# The Imalab Method for Vision Systems

Augustin Lux *

Laboratoire GRAVIR/IMAG
Institut National Polytechnique de Grenoble

**Abstract.** We propose a method to construct computer vision systems using a workbench composed of a multi-faceted toolbox and a general purpose kernel. The toolbox is composed of an open set of library modules. The kernel facilitates incremental dynamic system construction. This method makes it possible to quickly develop and experiment new algorithms, it simplifies the reuse of existing program libraries, and allows to construct a variety of systems to meet particular requirements.
Major strong points of our approach are: (1) Imalab is a homogeneous environment for different types of users, who share the same basic code with different interfaces and tools. (2) Integration facility: modules for various scientific domains, in particular robotics or AI research (e.g. Bayesian reasoning, symbolic learning) can be integrated automatically. (3) Multi-language integration: the C/C++ language and several symbolic programming languages - Lisp(Scheme), Prolog, Clips - are completely integrated. We consider this an important advantage for the implementation of cognitive vision functionalities. (4) Automatic program generation, to make multi-language integration work smoothly. (5) Efficiency: library code runs without overhead.
The Imalab system is in use for several years now, and we have started to distribute it.

## 1 Introduction

We propose to construct computer vision systems using a workbench composed of a large set of reusable modules, and a set of sophisticated tools for system construction, including an interactive programming shell, and a program generator to automatically integrate C++ source code into the shell.

We illustrate this method with the Imalab system, which is a research system combining all major features that are available in the workbench:

- A large set of C++ libraries (some 100 classes with more than thousand methods and functions) makes it easy to experiment algorithms on new images, and to develop new algorithms.
- An interactive shell, with a large subset of C++ as a shell language.

* This work was partially funded by the IST Programme of the European Commission under contract number IST-2001-32157 DETECT

– Homogeneous environment: the same programming language is used for interactive experimentation in the shell, for writing scripts to automate sequences of commands, and to extend the system's native code.
– Automatic library linkage: New libraries written in C++ can be added to the system, provided the header files are available.
– Incremental dynamic system construction, making it extensible according to user needs.

### Comparison with Existing Systems

There is a large spectrum of vision oriented software [**?**]. On one end of the spectrum, one finds libraries with code for image processing operations providing functionalities to be used in an application program. Important examples of libraries are the Image Understanding Environment[**?**] defining a large hierarchy of C++ classes modelizing all data structures necessary in computer vision in a general way, and Intel's Open Source Computer Vision Library[**?**] providing numerous C procedures with code written for efficiency. On the other end of the spectrum, one can find complete systems built around such libraries, containing sophisticated tools for the development of vision applications. One outstanding example of this kind is the Khoros system [**?**][**?**], which includes the *Cantata* visual programming environment [**?**].

The Imalab project lives on a much smaller scale; it places the main emphasis on *interactive development* and *modularity*, it aims to be particularly useful for the development of new algorithms, and for experimentation with new applications. In this respect, we have the same motivations as the authors of the ScilImage system[**?**]: "The immediate feedback interactive systems provide reduces the time needed to develop new applications" and "the ability to see the result of processing and to modify code or parameters within seconds brings new insights, such as how sensitive an algorithm is to a small change in the image, or how parameters should be tuned in response to certain shapes". However, in spite of so similar motivations, there is very little concrete resemblance between Imalab and ScilImage. One major reason for this is the use of C++ and object oriented programming, rather than using C, which completely changes the system architecture. The use of C++ as command language implies an important facility that an experimental environment "ideally" should provide [**?**]: "An essential facility during testing and debugging is the ability to monitor and examine both data and the interaction of system components".

Another specific characteristic of the Imalab approach is the concern to integrate external libraries; this is useful e.g. to compare different solutions to a given problem, to combine vision with other domains, to extend vision with AI programming.

### Outline of the paper

In section 2, we present the Imalab system as it is currently being used: a highly interactive programming environment featuring a large number of vision-related

data structures and algorithms. Section 3 presents the underlying method for constructing vision systems, available modules, and module generation tools.

## 2 The Imalab System

Imalab is an interactive programming shell for computer vision research. Its most prominent features are:

- A large choice of data structures and algorithms
- A subset of C++ statements as interaction language
- Extensibility through dynamic loading
- A multi language facility including Scheme, Clips, Prolog

By the use of standard or personalized scripts, the user initializes the system in a way that places him/her in a comfortable environment where he can efficiently work on a particular problem; the initial environment includes a set or a sequence of images, a window for image display, and a number of global variables allowing the detailed exploration of all data structures at any time during a session.

Any shell is characterized by an interaction language and an environment, which must have in common a set of data types: the Imalab shell uses C++ statements as interaction language, including a subset of C++ expressions which is "complete" in the sense that it gives access to all functionalities of the programs.

### 2.1 Data Structures and Algorithms

The standard Imalab environment contains about a hundred classes with a total of several thousand methods to be used for work on vision problems. A *help* command is available to get information on all class and function definitions; this is particularly useful to explore external libraries, and also helps to remember about your own programs.

Important basic classes are:

- Image classes. There is a hierarchy of image classes trying to realize a reasonable trade-off between efficiency, generality, genericity, and simplicity. Abstract classes provide a large number of virtual or generic methods for image management and basic processing (input/output, conversions, thresholding, histograms, etc.), base classes provide for one-, three-, or four-band images with byte, int, or float pixels. We do not use template classes for images in order to assure maximum portability, and a certain kind of simplicity.
- Image processing algorithms. Every user has a large set of algorithms to work on, so these modules are fluctuating quickly. In particular, we use fast algorithms for Gaussian filters[**?**] useful for working with scale space, a "color" module providing standard color encodings; a connectivity analysis module for image segmentation.

– Classes for image display and graphics. Using specific classes for windows and events, rather than giving direct access to the underlying system functions, simplifies shell programming: these system functions tend to have a large number of parameters, most of which have "natural" default values in the current environment.
– Objects of 2-D geometry: points, lines, rectangles, etc.
– Numerical routines, in particular matrix computations.

### 2.2 Friends

Excellent software exists for graphical plotting of numerical data, and for 3-D display. We can easily profit from these using a light interface through pipes. For example, Imalab communicates in this way with gnuplot and geomview, and there is a series of Imalab commands generating input for gnuplot and geomview, visualizing e.g. gradient, laplacian, or other filter values as curves or surfaces.

For the construction of graphical user interfaces, there are special provisions to simplify the use of FLTK[**?**] and QT[**?**].

### 2.3 Extensibility and Dynamic Loading

As can be seen from the enumeration of basic classes and algorithms, there is no point in trying to have a complete collection: the number of potentially useful algorithms is illimited. A vision system must be extensible in order to add new algorithms as they become available. One essential aspect of extensibility is dynamic loading. The Imalab command *require* loads the given module into the shell, making all classes, methods, and functions of the module source code available in the current shell environment. [1]

### 2.4 Basic Shell Interaction

The use of C++ as interaction and scripting language makes the shell easy to use and powerful, because the shell language is the same as the programming language for the source code [2]. In the shell, one can create objects, activate methods/functions, and inspect any data objects as can be done inside the source code.

Thus the shell gives the same feeling as writing a main procedure; in fact, it is much simpler, because a large number of initializations are carried out on starting the shell. These initializations, based on Unix-command parameters, define global variables that are handily used later on. In particular:

– The variable *CurrentImage* holds an image as a C++ object, shielding the user from details of image acquisition/conversion
– The variable *Screen* holds a window to be used for all kinds of image display.

[1] Module generation can be more sophisticated, see section 3.
[2] we will take up the multi language aspect in the next paragraph.

As all initializations are programmed in a script file, one can define a personnalized version of Imalab with complements to this script.

Using the same language for programming in the shell, for script files and source code is an essential feature to make the system practical and convenient: one can work out a sequence of image operations interactively, then put the same code into a script file (copying from the history file), and when it works correctly, "promote" this code into a compiled module, or as part of some library.

The shell language is a carefully chosen subset of C++, which is semantically much closer to Java: there is no pointer arithmetic, no indirection or reference operator, no casts [3]. Restrictions of this kind are necessary because C++ was designed for compilation, not for an interpretive shell.

### 2.5 Multi Language Feature

In the Imalab system, the term "multi-language" has two distinct meanings:

- A syntactic meaning. In the shell, the user can choose the syntax for his commands. The default syntax for the Imalab shell is C++, but one can just as well use Lisp (Scheme), or Prolog. One can switch from one syntax to another at any time; this changes the input reader, but not the shell interpreter.
- Source code language. The source code in any file may be written in any of Imalab's languages; being a collection of source files, a module may combine source code in different languages; there is a provision for easy cross-language calls.

Seamless integration in this respect is a strong feature: when creating an object, calling a method or a function, the shell user does not have to know which source language has been used to implement this particular code. Of course, this is possible only inasmuch as different programming languages share the same basic concepts, like object, method, function. [4] A basic step to achieve this integration was the extension of Scheme with a new data type *c-object* for "handles" to C++ data. Inversely, to access Scheme data from C++, nothing has to be done, because Scheme is implemented in C++.

As a consequence, Imalab can appear as a C++ shell (extended with Scheme), or as a Scheme shell (extended with C++), with the same functionalities. In practice, all Imalab users have knowledge of C++, and this is sufficient to use the shell, and to extend Imalab with new algorithms. Few users have knowledge of Scheme. In fact, Scheme becomes important to understand internal shell programming, and the tools of the workbench presented in section 3.

If the multi-language feature appears strange to you, you may consider it just an internal feature of the system. However, we firmly believe that it adds much power to a vision system workbench.

[3] Even though our goal always has been to implement a "comfortable C"++ shell", a fairly complete C++ interpreter progressively gets into reach. See the Ravi webpage for a discussion on this point.

[4] Much more shall be said on this in another paper.

### 2.6 Beyond C++: Memory management and Advanced Features

One good reason for combining C++ and Scheme is that Scheme, as a dialect of Lisp, includes important high level features one expects from a programming shell, which are not found in C++.

- Automatic storage management through garbage collection is precious for interactive use: in complex situations, it is impossible for a human to remember precisely which data structures still are in use. However, garbage collection in a Scheme system only concerns Scheme data, special care has to be taken if we also want to manage general C++ objects like images, windows, and the like. For this reason, the Scheme garbage collector has been extended to handle c-objects containing reference pointers, or to delete objects directly. Information about the way C++ objects are handled by the garbage collector has to be supplied during module generation.
- Dynamic typing allows for dynamic type checking. Indeed, the shell verifies all arguments for calls to C++ methods or functions. Dynamic typing also is an important ingredient for introspection and other reflective capacities.
- Error recovery and signal handling can be carried out by the virtual machine on which the Scheme implementation is based. This adds important functionality to the shell; for instance, the shell stays alive after a segmentation fault, so the user can go through all data to look for the problem. A control-C suspends a computation and recursively calls the shell in the interrupted environment: the user can inspect the situation, and then continue or abort the suspended computation.

## 3 The Imalab Workbench for System Building

The description in the preceding section clearly shows that the Imalab system is constructed in a highly modular fashion. The main point we want to make in this paper is that the Imalab system is just one example of the use of a very general workbench which allows to efficiently design and construct a large variety of vision systems with different behaviors.

This workbench is our response to the fact that it is impossible to construct a universal vision system: only a workbench can, eventually, be universal on the meta-level, enabling us to construct a state-of-the-art vision system for a given problem specification with little effort.

A workbench should propose a set of reusable modules, tools to create new modules from newly available software, and support for writing a system toplevel responsible for module integration, control, and other global aspects. These three points are taken up in the following subsections.

### 3.1 Building blocks: existing libraries

Section 2.2 has mentioned some of the vision related modules. The entire set of available modules is much larger, and also includes modules not directly related to vision. There are modules for

– learning algorithms
– language modules for Clips, Prolog, and a frame language
– Bayesian inference
– general data structures (tables, numerics, ...)

There may be redundancy in the modules. For instance, there are several modules implementing image classes which are quite equivalent. This eases the problem of portability: when importing new code, it is easier to import it with its data structures than to adapt the algorithms to our "standard" image structures. In fact, different implementations may coexist within Imalab - this is very useful for prototyping, and for testing combinations of different algorithms. In many cases, conversion between different image classes turns out to be trivial, because the pixel data are the same.

### 3.2 Tools for creating new modules

The major technical problem for the generation of new modules concerns the integration of "raw" C++ programs, which may be available as source code, or as dynamic libraries. Using C++ programs inside an interactive shell requires a fair amount of highly technical "interface code". The modern solution to this requirement is automatic interface generation. This role is played by the Ravi Interface Generator [5] which produces all necessary code by analysing C++ header files. Module generation does neither modify nor need the source code for libraries [6], nor does it need a special interface file [7]. However, it does need some informations that cannot be deduced from header files, e.g. about the use of reference counters, about the existence of output parameters, templates to be instantiated, etc.

### 3.3 Constructing a vision system

We now are able to sketch the basic steps of the Imalab method for constructing a vision system.

– Define the modules you want; you may (re-)use existing modules, or create new ones. Creation of new modules may take some time, even if all algorithmic problems are solved.
– According to the kind of system you want to create, write the basic script. Two typical situations are:
  • A stand-alone application system. In this case, the system structure is fixed. The modules can be linked statically, there is no interactive shell, or the shell will just be used by the system engineer.
  • An interactive system for use in teaching or research, as is the case with Imalab. The system structure is as open as possible, so we define a basic kernel, and each user loads dynamically whatever he needs.

[5] Ravi[**?**] is the name of the system shell Imalab is built on.
[6] as is the case with OpenC++[**?**]
[7] as is the case with Swig[**?**]

– Work the system's overall behavior. The workbench furnishes several language modules that make it possible to give a system a particular twist with little effort; for instance by using the production system module (Clips like) with an appropiate set of rules.

The workbench does not provide solutions to all problems, but gives important help to combine pieces for a solution into a single system.

One important point to note is that the whole workbench uses C++ as the basic implementation language. The source code for all tools and modules is available. We may also note that work on the tools never is finished!

## 4 Conclusion: Perspectives for Cognitive Vision

### 4.1 About the Multi Language Feature

Do we need AI languages for cognitive vision systems?

A system is more than just the sum of its parts: a system ties together the functionalities provided by its components, adding control and other high-level characteristics. We don't believe a single programming language can be well adapted for the implementation of all aspects of a vision system. We rather contend that, given that different programming languages each have their strengths and weaknesses, one should carefully choose the right programming language for each component of a system.

Argumenting about this point is subtle: from a theoretical standpoint, all programming languages have equal power, being all equivalent to a Turing machine. However, it also is true that programming languages differ in important aspects. In particular, each programming language proposes a type system which may be more or less adapted to a given problem. A good choice of programming language may greatly simplify the solution. For example:

– Lisp provides symbolic list structures, automatic memory management, functional programming, and dynamic typing. These properties are precious for the implementation of sophisticated object models, for the representation of knowledge within vision programs.
– C++ provides a rich set of tools for efficient implementation of sophisticated data structures. This is essential for image processing, all problems of "low-level" vision, and much more.
– Prolog provides unification and automatic backtracking. Our users don't seem to need this . . . for the moment.

The proof will be in the eating - our feeling is that dialects of logic programming languages, like Prolog and Clips, will show themselves useful to introduce symbolic processing and knowledge manipulation into vision systems. Significant work of this kind has been done long time ago [**?**][**?**], and should be taken up in current work.

### 4.2 The Imalab method

The essential assets of our approach are

- The set of libraries, usable as interactive modules
- The system kernel with the C++ interpreter
- The interface generator RIG

The set of libraries contains the basic building blocks that make up a vision system, as well as numerous user-specific libraries which generally are evolving at a very fast rate. The system tools also simplify the reuse of libraries, which is important given the tendency of research teams to produce new software at each generation of students. It is encouraging to see that the Imalab modules and libraries are now combining the work of three generations of thesis students.

The system kernel and RIG are not specific to computer vision. They are general tools, just as language processors are general tools. However, adapting such general tools for vision research will pave the way for progress in vision systems.

The Imalab system is in use for several years now, with a total of several dozen users in several research teams. We have started to distribute it under GPL (see the Imalab homepage[**?**]).

## References


1. `http://www.fltk.org/`
2. `http://www.aai.com/AAI/IUE/IUE.html`
3. `http://www.intel.com/research/mrl/research/opencv/`
4. `http://www.khoral.com/khoros/`
5. `http://www-prima.inrialpes.fr/lux/Imalab/`
6. `http://doc.trolltech.com/3.0/`
7. `http://www-prima.inrialpes.fr/Ravi/`
8. `http://www.swig.org/index.html`.
9. D.H.Ballard, C.M.Brown, J.A.Feldman. *An approach to knowledge-directed image analysis.* in [**?**].
10. Shigeru Chiba. *OpenC++ 2.5 Reference Manual.* University of Tsukuba.
11. V. Colin de Verdière and J. L. Crowley (1998) *Visual Recognition using Local Appearance.* European Conference on Computer Vision ECCV'98, Freiburg, June 1998.
12. J.L.Crowley and H.Christensen (editors). *Experimental Environments for Computer Vision and Image Processing.* World Scientific, Machine Perception Artificial Intelligence Series, Vol. 11, 1994.
13. A.R.Hanson, E.M.Riseman (eds.) *Computer Vision Systems.* Academic Press 1978.
14. Augustin Lux (2001). *Tools for automatic interface generation in scheme.* In *2nd workshop on Scheme and Functional Programming*, Florence, Italy, September 2001.
15. J.Rasure, S.Kubica (1994). *The Khoros Application Development Environment* In [**?**].
16. J.Rasure, M.Young (1995). Cantata: *Visual Programming Environment for the Khoros system.* Computer Graphics, A Publication of the ACM Siggraph, 29:22-24.
17. R.van Balen et al. (1994) ScilImage: *A Multi-Layered Environment for Use and Development of Image Processing Systems.* In [**?**].
18. I.T.Young, L.J. van Vliet (1995). Recursive Gaussian Filtering In *SCIA'95.*