

Iplt—image processing library and toolkit for the electron microscopy community

Ansgar Philippsen,^{*} Andreas D. Schenk, Henning Stahlberg, and Andreas Engel

Maurice Müller Institute for Structural Biology, Biozentrum, Klingelbergstr. 70, 4056 Basel, Switzerland

Received 4 June 2003, and in revised form 28 July 2003

Abstract

We present the foundation for establishing a modular, collaborative, integrated, open-source architecture for image processing of electron microscopy images, named *iplt*. It is designed around object oriented paradigms and implemented using the programming languages C++ and Python. In many aspects it deviates from classical image processing approaches. This paper intends to motivate developers within the community to participate in this on-going project. The *iplt* homepage can be found at <http://www.iplt.org>.

© 2003 Elsevier Inc. All rights reserved.

Keywords: Electron microscopy; Image processing; Collaborative software

1. Introduction

The extent of structural information acquired by electron microscopy techniques depends on the sample quality, the instrument and the data analysis. Progress is inherently coupled to advancements in each of these areas. This contribution concentrates on the third one.

The electron microscopy (EM) community has produced a large collection of sophisticated image processing tools and pioneering mathematical approaches (for a summary of available software see <http://3dem.ucsd.edu>). Several powerful software packages, a number of them originating from the 1960s, are utilized today, and some continue to undergo cycles of reconstruction and extension. From a software architecture point of view, most of these packages follow what we would like to call the “classical” approach to image processing: individual executables solving specific tasks are generated, usually based on some underlying image processing library. Some sort of scripting mechanism chains the executables together to form an abstraction layer.

There is nothing inherently wrong with this approach, on the contrary, it has proven itself within this

scientific community and others. Nevertheless, we would like to point out the following drawbacks:

The user is faced with several packages, each using different command syntax, formats, definitions, installation procedures, etc. As a consequence, time is spent in learning technicalities instead of producing results.

The developer that is confronted with the task of implementing a novel algorithm is limited to two choices: add code to an existing package, or write from scratch (we omit the third possibility of one person begging the other to implement a specific feature). Both possibilities have obvious disadvantages: The first one requires the availability and in-depth understanding of the source-code; the second one necessitates the implementation of many standard routines, so that only a fraction of the coding is devoted to the actual novelty. And both cases require the developer to have attained a certain programming skill level.

The aforementioned software packages are developed by a small number of specialized laboratories and extended by others to solve specific problems. While the creative potential is most obviously present, it is unfortunate that the EM community has not been able to concentrate its efforts on creating a common software platform akin to those available in X-ray crystallography (most prominently CCP4 (CCP4, 1994), CNS (Brunger et al., 1998), and lately PHENIX (Adams et al., 2002)).

^{*} Corresponding author. Fax: +41-61-267-2109.

E-mail address: ansgar.philippsen@unibas.ch (A. Philippsen).

We therefore propose to establish an open-source software tool for the 3D EM community that is build by a collaborative effort of the community itself. To this end, we have designed a novel software architecture we deem suited for this approach. This paper presents its design and initial implementation, and it is to be understood as an invitation to community members to participate in its future development.

The recent emergence and success of open-source software (<http://www.opensource.org>, <http://www.sf.net>), such as Linux (<http://www.kernel.org>), Apache (<http://www.apache.org>), GIMP (<http://www.gimp.org>), or similar projects of the medical imaging community (<http://www.itk.org>) and X-ray crystallography community (Adams et al., 2002), serve as examples on how *iplt* is meant to develop.

We are fully aware that several open-source image processing libraries are already available; we hope to convince the critical reader that our integrated approach justifies a design and implementation “from scratch”, as will be discussed at the end of this paper.

2. Design criteria

1. The system should appeal to novice and expert alike, providing several levels of user interaction and multiple possibilities for user contributions. The learning curve should be as linear as possible, requiring little investment from the novice to get started, yet enabling the expert to adapt the system to particular requirements.
2. The system should provide a core set of functionality (the *base*) that is easily extendable by *modules*. How modules interact with the base and how new modules can be implemented are most important aspects of the design.
3. A comprehensive set of documentation must be available at every level, comprising source-code documentation, usage manuals, as well as prepared examples and tutorials.
4. The system should work on all three major operating systems in use (Linux and other Unix variants, Microsoft Windows and Apple OSX).
5. To make *iplt* an open source project, an online platform (*web portal*) needs to be established and maintained. This platform contains the documentation and a repository of scripts, modules and other material; it also allows exchange of ideas and feedback on the software.

3. Roadmap

The project may be roughly divided into three phases:

- (i) Evaluation of existing software tools, design, and initial implementation of the new architecture (completed).

- (ii) Implementation of the most important existing algorithms and procedures to make the system usable in a working environment; involvement of community developers; setup of web-portal. This is the major focus for the coming months.
- (iii) Routine utilization; implementation of novel methods to advance the field. This is the ultimate goal and vision.

This paper marks the end of the first phase (i) and the beginning of second. The challenge that the project thus faces is reminiscent of the chicken-and-egg problem: the new software will only be used if algorithms are implemented, but they will only be implemented if the system is really worth to invest time in. Our group will focus on the implementation of established and novel electron crystallography algorithms.

4. Current state of implementation

4.1. Design components

In the implementation of the aforementioned design criteria 1. and 2., we have chosen to strictly follow object oriented paradigms throughout the construction of *iplt*. Two languages were selected, namely C++ (Stroustrup, 1997) and Python (<http://www.python.org>). They form an ideal combination: C++ is a compiled, efficient, strongly type-checked language, Python complements with a simple yet powerful syntax, interpreter style, an easy plugin mechanism that supports either Python code or a shared library, a wide community and a plethora of components and add-ons. Both languages enjoy widespread popularity and an ever-growing user base.

The overall software architecture layout is shown in Fig. 1, comprising several layers of encapsulation and abstraction.

At the heart lies the base class library, which encapsulates the essential data representations, such as an image or a function, and helper classes, such as size, point, vector. In addition, the abstract classes that are used for algorithms and gui components are also defined here.

The interface of the base class library is available to the three components that extend it, namely the algorithms (*alg*), the graphical user interface (*gui*), and the IO plugins, each explained in its own section further below.

The next abstraction layer is delivered by a layer of wrappers, which map a specified set of C++ classes and their interface to Python, giving rise to the three main Python modules *iplt*, *iplt.alg*, and *iplt.gui*.

These modules integrate seamlessly with standard Python and thus enable further abstraction layers, written in Python themselves.

We denote by *algorithm* an independent, C++ based, exported Python module. A *procedure* is a combination of algorithms at the level of Python.

4.2. Base

4.2.1. Image encapsulation

The central class, and indeed the most complex one, is the image. It encapsulates the actual discrete values as well as the image state (using the *state* pattern (Gamma et al., 1994)). This image state is characterized by the following:

Extent: encoded by a start- and end-index, specifies the size, offset from origin, and dimensionality. An index is always an integer triple, therefore the extent defines the dimensionality of an image: 1D (width, height = depth = 1), 2D (width, height, depth = 1) or 3D (width, height, depth).

Domain: is either spatial or frequency. Several discriminations are based on this property, such as the direction of the FFT (see below) or the calculation of pixel scale.

Type: one of real, complex, or half-complex, the underlying states encode all values in double precision. (It should be pointed out that the encapsulation of the image state allows implementation of other precisions, if required).

The image class interface comprises the set of methods available to interact with an image object and includes the following functionality:

GetValue, *SetValue*: based on an index (an integer triplet), a value can be either directly set or retrieved. For half-complex data, the complex conjugate is automatically returned for values falling in the “other half.”

GetIntpolValue: an interpolated value is returned based on a vector (a float triplet); the coordinate system is the same as for the indexed version above, e.g., a vector of (1.4, 3.1, -0.9) will return an interpolated value from the index block (1, 3, -1) (2, 4, 0).

FFT: a fast Fourier transform is applied to the data; the direction is given by the domain: spatial data is forward transformed, frequency data backward. It can optionally be performed using a memory efficient algorithm in-place transform.

Convolute: the image is convoluted with another image, a function or a kernel. It can be optionally performed in-place for memory efficiency reasons.

Transform: apply a linear transformation (such as rotation or scaling) to an image. The abstract class Transformer can be used to derive new transformations.

Subimage: a region indicated by a start- and end-index is returned as a new image object.

The public interface of the image class is kept minimalistic on purpose. In contrast to other image processing class libraries, it does not contain a long list of methods that implement various algorithms and procedures. Instead, it uses two specific object oriented patterns (Gamma et al., 1994) to interact with algorithms and gui elements: *visitor* and *observer*, as explained in the algorithms resp. gui section below.

Access to consecutive image values is provided by means of an iterator, which encapsulates the explicit start- and end-index.

4.2.2. Encoding meta-data

A recurring problem in any image processing application is the diversity of image formats, in particular the additional information describing an image, usually residing in the header of the file. We call this additional information meta-data, and—for the sake of discussion—have divided it into the following categories:

- (i) minimal set required to read and store the image into memory: size, origin and pixel-encoding.
- (ii) meta-data concerning a single, isolated image, such as acquisition date, instrument parameters or sample parameters.
- (iii) meta-data concerning the processing procedure, such as batch number, processing history, assigned characteristics.

Due to the diverse and ever-changing nature of this meta-data, it is not useful to define a fixed encoding (i.e., a fixed image format). Instead, we have devised the following encapsulation scheme in our class library: The minimal information is part of the image class itself, in fact this is the defining characteristics mentioned above. The remaining meta-data is represented by a separate info object, which is organized in a tree-like hierarchy (Fig. 2): branches designate specific groups, and the leafs contain the actual data items. Each image object has a data info object associated with it, allowing the contained meta-data to be queried and modified.

The current info implementation wraps the groups and items around an XML-DOM representation (<http://www.w3.org/XML>), allowing direct retrieval from and storage to an XML file.

This does not entirely remove the conflict of different image formats and different conventions. It has just shifted the discussion from “which byte represents which feature” to “how should we name this feature.” We propose to base this naming convention on the one introduced by the EBI for their EMDEP project (Tagari et al., 2002).

4.3. IO plugins

To actually deal with the multitude of various image formats, a plugin mechanism has been implemented, which enables the base class library to be extended with specific routines that can read and/or write a specific image format and convert it to/from the internally used representation. The read/write operation is not limited to local disks, of course, but can be performed over the network as well.

Each IO plugin must use the data-info class to convert to/from the specific image format it implements.

4.4. Algorithms

The algorithmic component is a collection of modules that extract information from an image and/or modify it. One can distinguish four different types of algorithms:

- extraction only, without input parameters (e.g., statistics calculation)
- extraction only, input parameters required (e.g., peaks search)
- modification, without input parameters (e.g., autocorrelation)
- modification, input parameters required (e.g., CTF correction)

To incorporate all of these possibilities using a single syntax, the following scheme has been devised, given an existing image object (depending on the nature of the algorithm, point 2 and/or 4 might be skipped):

1. creation of an algorithmic object (an instance of an algorithmic class)

2. setting of parameters (via algorithm object interface)
3. application to image (via image object interface)
4. extraction of results (via algorithm object interface)

A code example is given in Fig. 3 (left-hand side) to illustrate this scheme.

As mentioned above, the interaction between an algorithm and an image is implemented using the *visitor* pattern (Gamma et al., 1994): To this end, the algorithmic class—instantiated in 1.—is always derived from the visitor class (part of the base class library). This visitor class defines an interface that the algorithm must implement. The image uses this interface polymorphically to apply the algorithm to itself.

5. GUI

The graphical user interface is the third major component of *iplt*. It is implemented using wxWindows (<http://www.wxwindows.org>), ensuring cross-platform

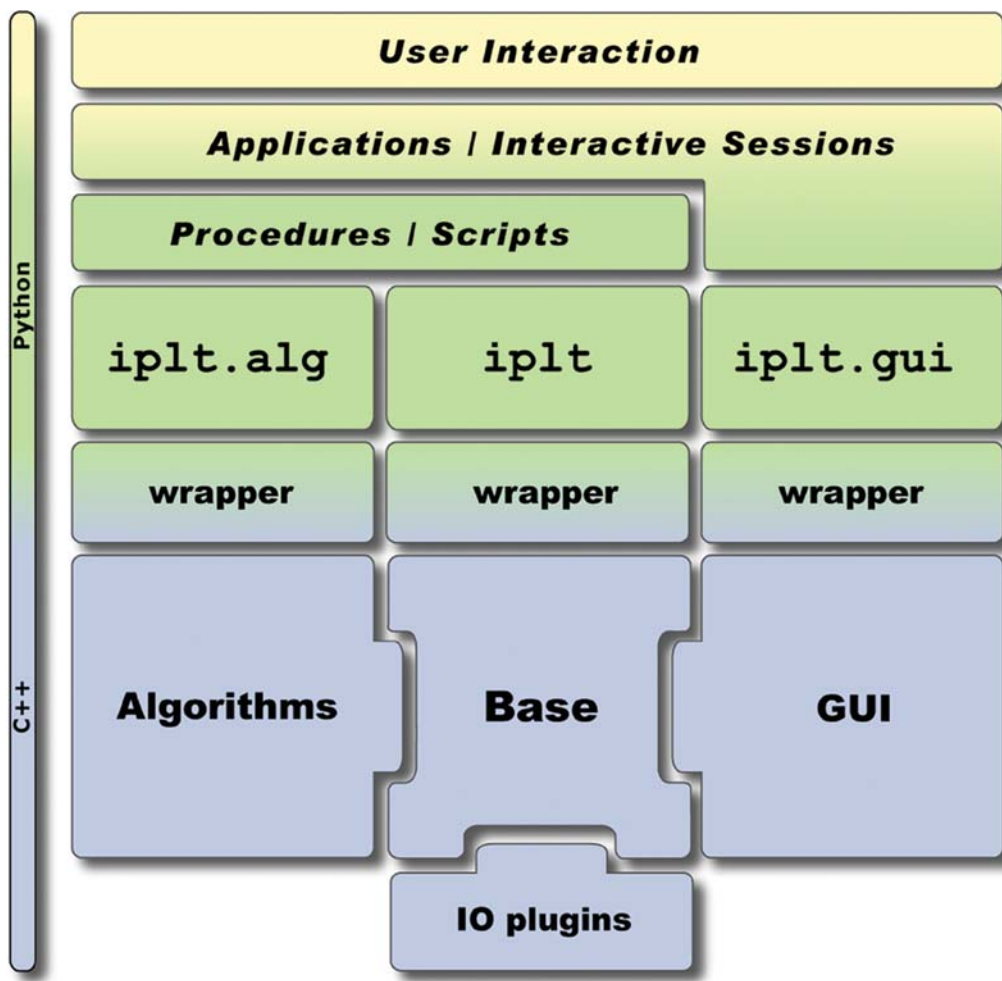


Fig. 1. Layout of *iplt*. At the heart lies the base class library, which is extended by algorithms, IO plugins and the GUI. Wrappers reflect the C++ classes into Python, in the form of three Python modules. Upon these modules, scripts, procedures, and applications written in Python are layered, with which the user interacts.

compatibility. It is not mandatory, meaning that the base and algorithmic components will function perfectly without it.

To facilitate the GUI, the *observer* pattern (Gamma et al., 1994) has been implemented in the image class. A GUI element that is designed to display image data (such as a 2D image viewer) must be derived from an observer class (part of the base class library). This observer class defines an interface that each derived class must implement. This allows a derived class to register with an image and to receive status update notifications. From the image point of view, it “knows” that one or more observers are watching it, but it has no information on the specific implementations of each observer; this is again made possible by the use of polymorphism.

It must be pointed out that the *architecture* to implement a GUI is finished, but not a fully functional GUI itself: at the moment, the 2D image viewer will simply display a given image and allow zooming, translation and individual pixel picking; it serves more as a “proof of concept” than anything else.

5.1. C++ to Python wrappers

The seamless transition between C++ and Python is ensured by the `boost.python` library, which is part of the boost project (<http://www.boost.org>). In contrast to other

tools that provide similar functionality, the interaction between C++ and Python is encoded explicitly, yet straightforwardly, in C++. Subsequent compilation of such wrapper code leads to the dynamically loadable Python module. This approach has been successfully used in a novel set of tools for X-ray crystallography (Grosse-Kunstleve et al., 2002).

An example of how this applies to *iplt* is given in Fig. 3: On the one hand, `boost.python` maps C++ classes and their interfaces to corresponding Python classes. On the other hand, it transparently converts Python objects back to C++ pointers or references, thus providing dynamic, run-time dependent interaction between C++ based objects.

This capability is especially exploited for the interplay between image class and algorithmic modules: An image object is unaware of any specific algorithmic implementation, it does however “know” how to handle a generic visitor class. A specific algorithmic implementation is loaded as a Python module, an algorithmic object is constructed within Python, and passed to the `Apply` method of an image object. The underlying C++ code evaluates the algorithm object as a polymorphic pointer, thereby calling the correct code within the specific algorithmic implementation. As dictated by object-oriented concepts, the algorithms implementation contains both the methods on how to apply itself to an image, as well as the parameters and results.

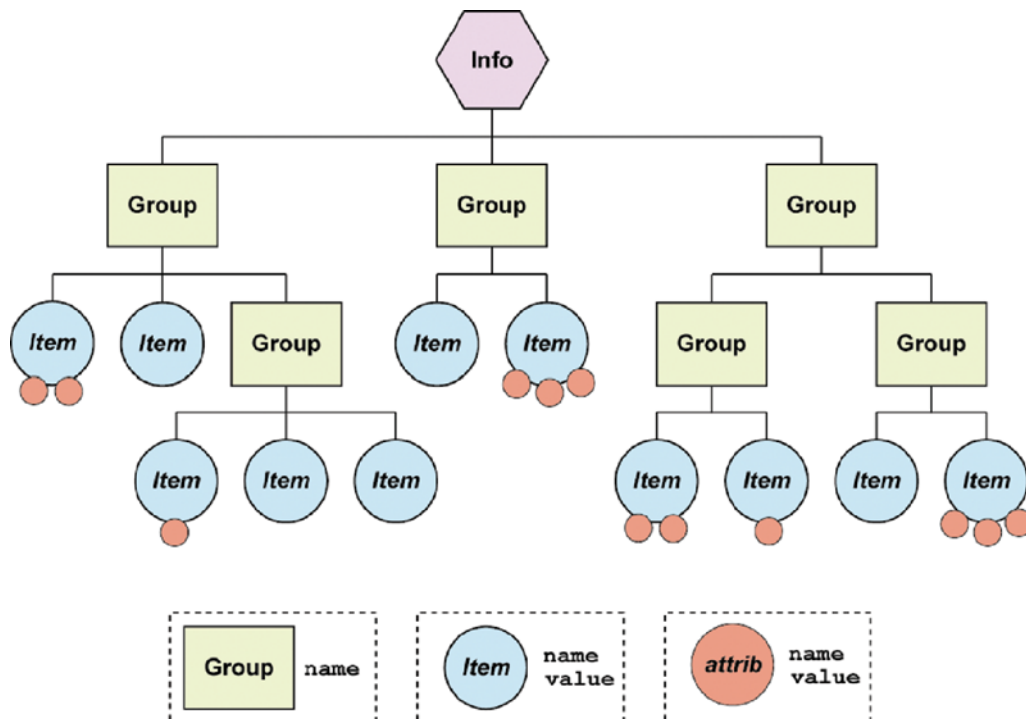


Fig. 2. The schematic layout of the meta info class. It contains a collection of groups and items, arranged in a tree-like hierarchy. A group is characterized by a name and may contain several subgroups and items. An item is characterized by a name and value, it may contain several attributes, which are also characterized by a name and value.

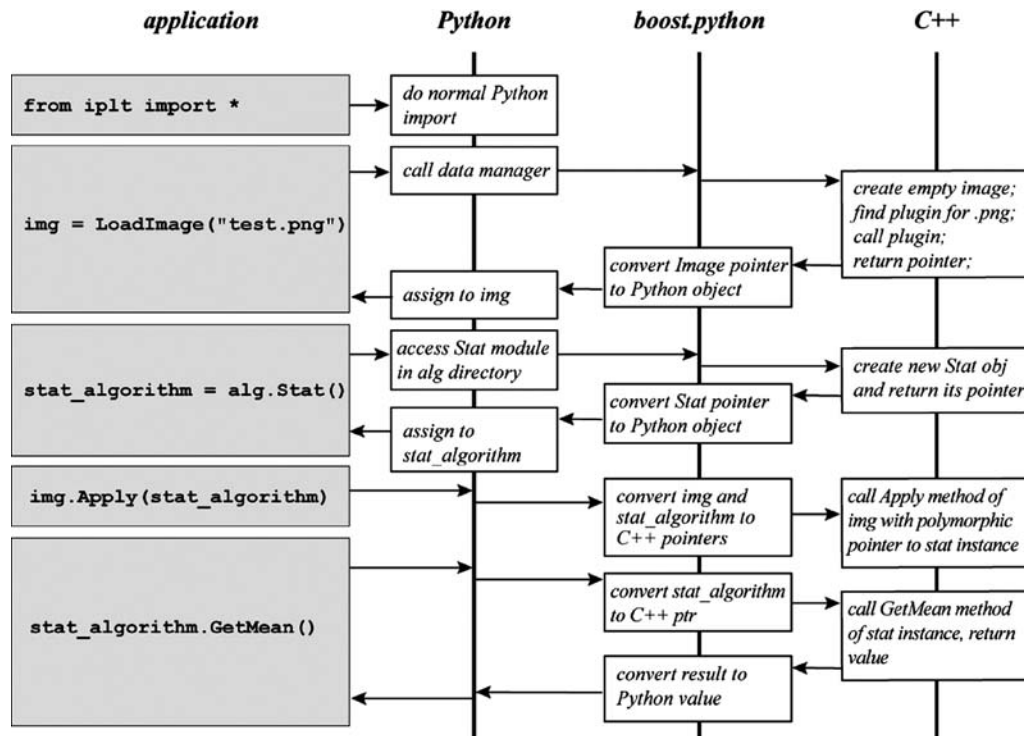


Fig. 3. Code example and the complex interplay between Python, C++ and its wrapper, mediated by boost.python. The wrapper is responsible for converting Python objects to and from C++ pointers. From top to bottom: (i) The *iplt* package is imported (internal Python command). (ii) An image is loaded from a file and stored in a Python object called “img.” (iii) A new instance of a specific algorithm is created (in this case destined to calculate some statistics), stored in “stat_algorithm.” (iv) The algorithm instance is applied to the image instance “img.” (v) The algorithm instance is used to retrieve results.

5.2. Building process

Two issues concerning the build process must be mentioned, both of which are addressed by the tool we have chosen to use, namely SCons (<http://www.scons.org>):

The aim for cross-platform compatibility is ideally extended to the building tool itself: SCons is setup in a way to automatically detect the platform and available compiler, the building rules specified within a SCons definition file are abstract.

A developer must maintain two different types of source within the *iplt* tree: the files from the distribution and his own code. The usage of SCons allows unproblematic coexistence of these two source types, integrating the custom code seamlessly within the build and install process.

6. Information exchange

6.1. Proposed documentation

There are several sets of documentation that will be offered along with the software:

The *design guide* is the fundamental document. It explains the philosophy behind the project, it defines

guidelines and rules for programming style, it lists and explains the software libraries and tools used, and it reflects the vision for future development. This is work in progress and a first version should be released around the same time as the publication of this paper.

The so called *in-source documentation* is assembled automatically from specially formatted comments and the object oriented hierarchy intrinsically defined in the source code. It is aimed primarily at developers who wish to contribute algorithms or new graphical user interface elements. In essence, it reflects the information embedded in the source-code, presented in a more accessible way. It is available from the web-portal and routinely updated from the source-code.

The *standards document* will contain detailed description of the EM specific definitions and conventions used throughout the project, as well as a description of the various image formats implemented with plugins. This is work in progress and subject to change based on future discussions among the project participants; a first draft, however, is being prepared, based on conventions circulated within the community (D. Belnap, personal communication; R. Marabini, personal communication).

The *user manual* will provide a comprehensive overview of the system and serves as an entry point for new users, offering a somewhat non-technical introduction

and several code examples. The writing of this manual is postponed until the project has matured.

6.2. The web portal

The project homepage can be found at <http://www.iplt.org>. At the time of writing (July 2003), it offers access to the source-code and some documentation, in particular the automatically generated in-source docs, detailed instructions for building and installation, and guides for adding new algorithms and IO-plugins. In addition, it contains information on the official mailing list, which will serve as the first medium to facilitate user and developer exchange.

7. Discussion

We have designed and implemented a novel software architecture for 3D EM image processing, with the goal to establish the first truly collaborative, open-source software project in this field. This architecture, as presented in Fig. 1, is a carefully crafted interplay of separate components and abstraction layers. It reflects our pursuit of object oriented paradigms, which are rooted in the choice of C++ and Python as the two programming languages that comprise the system.

The approach presented herein allows our main design goals to be met: (a) Provisions are made for several levels of user interaction, from the GUI to the use of ready-made Python scripts to the direct interaction with the main Python modules. (b) Community members may contribute in multiple ways and on several levels; the abstraction layers ensure a relative independence and flexibility of these contributions. (c) The incorporation of algorithm modules and I/O plugins is straightforward and allows flexible extension of the base functionality.

We chose to implement the underlying C++ image processing class library from scratch, well aware that several existing open-source libraries are available (such as <http://www.itk.org>). Our rationale for this path was and is the optimal realization of our design criteria, in particular the interplay with Python and the implementation of algorithm objects, which necessitated this seemingly drastic measure.

The Python language is becoming more and more popular in many different fields, including the structural biology community: in X-ray crystallography, the PHENIX project (Adams et al., 2002) incorporates Python in a similar way to *iplt* (and in fact served as an inspiration); in the 3DEM community, at least one software package (EMAN, Ludtke et al., 1999) features a fully functional Python interface. The most prominent features of Python, in our opinion, are the versatility as both scripting and programming language, incorporation of object-oriented paradigms, massive

amount of various modules, and easy extensibility with C/C++.

We hope to largely eliminate the problems and difficulties stemming from the variety of image file formats by the implementation of IO plugins and the encapsulation of meta-data (as shown in Fig. 2). One of the remaining issues is the comprehensive storage of the internal meta-data representation on the filesystem; with the current implementation, a separate XML file must be used to capture the full extent of the meta-data, since the commonly used file formats will only store a subset. A remedy might be the incorporation of the HDF5 file format (<http://hdf.ncsa.uiuc.edu/HDF5/>), which allows arbitrary data and meta-data to be stored within a single file.

The separation between the image and its algorithmic modules might seem cumbersome at first. In comparison to the more “classical” approach of implementing each possible algorithm as a method in the image interface, we would like to note the following advantages of our design: (a) Algorithms can be added to the system without recompiling the base component, they are independent entities. (b) The concept of *algorithmic objects* introduces a flexible and intuitive way to deal with the variety of potential algorithms: their individual parameters can be set or queried at any time, they can be re-used on several images, thereby even accumulating results, and they allow the various algorithm types to be incorporated using a single, comprehensive syntax. (c) The algorithm developer is forced to use the image interface, thus allowing the internals of the image class to be modified if necessary.

The graphical user interface forms an intricate, yet clearly separate component of the architecture. It is built upon wxWindows (<http://www.wxwindows.org>), a powerful cross-platform toolkit. Its independence from the rest of the code is established by implementing it both as a separate component as well as running it in a separate thread. As a consequence, it remains a non-mandatory component that is nevertheless tightly integrated should graphical user interaction be required.

The dependence of the *alg* and *gui* components on the base component is unidirectional: While subclasses of a visitor (algorithms) or observer (*gui*) exist, their specific implementation is irrelevant to the base. Therefore, changes in an algorithm or the GUI do not affect the base class library. On the other hand, both the algorithms (visitors) and *gui* (observer) heavily use the base class library and thus changes in the base class library potentially affect any algorithmic or *gui* implementation.

The collaborative aspect is manifold, following the open source philosophy “every user can contribute”. The possible list of contributions includes providing constructive feedback, requesting novel features, submitting own procedures, scripts, tips, and tricks through the web portal, incorporating more algorithms, tweak-

ing existing code, and extending the core functionality by working on the architectural foundation.

The specifications and standards adopted by *iplt* are not explicitly mentioned here, mainly because they are subject to change based upon opinions and decisions emerging from the discussion among contributors.

The current code does not include any specific parallelization features. This fact does not impede or even prohibit the later introduction of such features, because the architecture allows parallelization to be implemented at two very different levels: (a) In the context of C++, the internal functionality of classes in the base or algorithm module can be modified to utilize multiple processors (such as the FFT method of the image class). As long as the interfaces remain unchanged—and this can be expected for parallelization—these modifications do not require any adjustment outside the class. We would like to defer such parallelization to a time when the code has reached full functionality and optimization is performed. (b) In the context of Python, the iterative application of algorithmic sequences (procedures) over a potentially large number of individual images can be distributed in a multi-node or grid environment. At the moment, *iplt* has not yet reached the stage where procedures have been written and are routinely applied; as a consequence, the actual implementation of this sort of parallelization must be kept in mind until then.

The short-term goals are dictated by the roadmap, as outlined above: the next project phase marks the beginning of the actual collaboration; as a consequence, our energy is now invested in the enhancement of the web-portal, the completion of various pieces of documentation, and of course implementation of algorithms.

To summarize, we have taken a first step in setting up a collaborative image processing tool for the 3D electron microscopy community. Its success depends on both our continuing contribution as well as the acceptance within the community. This work may raise sufficient motivation for interested laboratories to invest some initial time and energy, with the potentially rewarding outcome: a novel software tool that has emerged by the creative effort of the community itself.

8. Software tools and methods

8.1. Introduction to object oriented design

We would like to briefly mention several terminology techniques from that field in order to facilitate the discussion. For comprehensive treatment of this subject, the reader is referred to the wide range of excellent textbooks on this matter, such as (Eckel, 1995; Meyer, 1997; Stroustrup, 1997).

In an object oriented model, both the data itself as well as the *methods* (functions) that act on that data are combined into a *class*; this is called *encapsulation*, implicating that the data items themselves are not directly accessed, but rather through their class methods. This set of methods is called the *interface* of the class. As a consequence, the intricate detail of data organization and encoding are shielded from the “outside” (the program), and the only exposed component is the interface. An instance of a class during runtime is usually called an *object*.

The principle of code-reusage is very important in object oriented design. To this end, classes may become part of other classes data by what is called *composition*, or they may be subclassed by a mechanism called *inheritance*. The latter is additionally empowered by a feature called *polymorphism*, which enables a subclass to assume the identity of its base (parent) class, yet override some or all of the base classes methods.

8.2. Software and libraries

All software tools and libraries that are used within *iplt* are cross-platform compatible, (they are supported at least on Linux, Microsoft Windows, and Apple OSX), their source-code is available at no cost, and they allow incorporation into open-source software.

All C++ code is Standard C++ (ISO/IEC 14882) compliant. The STL (Standard Template Library (Josuttis, 1999)) and the Boost library (<http://www.boost.org>) are utilized as much as possible.

The Python version that is currently employed is 2.2 (<http://www.python.org>).

To facilitate refactoring (Fowler, 2000), unit tests are implemented using CppUnit (<http://cppunit.sf.net>), and are run after each build.

The GUI component is based on wxWindows (<http://www.wxwindows.org>).

Fast Fourier transform routines are from the fftw library (Frigo and Johnson, 1998, <http://www.fftw.org>).

In-source documentation is parsed and formatted with doxygen (<http://www.doxygen.org>).

The XML-DOM implementation is done with Xerces from the Apache project (<http://www.apache.org>).

Vector and Matrix classes are from tvmet (<http://tvmet.sf.net>).

The building process is controlled by SCons, a python-based tool (<http://www.scons.org>).

Source code revisions are handled by CVS, the Concurrent Version System (<http://www.cvshome.org/>).

Acknowledgments

This work was supported by the M. E. Müller-Foundation of Switzerland, the Swiss National Foundation

(Grant No. NF 31-59415.99) and the Swiss National Center of Competence in Research (NCCR) ‘Structural Biology’.

References

- Adams, P.D., Grosse-Kunstleve, R.W., Hung, L.-W., Ioerger, T.R., McCoy, A.J., Moriarty, N.W., Read, R.J., Sacchettini, J.C., Sauter, N.K., Terwilliger, T.C., 2002. PHENIX: building new software for automated crystallographic structure determination. *Acta Cryst. D* 58, 1948–1954.
- Brunger, A.T., Adams, P.D., Clore, G.M., DeLano, W.L., Gros, P., Grosse-Kunstleve, R.W., Jiang, J.-S., Kuszewski, J., Nilges, N., Pannu, N.S., Read, R.J., Rice, L.M., Simonson, T., Warren, G.L., 1998. Crystallography and NMR system (CNS): a new software system for macromolecular structure determination. *Acta Cryst. D* 54, 905–921.
- Collaborative Computational Project, Number 4, 1994. The CCP4 Suite: Programs for Protein Crystallography. *Acta Cryst. D* 50, 760–763.
- Eckel, B., 1995. Thinking in C++. Prentice Hall, Englewood Cliffs, NJ. ISBN 0-13-917709-4.
- Fowler, M., 2000. Refactoring: Improving the Design of Existing Code. Addison Wesley, Reading, MA. ISBN 0-201-48567-2.
- Frigo, M., Johnson, S.G., 1998. FFTW: an adaptive software architecture for the FFT. In: Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing 3, pp. 1381–1384.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Pattern: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, MA. ISBN 0-201-63361-2.
- Grosse-Kunstleve, R.W., Sauter, N.K., Moriarty, N.W., Adams, P.D., 2002. The computational crystallography toolbox: crystallographic algorithms in a reusable software framework. *J. Appl. Cryst.* 35, 126–136.
- Josuttis, N.M., 1999. The C++ Standard Library: A Tutorial and Reference. Addison Wesley, Reading, MA. ISBN 0-201-37926-0.
- Ludtke, S.J., Baldwin, P.R., Chiu, W., 1999. EMAN: semiautomated software for high-resolution single-particle reconstructions. *J. Struct. Biol.* 128, 82–97.
- Meyer, B., 1997. Object Oriented Software Construction, second ed. Prentice Hall, Upper Saddle River, NJ. ISBN0-13-629155-4.
- Stroustrup, B., 1997. The C++ Programming Language, third ed. Addison Wesley, Reading, MA. ISBN 0-201-88954-4.
- Tagari, M., Newman, R., Chagoyen, M., Carazo, J.M., Henrick, K., 2002. New electron microscopy database and deposition system. *Trends Biochem. Sci.* 27 (11), 589.