Gamera: A Python-based Toolkit for Structured Document Recognition

Karl MacMillan

Digital Knowledge Center

Milton S. Eisenhower Library

Johns Hopkins University

karlmac@peabody.jhu.edu

Michael Droettboom and Ichiro Fujinaga

Peabody Conservatory of Music

Johns Hopkins University

{mdboom,ich}@peabody.jhu.edu

Abstract

This paper presents Gamera, a new toolkit for the creation of domain-specific structured document recognition applications by domain experts with limited programming experience. The goal of the Gamera system is to leverage the user's knowledge of the target documents to create custom applications rather than attempting to meet the needs of diverse users with a monolithic application. The system allows a knowledgeable user to combine image processing and recognition tools in an intuitive, interactive, graphical scripting environment based on Python. The use of Python in Gamera creates a simple yet powerful and flexible programming environment for novice programmers. Additionally, the resulting applications are suitable for a large-scale digitization project because they can be run in a batch-processing mode and easily integrated into a digitization framework. Finally, the Python module system has been extended to allow the easy creation of plug-ins using Python or C++.

1 Introduction

The realization that digital libraries can provide increased access and functionality to a wide variety of users has motivated many libraries to undertake large-scale digitization projects. The creation of digital libraries from existing physical collections is a complex and error-prone process, however. The first steps in the creation of a digital library, the generation of digital images from the documents and the collection and organization of associated metadata is a daunting task; the conversion of the resulting digital images with software to a symbolic representation suitable for searching and analysis is even more difficult, particularly in the case of historical or degraded documents. The scale of these projects causes many problems, as the creation and management of large amounts of digital media is a formidable task even with the recent advances in these areas (Witten et al. 1999). Additionally, few of the available tools for document analysis are flexible enough to deal with varied or degraded documents. To address many of these issues, the Lester S. Levy Sheet Music Project at the Milton S. Eisenhower Library at the Johns Hopkins University was started to create an efficient workflow management system for streamlining the creation of digital libraries from existing collections. The workflow system addresses all aspects of the digitization process including the initial image creation, the automatic conversion of the images to a symbolic format, and the implementation of efficient content and metadata searching (Choudhury et al. 2000).

Phase one of the Levy Project created digital images of the 29,000 pieces of popular American sheet music that comprise the Lester S. Levy Collection of Sheet Music. The digitization of this collection, which spans the period of 1780 to 1960, has created a valuable resource for musicologists and historians, which is available at http://levysheetmusic.mse.jhu.edu. The current phase of the project (Phase Two) aims to create a framework of tools for the automatic recognition of the digital images to reduce the time-consuming and costly manual input of the material (Choudhury et al. 2001). The conversion of the Levy collection is being used as a test for the framework.

A key component of this framework is Gamera (http://mambo.peabody.jhu.edu/omr), a Python-based toolkit for document recognition that facilitates the creation of domain-specific document recognition applications by domain experts. Gamera combines the ease-of-use and power of Python with extensions for document analysis. Additionally, an extensive user-interface is included. The system is usable by novice programmers with specific knowledge of a collection of documents to create custom document recognition applications that will offer superior performance to existing, off-the-shelf document recognition systems for many types of documents.

This paper will discuss the motivation and goals of the Gamera system, give an overview of its features, and present details of the architecture. Throughout the paper, the features of Python that give Gamera much of its power and flexibility are highlighted along with the issues raised by using modern C++ features in Python extension modules.

2 Motivation and Goals

From the beginning of Phase Two, one of the most important goals was the development of a flexible OMR system, capable of converting the historical sheet music in the Levy Collection into a symbolic format. To achieve this goal, an adaptive OMR system developed by Ichiro Fujinaga of the Peabody Conservatory of Music at the Johns Hopkins University (Fujinaga 1997) was chosen as the basis for the Levy Project software and expanded with an Optical Music Interpretation system (Droettboom and Fujinaga 2001). OMR is not sufficient to fully analyze the documents in the Levy Collection, however; text is present as score markings, lyrics, and metadata. It was hoped that the text recognition needs of the project could be met with an existing optical character recognition (OCR) system. Unfortunately, testing revealed that this was not a viable solution. The OCR systems tested either performed poorly or were not suitable for the batch-processing approach necessitated by the size of the Levy Collection (Choudhury et al. 2001).

The failure to find an existing OCR system appropriate for the Levy project, alerted us to the general need for flexible document recognition tools suitable for the creation of content for digital libraries. In particular, document recognition tools are generally not available for documents in ancient languages, which contain non-standard printing, or for a variety of reasons differ from common documents. This lack of tools specifically targeted at these types of documents is not likely to change in the future because of the limited market for them. Many of these problems are similar to those solved by the OMR system, however. At the core of the Levy OMR system is a general symbol recognition system capable of learning new symbols. In order to address the OCR needs of the Levy Project and other future projects it was decided to generalize the existing OMR technology to make it suitable for a variety of document recognition tasks, including text and music recognition.

3 Gamera Overview

The system resulting from the generalization of the Levy OMR system, called Gamera, is a toolkit for the creation of domain-specific document recognition systems by document experts. The overall design is inspired by systems like MathWorks Matlab and similar to CVIP tools (Umbaugh 1998), HUE (Cracknell and Downton 1999), and Feature Center (Thibadeau et al. 1995) . Gamera is implemented as a set of extensions to Python written in C++ and Python. The goal is to leverage the user's knowledge of the target documents to create custom applications rather than attempting to meet the needs of diverse users with a monolithic application. Gamera is an easy-to-use environment that allows users without extensive programming experience to be productive with a minimum amount of training. The provided components in Gamera can be combined to create complete document analysis applications, but additional tools and approaches will be useful for certain document types. To address this need, Gamera includes a system for the easy creation of plug-ins by third parties in either C++ or Python.

The specific features of Gamera are:

1. Intuitive environment for the creation of custom document recognition applications.
2. Flexible tools including a learning symbol recognition system.
3. Rich user-interface components for development and training.
4. Suitable for use in large-scale digitization projects.
5. Extensible by third parties with Python and C++.
6. Open-source and standards-based for easy integration with workflow management systems.

3.1 Recognition Process Overview

In order to understand the architecture of Gamera, it is important to be familiar with the document recognition process used. In Gamera, components perform one of five document recognition tasks:

1. Pre-processing
2. Document segmentation and analysis
3. Symbol segmentation and classification
4. Syntactical or semantic analysis

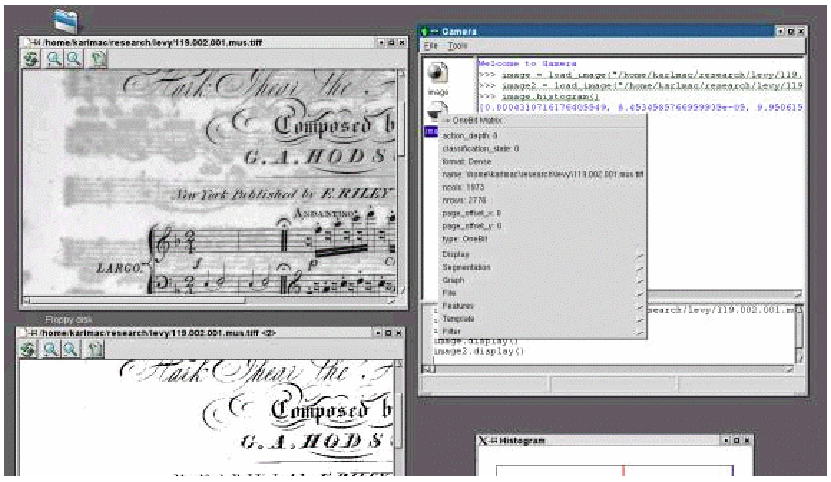
5. Output

Each of these tasks can be arbitrarily complex, involve multiple strategies or components, or be removed entirely depending on the specific recognition problem. Additionally, keeping with the toolbox philosophy of Gamera, the user of the system has access to a range of tools that fall within the general category of these tasks. The actual steps that make up the recognition process are completely controlled by the user.

3.1.1 Pre-processing

Pre-processing can involve almost any standard image-processing operation including noise removal, blurring, de-skewing, contrast adjustment, sharpening, binarization, or morphology. Any number of operations may be necessary to take a raw input image and prepare it for recognition, but the output of this step must be a binary image for the rest of the recognition process.

Many documents, particularly historical documents like those in the Levy collection, will depend on this part of the recognition process to ensure overall good performance of the system. Discoloration of the documents makes binarization difficult and often requires locally-adaptive algorithms (e.g. Trier and Torfinn 1995). Figure 1 shows an image from the Levy Collection, an automatically binarized copy, a histogram of the original image, and the main Gamera window. Additionally, broken lines often cause problems in the segmentation of symbols. Experiments suggest that simple blurring or morphology (Serra 1982) may help with these difficulties.



```
Welcome to Gamera
>>> image = load_image("/home/karlmac/research/levy/119.
>>> image2 = load_image("/home/karlmac/research/levy/119
>>> image.histogram()
[0.000431071617640559, 6.4534589766999935e-05, 9.950615
```

action_depth: 0
classification_state: 0
format: Dense
name: /home/karlmac/research/levy/119.002.001.mus.tiff
ncols: 1973
nrows: 2776
page_offset_x: 0
page_offset_y: 0
type: OneBit

```
search/levy/119.002.001.m
image.display()
image2.display()
```

Figure 1 - Gamera session with an image from the Levy Collection.

3.1.2 Document segmentation and analysis

Before the symbols of a document can be classified, an analysis of the overall structure of the document is often necessary. The document segmentation and analysis process is designed to analyze the overall structure of the document, segment it into sections, and perhaps remove elements (Haralick 1994; Yanikoglu 1998). For example, the proper identification of the staff lines and the grouping of the lines into staves and systems is convenient for the classification of symbols and later to the interpretation of those symbols. Similarly, text documents may require the identification of columns, paragraphs, lines, or tables.

3.1.3 Symbol segmentation and classification

The segmentation and classification of symbols is the core of the Gamera system. The current implementation provides tools for the creation of simple heuristic classifiers, template-based image matching, and a learning classifier using the k-nearest neighbor algorithm (k-NN) enhanced with a genetic algorithm. Other possible classification algorithms include neural-nets, decision trees, or hidden Markov models. The use of both learning and heuristic classifiers allows for the balancing of flexibility, accuracy, training time, and recognition speed.

3.1.4 Syntactical or Semantic analysis

The syntactic and semantic analysis process reconstructs a document into a semantic representation from the individual symbols. Examples include combining stems, flags, and noteheads into musical notes, or grouping words and numbers into a table. Obviously this process is entirely dependent on the type of document being processed and is a likely place for large customizations by knowledgeable users.

3.1.5 Output

Output converts either the raw symbols or the structurally interpreted data into a suitable format for storage.

4 Architecture

As was previously noted, Gamera is implemented as a set of extensions to the Python programming language. These extensions include standard Python modules and special C++ modules for the storage and manipulation of images. The C++ extensions are written following certain conventions and then compiled using the tools supplied by Gamera to produce Python extension modules with Gamera-specific features. The tools for creating C++ extensions in Gamera are based on the excellent Boost Python Library (http://www.boost.org).

To provide the necessary support for the five document recognition tasks, Gamera includes facilities for image storage and manipulation, symbol classification, and syntactic and semantic analysis. Additionally, Gamera includes many facilities for the creation of user-interfaces. The portions of Gamera used for the syntactic and semantic analysis are not discussed in this paper, but are detailed in Droettboom and Fujinaga (2001).

4.1 Image storage and manipulation

The storage and manipulation of images is one of the most important aspects of Gamera. Gamera has to provide not only general-purpose image manipulation functions, but also infrastructure to support the symbol segmentation and analysis. The requirements for the image processing portions of Gamera include:

1. Storage of multiple pixel types, including color, grayscale, and bi-level images.
2. Efficient storage of bi-level images including support for compression.
3. A consistent programming interface for both C++ plug-ins and Python regardless of the pixel or storage type.
4. Flexible and efficient representation of portions of images, including non-rectangular portions.

5. The runtime addition of methods to the Python matrix classes by C++ plug-ins.

To meet these requirements, a set of classes representing images as matrices was created. These classes separate matrices into two components: the matrix data, which are the actual values stored in the matrix, and matrix views, which are lightweight objects that present a matrix interface to all or a portion of the matrix data. The use of matrix views provides a flexible and efficient paradigm for the manipulation of images in the context of recognition.

The classes are implemented in two layers: a C++ layer emphasizing speed and flexibility and a Python layer that presents a consistent interface and handles memory management. The two layers work together to allow the creation of fast, type-safe algorithms in C++ without sacrificing the dynamic type system and automatic memory management of Python.

4.1.1 C++ Matrix Classes

The requirements for the matrix classes are met in C++ using generic programming techniques (Stroustrup 1997). Generic programming techniques in C++ place an emphasis on compile-time type safety and code generation. While this is a good design choice in terms of efficiency, it contrasts strongly with the dynamic, runtime type system of Python. In particular, generic programming in C++ uses templates to allow the flexible creation of specific types at compile time from generic components. For readers unfamiliar with the generic programming features of C++, a brief introduction is provided below.

Containers and algorithms in C++ are defined without type through the use of templates; the specific types for the containers and algorithms is generated at compile time based on the types specified in the declaration. For example, the C++ standard library includes a resizable array container class called `vector`. To use this class, a user declares it with a specific type, e.g. a vector of floats would be declared `vector<float>`. The specific code required to manipulate the `vector<float>` would be created at compile time. Additionally, the code for an algorithm, in the form of templatized functions, is generated at compile time based on the specific types passed as arguments. The template arguments for a function can be either automatically deduced or specified explicitly. Figure 2 shows an example of a generic algorithm and its use with a vector.

```
#include <vector>

template<class T>
void my_algorithm(T& a_vector) {
  // code to manipulate a_vector of type T& would go here.
  // The specific type of T& depends on how the function is called.
}

int main(int argc, char** argv) {
  // create a vector holding floats
  std::vector<float> vec;
  std::vector<int> int_vec;

  // use my_algorithm and let the compiler determine
  // the type of T
  my_algorithm(vec);
  my_algorithm(int_vec);
  // use my_algorithm specifying the type explicitly
  my_algorithm<vector<float> >(vec);
  my_algorithm<vector<int> >(int_vec);
}
```

Figure 2 - A demonstration of generic programming in C++

This type of programming provides both the speed and flexibility necessary for the image storage and manipulation portions of Gamera, but it presents a problem when integrated with Python. A Python binding would require the creation of a specific type for each desired type of `vector`, for example binding for floats and doubles would require the creation of a `vector_float` and `vector_double` class in Python. This creates an interface that is not idiomatic to Python and couples the details of the C++ implementation too closely to Python. Within Gamera, two types of matrix views, one for rectangular views and another for non-rectangular views, with four pixel types are exported to Python, creating eight separate C++ matrix types. Currently, compressed storage is not implemented, but in the future the number of C++ classes may increase to accommodate run-length compressed bi-level images. The next sections details how these types are hidden from the user to create a natural Python interface. Additionally, for each type bound in Python, the templatized functions would have to be explicitly generated and the correct function called based on the type. Section 4.1.3 shows how this is handled in Gamera.

4.1.2 Python Matrix Classes

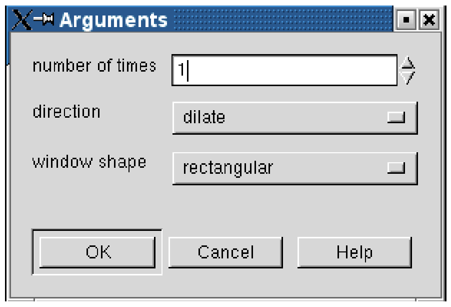
Gamera provides a Python matrix class to hide the details of the C++ types from users. This class holds a reference to a C++ matrix view and contains a variety of methods to manipulate this view. By not directly mapping the C++ types in Python, it is possible for the Python classes to retain the dynamic nature of Python objects and to easily identify objects by a subset of their types. For example, it is possible to inspect the pixel type of the Python matrix classes without worrying about the view or storage type. Additionally, changing the type of a Python matrix object, which means the creation of an entirely new C++ matrix object, can be easily accomplished. This encapsulation insulates the users from many of the implementation details. Finally, each instance of the Python matrix class holds a reference to the matrix data. This allows the standard Python memory management facilities to determine the lifetime of the matrix data.

In addition to providing a more convenient interface, this decoupling of the Python interface from the details of the C++ classes eases maintenance of Gamera. Major revisions have already been made to the C++ matrix classes, including the removal of several matrix types, without any effect on the existing Python code. Though not an initial design goal, it is a welcome side effect.

4.1.3 Plug-in Features

All of the image manipulation in Gamera is done through C++ and Python functions that are automatically added as member functions to the Python matrix classes. Each function is contained in its own Gamera plug-in. These plug-ins, which are standard Python modules that follow certain conventions, provide a variety of features including automatic dialog-box generation and descriptive data about the plug-ins. The next section (4.1.4) also details some of the features specific to C++ plug-ins.

In general, Gamera plug-ins provide image manipulation services through functions that are added to the Python matrices. The function interface is convenient for the creation of stand-alone document recognition scripts, but when used in the graphical development environment it is useful to have a dialog-box to specify function arguments and assign the returned data to variables. To remove the burden of creating these dialog-boxes, Gamera allows the plug-in developer to specify the function argument types and ranges and whether a value is returned in a concise syntax. This information is then used to generate the dialog-box. Figure 3 is an interface generated for one of the morphology plug-ins.

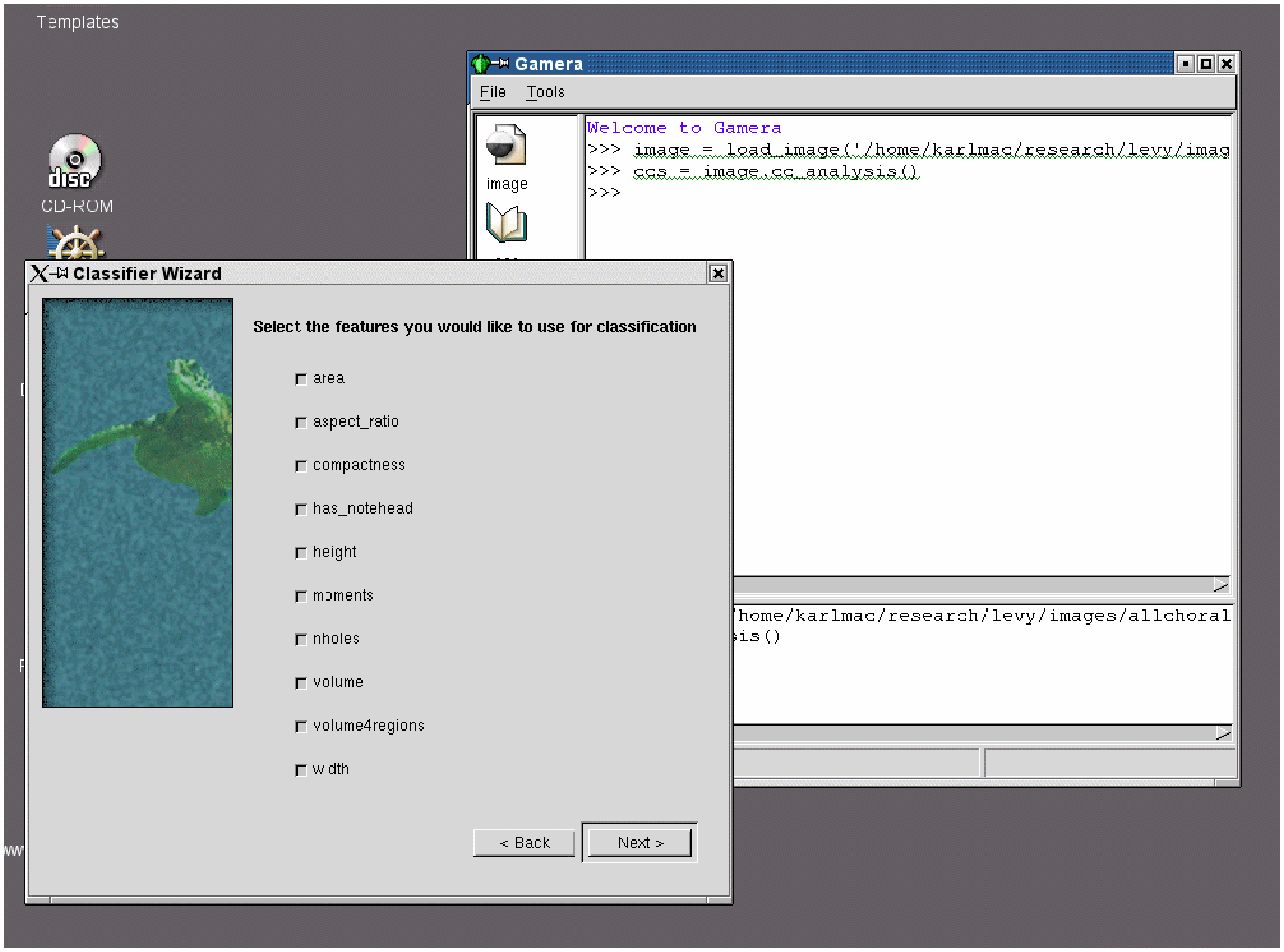


number of times: 1
direction: dilate
window shape: rectangular

```
Args([Int('number of times', range=(0, 10), default=1), Choice('direction', ['dilate', 'erode']),
Choice('window shape', ['rectangular', 'octagonal'])] )
```

Figure 3 - A dialog box automatically generated for a Morphology plug-in and the declaration of the function argument types and ranges.

Plug-ins also export descriptive data, including their category and the matrix types on which they operate. This information is used in a variety of ways, including the user-interface generation described in section 4.3.1 and in automatically choosing feature extraction functions for the classifier. Figure 4 shows the classifier wizard presenting the user with all of the currently available feature extraction plug-ins.



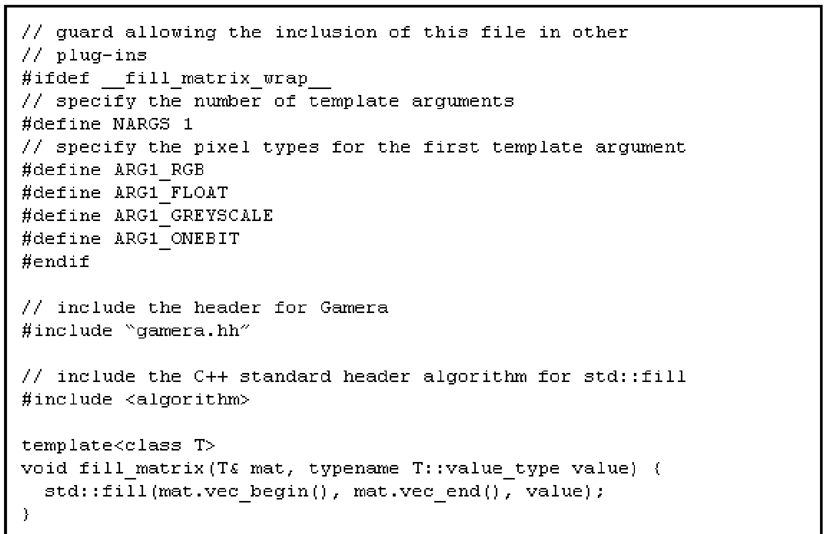
```
Welcome to Gamera
>>> image = load_image('/home/karlmac/research/levy/imag
>>> ccs = image.cc_analysis()
>>>
```

```
/home/karlmac/research/levy/images/allchoral
sis()
```

Figure 4 - The classifier wizard showing all of the available feature extraction plug-ins.

4.1.4 C++ Plug-in Features

Generating and exporting all of the required functions to Python from a C++ plug-in would be a difficult task for a developer. This is especially complex if a function takes several different types of matrices. Additionally, it is necessary to allow the developer to limit the pixel types on which their plug-in will operate, but it is desirable for them to not have to know about the view and storage types (e.g. raw or compressed). To accomplish this, Gamera uses a custom plug-in build system that generates a Python binding based on C++ pre-processor defines in the plug-in. Figure 5 shows a plug-in that fills a matrix with a specified value. At the top of the plug-in, the number of template arguments is specified and the pixel types accepted for each argument is specified. To compile this, a Python script generates a C++ header file that defines several pre-processor symbols with information about the name of the plug-in. This header and the plug-in are included in a file that uses the C++ pre-processor to generate a Python binding using the Boost Python Library.



```
// guard allowing the inclusion of this file in other
// plug-ins
#ifdef __fill_matrix_wrap__
// specify the number of template arguments
#define NARGS 1
// specify the pixel types for the first template argument
#define ARG1_RGB
#define ARG1_FLOAT
#define ARG1_GREYSCALE
#define ARG1_ONEBIT
#endif

// include the header for Gamera
#include "gamera.hh"

// include the C++ standard header algorithm for std::fill
#include <algorithm>

template<class T>
void fill_matrix(T& mat, typename T::value_type value) {
  std::fill(mat.vec_begin(), mat.vec_end(), value);
}
```

Figure 5 - C++ Gamera plug-in.

The use of templatized functions for the manipulation of the C++ matrix classes relies on function overloading to provide a convenient interface. Function overloading is not a natural feature of Python, but it is a fundamental part of C++. Exporting these functions to Python requires the creation of a separate function, with a separate name, for each specific function instance. It is possible to create a Python function that accepts any type of matrix and calls the appropriate C++ function based on the Python type. The Boost Python Library handles this process automatically, allowing the creation of what appears to be overloaded functions in Python. Gamera uses this to dispatch function calls based on the matrix type.

4.2 Symbol Classification

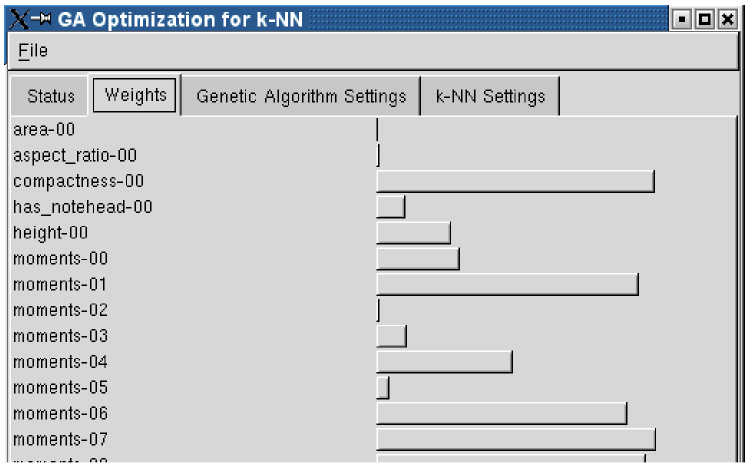
One of the main tasks of the Gamera system is the classification, or identification of symbols. This involves taking an individual symbol, an individual letter from a document for example, and assigning a identification based on its appearance. This portion of the document recognition process is concerned only with the classification of individual symbols based on image information. Other portions of the recognition process consider context, metadata, or other non-image data to further refine the symbol classification.

4.2.1 Classification Overview

The main system currently implemented in Gamera is an exemplar-based classifier. That is, it classifies symbols based on their similarity to known symbols. The use of a classifier capable of learning allows the arbitrary extension of the system by the user to recognize almost any type of symbol. This allows the same classification system to be used on the wide variety of music and text within the Levy collection and almost any other structured document. In addition to simple classification, the system allows the use of heuristic classifiers to be driven by the exemplar-based classifier. That is, it allows the system to learn to apply heuristic classifiers based on the input symbol.

Classification in Gamera is done by an instance of the `Classifier` class. The classifier presents a high-level interface that allows the classification of a list of symbols, the editing of a database of example symbols, or the creation of an example database. Additionally, the `Classifier` class allows the creation and editing of a list of possible symbols, called a symbol table. The symbol table in Gamera allows the categorization of symbols to ease the organization of the possible symbols. For example, in a text document, punctuation could be placed in a category called "punctuation", which would make the symbol for a period "punctuation.period". This classification scheme is convenient for users and allows other more advanced features described later in 4.2.2. The use of this high-level interface not only increases the usability by developers, it also allows the transparent substitution of the underlying classification algorithms. Currently, only one classification algorithm is implemented, but others are possible.

Classification of symbols in Gamera is currently primarily accomplished using a k-NN classifier (Cover and Hart 1967) enhanced with a genetic algorithm (GA) (Holland 1975). Classification of symbols using the k-NN algorithm is a simple process involving the computation of a mathematical description of the image, called a feature-vector, and computing the distance between the feature-vector of the unknown symbol and the entries in a database of known feature-vectors. The classification of the k closest feature vectors from the database is assigned to the unknown feature. The value of k is typically a small integer. The system learns through the addition of entries to the database of known feature-vectors, which can be done by the user.



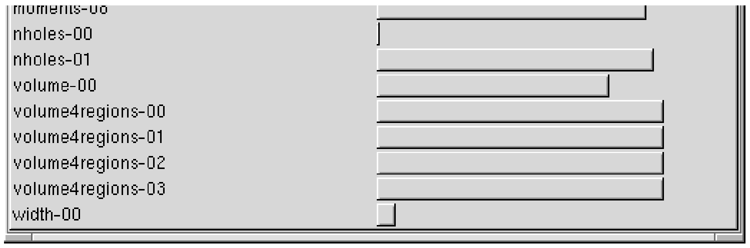


Figure 6 - Biollante optimizing a k-NN database using a genetic algorithm.

The performance of the k-NN classification can be dramatically improved through the use of feature weighting. This involves multiplying the distance along each feature dimension by a floating-point number, or weight. Determining the optimal feature weights, or weight-vector, would be prohibitively expensive computationally if a brute-force search was used. Gamera uses a genetic algorithm to determine the a very good weight-vector in a relative inexpensive manner. Figure 6 shows the application Biollante, an application created with the Gamera toolkit, which performs the GA optimization offline.

In addition to classification using the learning classifier, Gamera can use heuristic, or rule-based, classification either independently or driven by the k-NN classifier. Heuristic classifiers are useful because they can dramatically reduce the amount of training necessary, are often less expensive computationally, and perform better than the learning classifier for some symbols. Heuristic classification can be performed as a pre-processing step to the learning classifier or as part of the classification by the learning classifier. In the second case, the learning classifier can be taught to perform a heuristic classification on a general class of symbols. For example, segmentation of some musical symbols is difficult to perform. To overcome this, Gamera can be taught to recognize the larger musical symbols and perform additional segmentation when they are encountered. Figure 7 shows one of these symbols before and after the exemplar driven heuristic classification.

4.2.2 Classification Details

The k-NN classifier in Gamera is written in C++ and exported to Python using the Boost Python Library. The use of C++ for the classifier is essential because of the computational complexity of the classification process. The GA optimization uses the GALIB C++ library by Matthew Wall (http://lancet.mit.edu/ga) with a small C++ class that adapts it for use with the k-NN library and presents a limited interface suitable for exporting to Python. This class is also safe for use within Python threads, allowing the background optimization of the k-NN database from within Python. The use of Python threads instead of threads within C++ solves many problems, including portability. In general, it was considerably easier to make the C++ code aware of Python threads than to implement the threading in C++.

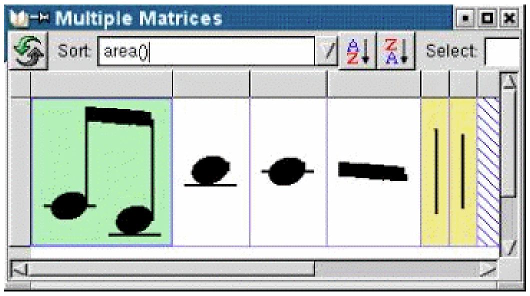


Figure 7 - A musical symbol before and after segmentation by a heuristic classifier.

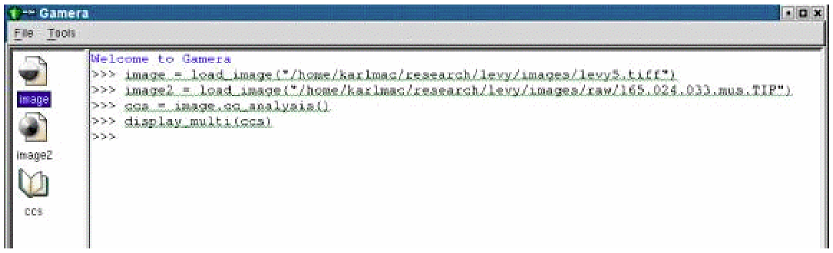
In a similar manner to the image processing portions of Gamera, the computationally intensive portions of the classification are done in efficient C++ and everything else is done in Python. In particular, all threading, file manipulation (which uses the standard Python XML modules), feature extraction, and integration of the learning and heuristic classification is done in Python. This use of Python helps the classification work easily with the rest of Gamera and also saved development time and complexity. This strategy was particularly effective in the integration of the learning and heuristic classification and the implementation of the file handling.

The use of the learning classifier to drive various heuristic classification strategies is one of the most effective features of Gamera and it is greatly aided by the dynamic nature of Python. The use of this technique has existed in previous versions of the OMR software written in C, but the possible heuristic classification strategies was fixed at compile time. The system implemented in Gamera is extensible by the user at runtime and is generally simpler and more robust. In order to use a heuristic classification strategy from the learning classifier, the user trains a symbol as part of the special category “action” with the appropriate method to perform the heuristic classification appended. The method, which accepts a symbol and returns a list of classified symbols, is then called on the symbol whenever the learning classifier classifies a symbol in that class. This simple scheme allows a user to call any heuristic classification method currently available in the system, including those implemented as plug-ins.

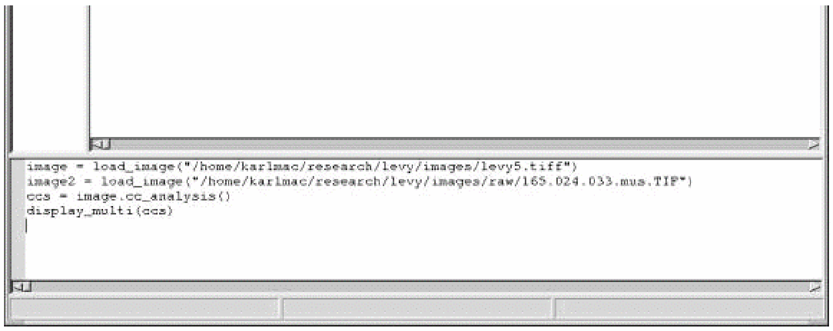
4.3 User Interface

The user-interface elements furnished by Gamera are used to create a development environment and provide an efficient interface for training the learning classifier. The creation of an effective user-interface was one of the most time-consuming aspects of developing Gamera, but it has proven to be the most useful. The wxPython library (http://www.wxpython.org) was chosen as the toolkit for the user-interface because of its portability and the completeness and flexibility of its widgets.

4.3.1 Development Interface



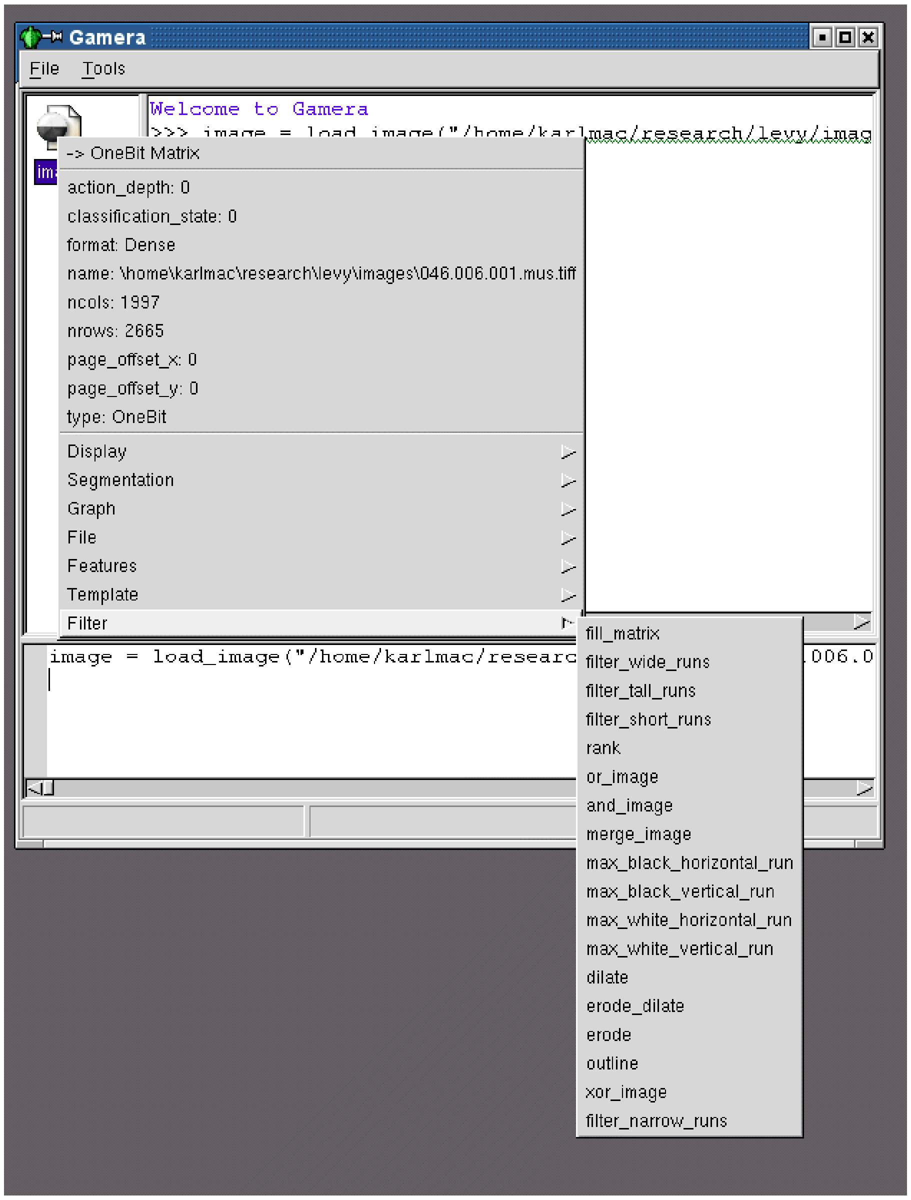
```
Welcome to Gamera
>>> image = load_image("/home/karlmac/research/levy/images/levy5.tiff")
>>> image2 = load_image("/home/karlmac/research/levy/images/raw/165.024.033.mus.TIF")
>>> ccs = image.cc_analysis()
>>> display_multi(ccs)
>>>
```



```
image = load_image("/home/karlmac/research/levy/images/levy5.tiff")
image2 = load_image("/home/karlmac/research/levy/images/raw/165.024.033.mus.TIF")
ccs = image.cc_analysis()
display_multi(ccs)
```

Figure 8 - The main Gamera window.

Gamera provides a graphical development environment that allows the intuitive exploration of recognition strategies. It allows the user to load and manipulate images using either an interactive Python interpreter or a point-and-click interface. Figure 8 shows the main development window. The left-hand pane shows all of the currently active images and lists of images, the top, right-hand pane is the interactive Python interpreter, and the bottom pane is the history of commands from the interpreter. Many modern user-interface functions are implemented to create a convenient and intuitive environment. For example, images can be loaded by dragging them into the left-hand pane. Also, wizards are provided for the creation of complex objects. All of the user-interface conveniences generate the equivalent Python code in the right-hand pane, however, allowing a user to easily transition from the development environment to stand-alone scripts. Figure 4 shows the classifier wizard with the code that it generates displayed in the console.



```
Welcome to Gamera
>>> image = load_image("/home/karlmac/research/levy/imag
```

```
action_depth: 0
classification_state: 0
format: Dense
name: \home\karlmac\research\levy\images\046.006.001.mus.tiff
ncols: 1997
nrows: 2665
page_offset_x: 0
page_offset_y: 0
type: OneBit
```

```
image = load_image("/home/karlmac/researc                     .006.0
```

Figure 9 - An automatically generated menu.

The development interface makes good use of the features of Python to generate many elements dynamically. For example, the list of active images is created by examining the current Python namespace for instances of the matrix objects. Also, Figure 9 shows the menu displayed when an image icon is selected with the right mouse button. The menu is generated by using information about the image type and the available plug-ins. See section 4.1.3 for more information on how the descriptive data about the plug-ins, including the matrix types on which they operate and their category, is automatically exported when the plug-ins are created.

The dynamic user-interface generation helps limit the coupling between the general Gamera code and the user-interface code. This is particularly important because of the need to use the Gamera components in a non-interactive batch mode. If Python did not have rich runtime object inspection capabilities, the user-interface generation would be more difficult.

4.3.2 Training Interface

One of the most difficult and time-consuming tasks involved in using a learning symbol recognition system is supervised training. With this in mind, a training interface was created for Gamera that would be both easy-to-use initially and efficient to use with training and experience. This interface includes facilities for the creation of a symbol table, display of a list of symbols, display of symbols in context when highlighted, sorting, and automatic selection of symbols for sequential classification. The interface can be used in a variety of ways to allow the user to determine the most efficient training method. The same interface can also be used to edit or correct existing databases. Figures 10 and 11 show the training interface.

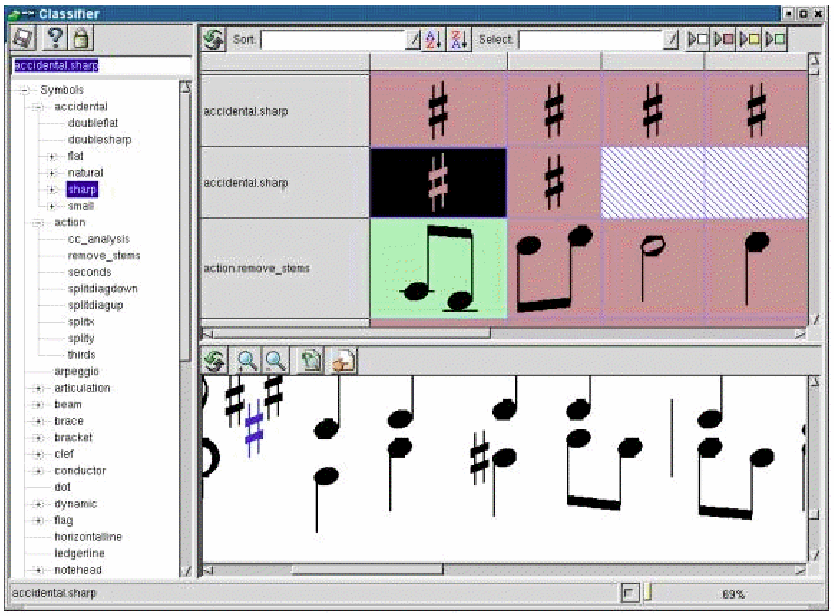


Figure 10 - The training interface being used on music.

In addition to facilities to ease the manual classification of symbols, this interface attempts to leverage the learning system to speed training. To accomplish this, the user can use the information currently in the database to automatically classify all of the current symbols. Once a large database has been created, the training process is reduced to the correction of incorrect classifications. Iteratively training manually and automatically can greatly simplify the training process. Additionally, the classifier can be constantly optimized in the background using a GA. This optimization greatly increases the accuracy of the classifier, substantially reducing the training time.

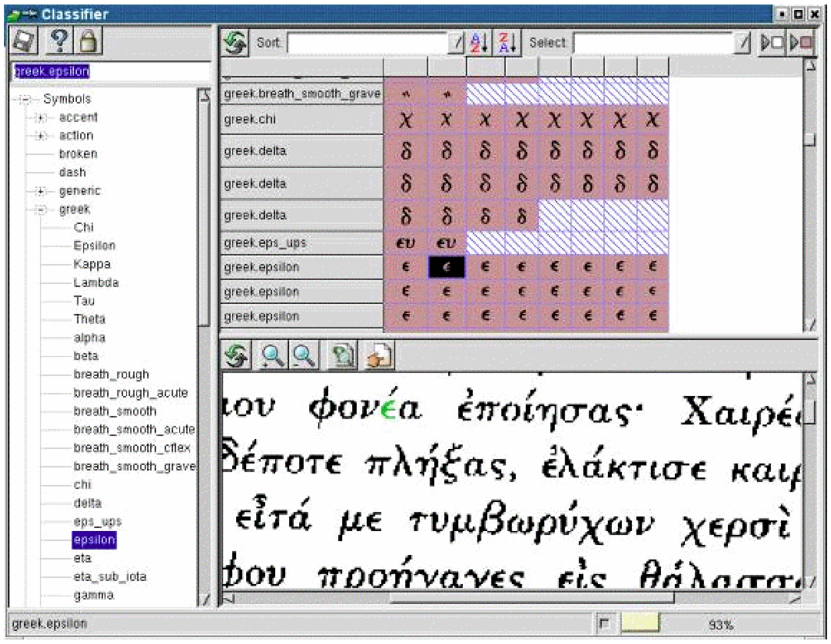


Figure 11 - The training interface being used on Greek text.

5 Conclusion

Gamera is a Python-based toolkit for structured document analysis that allows domain experts to create custom document recognition applications. Gamera leverages the power and flexibility of Python to create an easy-to-use scripting environment that can be used productively by novice programmers. Additionally, a plug-in system is provided for the extension of the system using Python and C++.

Acknowledgements

The Levy Project has received funding from the National Endowment for the Humanities, the National Science Foundation, the Institute for Museum and Library Services, and the Levy family.

Reference

Choudhury, G. S., T. DiLauro, M. Droettboom, I. Fujinaga, and K. MacMillan. 2001. Strike up the score: Deriving searchable and playable digital formats from sheet music. *D-Lib Magazine* 7 (2). http://www.dlib.org/dlib/february01/choudhury/02choudhury.html.

Choudhury, G. S., C. Requardt, I. Fujinaga, T. DiLauro, E. W. Brown, J. W. Warner, and B. Harrington. 2000. Digital workflow management: The Lester S. Levy digitized collection of sheet music. *First Monday* 5 (6). http://firstmonday.org/issues/issue5_6/choudhury/index.html.

Cover, T., and P. Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13 (1): 21–7.

Cracknell, C. and Downton, A. C. 1999. A handwriting understanding environment (HUE) for rapid prototyping in handwriting and document analysis research. In *Proceedings of the 5th International Conference on Document Analysis and Recognition,* 362–5.

Droettboom, M., and I. Fujinaga. 2001. Interpreting the semantics of music notation using an extensible and object-oriented system. In *Proceedings of the Ninth International Python Conference,* 71–85.

Fujinaga, I. 1997. Adaptive optical music recognition. Ph.D. Dissertation. McGill University.

Haralick, R. M. 1994. Document image understanding: Geometric and logical layout. In *Proceedings of the IEEE Computer Society Computer Vision and Pattern Recognition,* 385–90.

Holland, J. H. 1975. Adaptation in natural and artificial systems. Ann Arbor: University of Michigan Press.

Serra, J. P. 1982. Image analysis and mathematical morphology. London: Academic Press.

Stroustrup, B. 1997. The C++ programming language. 3rd ed. Reading, MA: Addison-Wesley.

Thibadeau, R. H., R. Romero, and D. S. Touretzky. 1995. Feature Center: Getting the picture from documents and drawings. Technical Report CMU-RI-TR-95-32. Robotics Institute, Carnegie Mellon University.

Trier, O. D., and T. Torfinn. 1995. Evaluation of binarization methods for document images. In *Proceedings of the IEEE Computer Society Pattern Analysis and Machine Intelligence,* 312–5.

Umbaugh, S. E. 1998. Computer vision and image processing: A practical approach using CVIPtools. Upper Saddle River, NJ: Prentice Hall.

Witten, I., A. Moffat, and T. Bell. 1999. Managing gigabytes. 2nd ed. San Francisco, CA: Morgan Kaufmann.

Yanikoglu, B. A., and L. Vincent. 1998. Pink Panther: A complete environment for ground-truthing and benchmarking document page segmentation. *Pattern Recognition* 31: 1191–204.