make sure that the new module *example.py* is imported in *__init__.py*, as described in the preceding section.
- Compile your toolkit again and install it with

  ```
  python setup.py build && sudo python setup.py install
  ```

- Test your toolkit with the following simple script (the result should be around 0.09):

  ```
  from gamera.core import *
  init_gamera()
  from gamera.toolkits.myplugins import *
  img = Image(Point(0,0), Point(9,9))
  img.draw_filled_rect(Point(3,3), Point(5,5), 1)
  print img.countvolume()
  ```

The code for this simple plugin *countvolume* already shows a number of peculiarities of C++ plugins:

- What is an image *property* on the Python side, is typically a *method* on the C++ side (see e.g. *ncols*).
- On the C++ size the image data type is templatized. The data type(s) is (are) specified instead in the Python wrapper file in the parameter *self_type*.

The heavy use of templates can lead to very weird error messages during compilation, so make sure that you first get the interface of a new plugin right by compiling a dummy implementation with the planned interface, before you fill your new plugin with algorithmic content.

3.2.1 Returning images from plugins

While the above example plugin poses no problem because the managing of the image types is done by the Gamera interface, it gets more intricate when you want to return an image, which means that you must yourself create an image in your code with a data type that matches the Gamera interface.

You must always return the image *view* and never the underlying *data*. Once the image view is returned, Python takes care of the memory management with its garbage collection. If you need temporary images in your plugin that are not returned to Python, make sure that you delete them, both data and view.

In the simplest case, you want to return an image of the same type as the input image. This can be done with the “ImageFactories” described in the HowTo “The Gamera C++ Image API” as follows:

```
template<class T>
Image* myplugin(const T &src) {
    typedef typename ImageFactory<T>::data_type data_type;
    typedef typename ImageFactory<T>::view_type view_type;
    data_type* dest_data = new data_type(src.size(), src.origin());
    view_type* dest = new view_type(*dest_data);
    // ...
    return dest;
}
```

If you need to always return a specific data type independent from the input image, you can create it as follows (in this example we create a onebit image):

```
template<class T>
OneBitImageView* myplugin(const T &src) {
    OneBitImageData* dest_data =
        new OneBitImageData(src.size(),src.origin());
    OneBitImageView* dest =
        new OneBitImageView(*dest_data,src.origin(),src.size()
    // ...
    return dest;
}
```

3.2.2 Dealing simultaneously with different image types

C++ plugin functions are template functions, which means that the same function is generated at compile time for all images types specified in the Python interface of your plugin. There are a number of techniques for making your code work simultaneously with different data types:

- The pixel value type of the generic image type *T* of your plugin is given by

  ```
  typedef typename T::value_type value_type;
  ```

- The particular values for white and black (note that 0 is white for onebit images, but black for greyscale images) can be obtained with *white(img)* and *black(img)* where *img* is an image view of the data type in question. Moreover, you can check whether a pixel value is black or white with *is_black* or *is_white*, respectively.

- For storing arithmetic combinations of pixel values, you can use the template *NumericTraits*:

  ```
  typedef typename T::value_type value_type;
  typedef typename NumericTraits<value_type>::Promote sum_type;
  typedef typename NumericTraits<value_type>::RealPromote avg_type;
  ```

  In this example, *sum_type* can hold sums of pixel values and *avg_type* can hold *sum_type*s multiplied with a floating point number.

Unlike on the Python side, there is *no way* to query the pixel type of an image on the C++ side[7]. Therefore, *template specialization*[8] is the way of choice for implementing algorithms that need to distinguish between different image types. Suppose you want implement a plugin that returns an image of the same

[7] See M. Droettboom's emails to the Gamera mailing list from 2004/11/15 for a detailed discussion of this problem [5].
[8] For an introduction to *template specialization*, see e.g. [6].

type as the input image and works the same on onebit and greyscale images, but needs different code on color images. This can be done as follows:

```
// generic version for ONEBIT and GREYSCALE
template<class T>
Image* spectest(const T &src) {
    typedef typename ImageFactory<T>::data_type data_type;
    typedef typename ImageFactory<T>::view_type view_type;
    data_type* dest_data = new data_type(src.size(), src.origin());
    view_type* dest = new view_type(*dest_data);
    // ...
    return dest;
}
// specialized version for RGB images
template<>
Image* spectest(const RGBImageView &src) {
    RGBImageData* dest_data =
        new RGBImageData(src.size(),src.origin());
    RGBImageView* dest =
        new RGBImageView(*dest_data,src.origin(),src.size());
    // ...
    return dest;
}
```

Exercises for Section 3

Exercise 3.1 Write a C++ plugin that flips every second white pixel of a onebit image to black.

- Call the plugin *flip1* and let it change the input image.
- Call the plugin *flip2* and let it return the resulting image.

Exercise 3.2 Write two variants of the plugin *flip2* of the previous exercise that both access the individual pixels with *get(Point(x,y))* and *set* in a loop over x and and y. One variant shall have the x-loop outside and one the y-loop.

Show by measuring the runtimes of both variants that it is always better in Gamera to have the outer loop vary over y and the inner loop over x. You can measure the runtime as follows:

```
import time
t = time.time()
res = img.flip2()
t = time.time() - t
print ("Runtime: %f" % t)
```

Exercise 3.3 Write a C++ plugin that replaces the pixel value at position (x, y) with the average over positions $x - 1$ to $x + 1$. In other words, an image $f(x, y)$ shall be transformed to

$$g(x, y) = \frac{1}{3} \sum_{i=-1}^{+1} f(x + i, y)$$

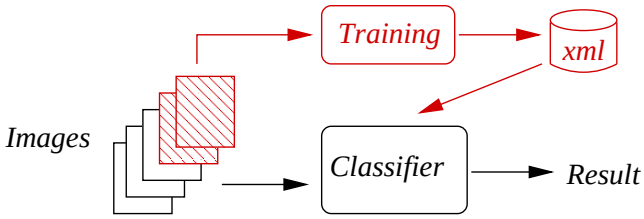

Figure 6: Before symbols can be recognized, a training database has to be created from sample images.

The result shall be returned as a greyscale image. Find a solution for the problem that the values for white and black are exchanged in onebit and greyscale images. Hints:

- The solution consists either in the use of template specialization or a (slightly) tricky utilization of the functions *black* and *white*.
- Use *NumericTraits* for computing the average.
- Make sure that the sum does not go beyond the image borders. Why is it better to take care of the borders in the range of the for-loops rather than doing a range check inside the for-loops?

4 Symbol Recognition

Symbol recognition in Gamera is a two stage operation (see Fig. 6):

a) First you must take some sample images and assign "classes" manually to the occurring symbols. This stage is called *training*. The result of this step is a *training database*, which is stored in an XML file.

b) The sample images in the training database are then used by Gamera's classifier to actually classify symbols passed to it.

The training step is only done once, and the resulting training database can then be used to recognize an arbitrary number of images.

4.1 Training

Training is done over the Gamera GUI and is described exhaustively in the HowTo "Training Tutorial" of the Gamera online documentation. The only tricky part is how to specify broken glyphs as a single symbol, be it deliberately broken characters like lower case "i", or randomly broken characters due to low print or scan quality. For these symbols, the prefix "`_group`." must be added to the class name. They will then be stored in the training database as a combined image with a class name not carrying the "`_group.`" prefix.

4.2 Features and kNN classification

Symbols are differentiated by means of *features*. Consider for instance the characters lower case "a" and "b": these have a different aspect ratio, so the real number *aspect_ratio* can be used for classifying unknown symbols as either "a" or "b". Features are implemented in Gamera as plugins and are documented in the reference "Plugins/Features" in the online documentation. All features, even those consisting of a

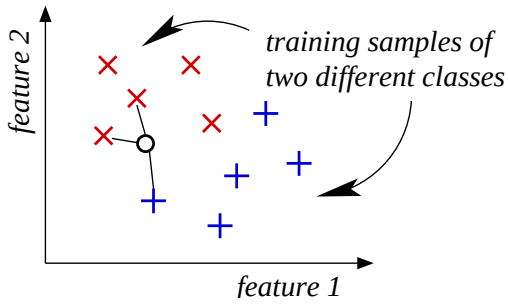

Figure 7: The *kNN* classifier assigns an unknown point in feature space to the majority class among its k nearest neighbors.

single value like *aspect_ratio*, return a floating point vector. So you can query the aspect ratio of a CC with *cc.aspect_ratio()[0]*.

The computation of features from a symbol maps the symbol to a point in *feature space*. The *classifier* then assigns a point in feature space to a specific *class*, thereby utilizing the training database. The most important classification technique built into Gamera is the *kNN classifier*, where "kNN" stands for "k nearest neighbor" [7].

The principle of the kNN classifier is shown in Fig. 7. The red crosses and blue pluses denote the feature space points of training symbols belonging to two different classes. The back circle denotes the feature space point of an unknown symbol. For $k = 3$, it is assigned to the class of the red crosses, because among its $k = 3$ nearest neighbors the majority are red crosses.

During classification, you can choose two parameters: the parameter k in the *kNN* classifier, and the set of features. The choice of k depends on the minimum number of training samples you have for each class. [7] gives some rules of thumb, but in most practical cases this boils down to $k = 1$, unless you have an abundant amount of traing data.

Concerning the set of features, the combination *aspect_ratio*, *moments*, *volume64regions*, and *nrows_feature* has turned out to yield good results for different printed documents in two recent studies ([8], [9]), both with regard to the holdout error rate and a cross correlation error estimate[9]. The "Classifier optimization" described in the HowTo "The Gamera GUI" in the Gamera online documentation is an alternative interesting method to let an automatic algorithm figure out the best feature set on his own, based only on the training data.

4.3 Using the classifier in scripts

For symbol recognition, you can use the class *kNNNonInteractive*, which is defined in the module *knn*. This module is not loaded by the Gamera core, so that you must load it explicitly with

```
from gamera import knn
```

The class *kNNNonInteractive* is described in details in the HowTo "Gamera classifier API" in the Gamera online documentation. To create a kNN classifier, use

```
cknn = knn.kNNNonInteractive([],
          ["aspect_ratio","moments","volume64regions","nrows_feature"],
          0,normalize=True)
```

where the second argument is the list of features. Alternatively, you can later set the features with the method *change_feature_set*. To swith on and off individual components of each feature, you can addi-

[9] In case you are interested in the technical definition of different error estimates, you can find them explained in [7].

tionally use the function *set_selections_by_feature*. The value k defaults to one and is accessible through the property *num_k*. A training data file (e.g. *data.xml*) can be read with the method

```
cknn.from_xml_filename("data.xml")
```

If you have a list of images like the list returned by *cc_analysis*, you can classify all images from the list *imagelist* with

```
cknn.classify_list_automatic(imagelist)
```

For each symbol from *imagelist*, all classes found among its k neighbors in feature space are stored in the property *id_name* of the image. The classes are stored as tuples *(confidence, classname)*, where *classname* is the actual class and *confidence* is some measure for the distance in feature space that can generally be ignored for classification (for more information about confidence computation in Gamera and what it can be used for, see [10]).

The most frequent class among the k nearest neighbors can be queried with the image method *get_main_id*, which simply returns *id_name[0][1]*. Here is a simple use case:

```
ccs = image.cc_analysis()
cknn.classify_list_automatic(ccs)
for c in ccs:
   if c.match_id_name(".*lower.*"):    # regex matching
      print ("lower case letter %s" % c.get_main_id())
```

You can also classify symbols without a classifier with *classify_heuristic*. This can be useful when some symbols shall be classified by decision rules rather than statistical training data, as in the following example:

```
for c in ccs:
   if c.ncols < 5 and c.nrows > 20:
      c.classify_heuristic("vertical_line")
```

classify_list_automatic is only useful when the segmentation of the symbols worked well, because it classifies each symbol individually. When you have deliberately or randomly broken symbols, it is more useful to use *group_list_automatic*, which automatically tries to join broken characters with the algorithm described in [11].

The *kNNNonInteractive* method *group_list_automatic* does not change the input image list, but instead returns the information which symbols are to be merged in two lists *(add, remove)*. You must then use this information to update the image list, theoretically. Practically, there is a convenience function that already does this: *group_and_update_list_automatic* returns the already updated list of images. It is called

```
classifiedimgs = cknn.group_and_update_list_automatic(imglist, \
                 grouping_function, max_parts_per_group=3)
```

where *grouping_function* controls whether two symbols are to be considered as candidates for belonging to the same group. It can be *ShapedGroupingFunction(max_distance)* or *BasicGroupingFunction(max_distance)*, which are defined in the module *gamera.classify*. The grouping algorithm checks a graph of grouping hypotheses, which has an exponential worst case runtime. You therefore should choose small values both for *max_distance* and *max_parts_per_group*.

4.4 Evaluating a classifier

To evaluate how good a classifier recognizes unknown symbols you can segment an image, load it into the classifier GUI and select the menu "Classifier/Guess all" oder "Classifier/Group and guess all". To inspect the classification result of a script, you can save the classified glyphs in your script to an XML file with *glyphs_to_xml*, e.g.

```
result = cknn.group_and_update_list_automatic(ccs, \
             grouping_function=BoundingBoxGroupingFunction(4), \
             max_parts_per_group=3)
gamera_xml.glyphs_to_xml("knn-results.xml",result,False)
```

To inspect the written XML file, you can load it as "page glyphs" into the classifier GUI (menu "File/Page glyphs/Open glyphs into page editor").

The method *evaluate()* for kNN classifiers is an alternative option to automatically evaluate the classifier on the training data alone with the "leave-one-out" method, also known as "n-fold cross-validation" [7]. It returns the recognition rate and is very useful for automatically optimizing certain parameters like k or the chosen feature set.

Exercises for Section 4

Exercise 4.1 Write on a sheet of paper 20 instances of one character (e.g. "a"), and on a second paper 20 instances of a different character (e.g. "b"). Make sure that all characters are a single CC (for easy segmentation with *cc_analysis*) and scan both pages to a raster image.

Plot the distribution of all of your symbols in a two dimensional feature space as follows:

- segment both images into CCs with *cc_analysis*
- read for each CC two feature values of your choice, e.g.

  ```
  feature1 = c.aspect_ratio()[0]
  feature2 = c.moments()[2]
  ```

- write a *Gnuplot* control file *plot.dat* of the form

  ```
  plot '-' with points title 'a', '-' with points title 'b'
  # feature values class a
  0.7692 0.1315
  ...
  e
  # feature values class b
  1.3421 2.9872
  ...
  e
  ```

- display the plot with the command *gnuplot -persist plot.dat*

Exercise 4.2 Use the two images from the preceding exercise to create a training dataset containing five from each of your two characters. Write a script that uses this training dataset to recognize all characters from both images with a *kNNNonInteractive* classifier.

Try different sets of features and values of k and measure the recognition rates for both characters (recogniton rate = number of correctly classified symbols divided by the total number of symbols).

Hint: How a training dataset is built from *several* images can be learnt in the section "On to the second page" of the "Training tutorial" in the Gamera online documentation.

Exercise 4.3 Create a training set from all your handwritten glyphs and use the method *evaluate* to find out the *individual* feature leading to the best recognition rate.

Hint: To change the feature set, use the kNNNonInteractive-method *change_feature_set*.

References

[1] M. Droettboom, C. Dalitz: *The Gamera Homepage.* (2008-2009) `http://gamera.informatik.hsnr.de/`

[2] D.M. Beazley: *Python Essential Reference.* Sams Publishing (third ed., 2006)

[3] N. Otsu: *A threshold selection method from grey level histograms.* IEEE Transactions on Systems, Man, and Cybernetics, vol. 9, pp. 62-66 (1979)

[4] T. Williams, C. Kelly et al.: *Gnuplot version 4.2* (2007). Freely available from `http://www.gnuplot.info/`

[5] Yahoo! Groups: *gamera-devel Message History.* (2004-2009) `http://tech.groups.yahoo.com/group/gamera-devel/`

[6] B. Stroustrup: *The C++ Programming Language.* Addison-Wesley (third ed., 1997)

[7] A. Webb: *Statistical Pattern Recognition.* Wiley-Interscience (second ed., 2002)

[8] C. Dalitz, T. Karsten: *Using the Gamera Framework for building a Lute Tablature Recognition System.* Proceedings ISMIR 2005, pp. 478-481, 2005

[9] C. Dalitz, G.K. Michalakis, C. Pranzas: *Optical Recognition of Psaltic Byzantine Chant Notation.* International Journal of Document Analysis and Recognition, vol. 11, pp. 143-158 (2008)

[10] C. Dalitz: *Reject Options and Confidence Measures for kNN Classifiers.* In C. Dalitz (Ed.): "Document Image Analysis with the Gamera Framework." Schriftenreihe des Fachbereichs Elektrotechnik und Informatik, Hochschule Niederrhein, vol. 8, pp. 16-38, Shaker Verlag (2009)

[11] M. Droettboom: *Correcting broken characters in the recognition of historical printed documents.* Joint Conference on Digital Libraries (JCDL'03), pp. 364-366 (2003)