Available online at www.sciencedirect.com
ScienceDirect
Journal of Structural Biology 157 (2007) 47–55
Journal of Structural Biology
www.elsevier.com/locate/yjsbi

SPARX, a new environment for Cryo-EM image processing

Michael Hohn [a], Grant Tang [b], Grant Goodyear [c], P.R. Baldwin [c], Zhong Huang [c], Pawel A. Penczek [c], Chao Yang [e], Robert M. Glaeser [d], Paul D. Adams [a], Steven J. Ludtke [b,*]

[a] *Lawrence Berkeley National Laboratory, Physical Bioscience Division, Berkeley, CA 94720, USA*
[b] *National Center for Macromolecular Imaging, Verna and Marrs McLean Department of Biochemistry and Molecular Biology, Baylor College of Medicine, Houston, TX 77030, USA*
[c] *The University of Texas, Houston Medical Center, Department of Biochemistry and Molecular Biology, Houston, TX 77030, USA*
[d] *The University of California, Department of Molecular and Cell Biology, Physical Biosciences Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*
[e] *Lawrence Berkeley National Laboratory, Computational Research Division, Berkeley, CA 94720, USA*

Received 9 May 2006; received in revised form 1 July 2006; accepted 7 July 2006
Available online 16 July 2006

Abstract

SPARX (single particle analysis for resolution extension) is a new image processing environment with a particular emphasis on transmission electron microscopy (TEM) structure determination. It includes a graphical user interface that provides a complete graphical programming environment with a novel data/process-flow infrastructure, an extensive library of Python scripts that perform specific TEM-related computational tasks, and a core library of fundamental C++ image processing functions. In addition, SPARX relies on the EMAN2 library and *cctbx*, the open-source computational crystallography library from PHENIX. The design of the system is such that future inclusion of other image processing libraries is a straightforward task. The SPARX infrastructure intelligently handles retention of intermediate values, even those inside programming structures such as loops and function calls. SPARX and all dependencies are free for academic use and available with complete source.
© 2006 Elsevier Inc. All rights reserved.

Keywords: CryoEM; TEM; Single particle analysis; 3-D reconstruction; Image processing

1. Introduction

Numerous excellent software packages are available for the TEM community, including SPIDER (Frank et al., 1996), IMAGIC (van Heel et al., 1996), BSOFT (Heymann, 2001), FREALIGN (Grigorieff, 1998), EM (Hegerl, 1996), IMIRS (Liang et al., 2002), SUPRIM (Schroeter and Bretaudiere, 1996), IMOD (Kremer et al., 1996), PHOELIX (Carragher et al., 1996), PFT (Baker and Cheng, 1996), the MRC reconstruction tools (Crowther et al., 1996) and Xmipp (Sorzano et al., 2004). Each of these packages has its own strengths and weaknesses, and although the general thrust of the methodologies is the same, each has its own set of particularly well developed methods used to achieve a final reconstruction. However, because of varying file formats, different conventions for Euler angles and the parameterization of the contrast transfer function (CTF) of the instrument, parameters of the image formation process, etc., moving between packages in order to take advantage of their respective strengths can be an exceedingly difficult and time consuming process.

Over the last two decades, single particle reconstruction has gone from a technique that initially was capable of achieving structures in the 20–30 Å range to a versatile mainstream tool frequently producing structures at subnanometer resolution, and in a few pioneering projects, approaching 4–5 Å resolution. One of the requirements to achieve high resolution is to limit the electron dose

* Corresponding author.
E-mail address: sludtke@bcm.edu (S.J. Ludtke).

1047-8477/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jsb.2006.07.003

delivered to the specimen in order to minimize the impact of radiation damage. Reduction of the dose causes a corresponding decrease in the signal-to-noise ratio in the image. Therefore, the techniques required to achieve high resolution structures must become ever more sophisticated in order to cope with very noisy data and at the same time to deliver highly accurate alignment parameters.

Numerous computational methods are used in a single particle reconstruction, and for each there are various choices of competing algorithms. For example, it is possible to compute a 3-D reconstruction from projection data using algebraic real-space methods often implemented as iterative algorithms, such as ART (Gordon et al., 1970) or SIRT (Lakshminarayanan and Lent, 1979), filtered back-projection methods, or direct Fourier inversion methods implemented in Fourier space (for detailed review see (Vainshtein and Penczek, 2006)). For 2-D alignment of noisy particles to low-noise projections, there are exhaustive search algorithms, iterative algorithms which separate the rotational alignment from the translational alignment, methods based on moments of the two images, and many others. Beyond this, there are choices between different methodologies for the overall reconstruction process, some of which use so-called 'inverse' methods, and avoid the reconstruction step entirely, as discussed below. In no case can a single algorithm be selected and make a claim to be 'the best' algorithm for all situations. Different algorithms react differently to variations in overall signal-to-noise ratio, shape of the object and CTF parameters. To discover the optimal methods for a particular reconstruction, it is often necessary to try many possible variants of the available algorithms. Incorporation of all or most possibilities into a single environment would dramatically ease this process of determining what works best in a particular experiment, and, in fact, may lead to a better understanding of why specific algorithms perform better in specific situations.

An additional difficulty arising in virtually all of the aforementioned software is the difficulty in organizing datasets which often consist of tens if not hundreds of thousands of particles drawn from tens to thousands of micrographs or CCD frames. Frequently, obtaining a final reconstruction involves gradually paring the data down to include only the highest quality micrographs, and/or only the highest quality particles from each micrograph. This process is a substantial data organization task, generally handled manually through careful organization of tiered directories containing hundreds of files each. This rapidly becomes untenable as reconstructions grow from a few thousand particles from tens of images to hundreds of thousands of particles from thousands of images. Development of an automatic data tracking system integrated with the reconstruction software has become an important task in the further development of single particle processing.

The goal in creating SPARX is to provide a uniform environment for end-users, in which to combine elements of different structure determination strategies, and develop their own approaches, without being forced to acquaint themselves with all available software suites. High-level strategies can be easily modified using a graphical programming approach, which does not require detailed knowledge of algorithms made available by the designers of the system. Issues such as file format conversion and Euler angle conventions are dealt with as automatically and transparently as possible. SPARX offers a core of robust image processing capabilities and an easy to use programming environment.

X-ray crystallography and cryo-EM single particle reconstruction are powerful techniques for macromolecular structure determination at intermediate to high resolutions. One of the goals of SPARX is to provide an integrated computational environment for both methods. This is of increasing importance as they are now often used concurrently. Crystallographic reconstructions of components of a macromolecular assembly can be readily combined with lower resolution structures of intact complexes solved by cryo-EM. In the future, as cryo-EM methods move to higher resolution it will be possible to take advantage of existing crystallographic tools for electron density interpretation. Conversely, crystallographic methods can benefit from the use of cryo-EM envelopes for phasing.

2. SPARX design

In recent years, a wide range of scientific disciplines have adopted the Python scripting language (http://www.python.org) in conjunction with low-level C++ code to address the need for flexibility without sacrificing performance. As discussed in Section 3, as an easy to learn object-oriented scripting language, Python permits very rapid development and debugging of new routines. Thus, SPARX makes use of C++ for compute-intensive code, while Python is used to implement complex higher level image processing tasks. The link between C++ and Python is generated using the Boost Python Library (Abrahams and Grosse-Kunstleve, 2003). This same approach has been used successfully in the PHENIX software suite for automated crystallographic structure determination (Adams et al., 2002, 2004).

The overall design of SPARX is diagrammed in Fig. 1. Users can interact with SPARX in three different ways: (i) through a graphical programming interface, which requires no formal programming background, (ii) through use of pre-written scripts from a command shell, (iii) through a text-based customized Python interpreter. The SPARX C++ core library integrates three components: a set of algorithms written specifically by SPARX developers, the core image processing library from the EMAN2 package (see the companion piece in this issue) and *cctbx* (Grosse-Kunstleve et al., 2002) from the PHENIX project, providing tools for manipulating crystallographic data and molecular models. Expanding SPARX by adding simple links to other image processing packages is a simple process. To demonstrate this, a partial wrapper for SPIDER (Frank et al., 1996) is provided, making use of the file format and

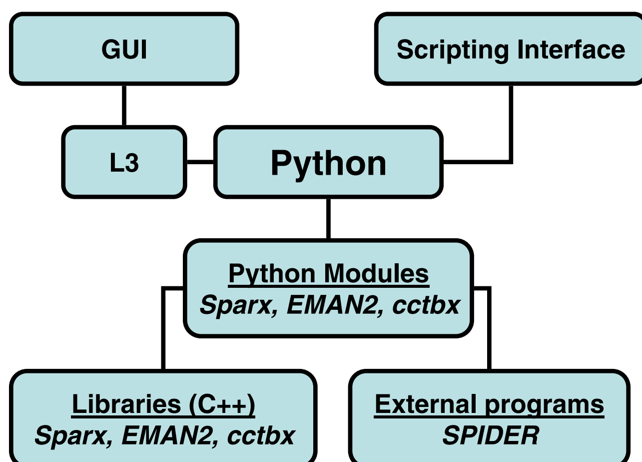

Fig. 1. Diagram of the overall design of SPARX. SPARX is built on top of several other toolkits including EMAN2 and *cctbx* from the PHENIX project through bindings to the Python programming language. External software suites not available as libraries (such as SPIDER) can be integrated through Python wrapper scripts.

mathematical convention conversions in SPARX. This wrapper gives access to many SPIDER algorithms from any SPARX interface. In the future, additional wrappers may be written by the SPARX developers or outside users for other image processing suites as the need arises.

2.1. EMAN2 library

The EMAN2 library provides the basic image processing functionality in SPARX, such as transparent read/write support for over 20 different image file formats, and several hundred image processing algorithms ranging from simple filtration to sophisticated 3-D reconstruction routines. The shared SPARX/EMAN2 library design makes use of an easy to use modular extensible class structure. All of these extensible classes support calling functions by name and passing of arbitrary parameters. Full introspection is available, meaning applications can get a list of available functions from the library when the GUI is executed. This permits the GUI to immediately become aware of any newly added algorithms, their parameters, and full documentation. As a simple example of the library interface, to apply a low-pass Gaussian filter to an image, one would simply call:

```
new_img = img.process ("filter.lowpass.
gauss", {"sigma" : 0.2}),
```

where img is the input image that will be filtered, new_img is the output image, filter.lowpass.gauss specifies the filter, which is a member of the process class, sigma is the name of the input parameter, i.e., the standard deviation in Fourier space of the filter Gaussian function, and 0.2 is the sigma value given in absolute frequency units, in which 0.5 is the Nyquist frequency.

The ‘process()’ and ‘process_inplace()’ functions provide access to a wide range of image processing algorithms, each accessed by name. Other algorithm categories also exist, and use a similar structure, such as ‘cmp()’ for comparing two image objects using a named similarity metric with optional parameters, or ‘align()’ for performing 2-D or 3-D registration of one image to a reference image. Documentation for each of the algorithms can be obtained interactively, for example, ‘dump_processors()’, through the on-line manual (http://www.macro-em.org/sparxwiki) or for programs making use of the library, through a set of introspection function calls. More details on this modular extensible class structure can be found in the companion manuscript on EMAN2 in this issue.

2.2. The computational crystallography toolbox (cctbx)

The *cctbx* is an open-source C++ library for crystallography and other scientific calculations (Grosse-Kunstleve et al., 2002). It provides a wealth of fundamental algorithms drawn from the PHENIX package (Adams et al., 2002, 2004), and is the source of the necessary crystallographic tools for SPARX. Consistent with the design goals of SPARX, it provides high-level interfaces to the underlying C++ algorithms via the Python scripting language. Indeed, the *cctbx* has been designed with an open and flexible architecture to promote extendibility and easy incorporation into other software environments. As with the other cores, the package is organized as a set of C++ classes with Python bindings. The *cctbx project* currently consists of the following modules, many of which are already available in SPARX:

libtbx: A build system common to all other modules (based on the open-source Scons software construction tool) and some associated general Python utilities for building and testing libraries and applications. The libtbx also includes *PHIL*, the Python-based Hierarchical Interchange Language (Grosse-Kunstleve et al., 2005) for user-friendly processing of input parameters.

boost_adaptbx: A very small *adaptor toolbox* with platform-independent instructions for building the Boost. Python library (Abrahams and Grosse-Kunstleve, 2003), a crucial tool for the efficient integration of Python and C++.

scitbx: Libraries for general scientific computing (i.e., not specific to crystallographic applications): a family of high-level C++ array types, matrix/vector manipulations, special functions, a fast Fourier transform library, and a C++ port of the popular L-BFGS quasi-Newton minimizer, all including Python bindings.

cctbx: Libraries for general crystallographic applications, useful for both small-molecule and macro-molecular crystallography. The libraries in the *cctbx* module cover everything from algorithms for the handling of unit cells to high-level building blocks for refinement algorithms and maximum-likelihood molecular replacement.

mmtbx: Libraries specific to macromolecular crystallography: absolute and relative scaling of protein and nucleic acid datasets, high-level PDB interpretation, automatic bulk-solvent correction, Cartesian molecular dynamics,

non-crystallographic symmetry restraints, and maximum-likelihood refinement targets.

iotbx: Input/output utilities to support the *cctbx* and mmtbx modules: automatic recognition and processing of all common reflection file formats, low-level PDB interpretation.

2.3. SPARX core

The core algorithms provided by the EMAN2 core and *cctbx* are augmented by a large set of algorithms reflecting the unique expertise of the contributors to the SPARX project. These include, for example, addition of highly accurate methods based on non-uniform Fourier transform, also known as gridding (Penczek et al., 2004) and a comprehensive set of 3-D reconstruction algorithms (for details see (Vainshtein and Penczek, 2006)). In addition, an implementation of a unified approach to the 3-D structure and projection orientation refinement using quasi-Newton algorithm (Yang et al., 2005) is being developed. In this method, the 3-D map and projection directions are updated simultaneously resulting in a rapid convergence rate, i.e., high resolution structures can be obtained faster than using standard 3-D projection alignment methods. These varied contributions will continue to expand as SPARX development continues.

3. Interactive SPARX (Python) interpreter

Over recent years, the Python scripting language has become increasingly popular in the scientific programming community, mainly because it dramatically shortens the turnaround time between concept and application. Complex ideas can be realized in a relatively easy to learn, but powerful scripting language opening many possibilities to non-specialists. Therefore, Python constitutes an ideal platform on which to base a modern scientific software package. As methodologies of EM structure determination are under continuous development, coding in Python makes it possible to rapidly implement, test and integrate new algorithms and strategies, making them immediately available to the EM community.

Python has been successfully used in numerous scientific visualization packages, for example, Vision (Sanner et al., 2002), Chimera (Pettersen et al., 2004) and Pymol (DeLano Scientific, http://www.pymol.org). The typical design pattern is the creation of a library in C++ or Fortran, containing compute-intensive operations. This library is then bound to the Python language in which all of the higher level logic of the final program is implemented. Python has native support for lists, dictionaries (hashes), sets and a variety of other data types, as well as string processing capabilities rivaling those of Perl. It also has an extensive set of standard libraries on all supported platforms, including a wide range of network protocols and mathematical operations. In many ways it represents a superior cross-platform development language to Java. Its object-oriented capabilities make it an ideal companion language to C++ based libraries.

```
bmb138> sparx
Welcome to SPARX.
In [1]: img = EMData()
In [2]: img.read_image("img001.spi")
In [3]: info(img)
Real image: nx = 64, ny = 64, nz = 1
  avg = 131.628, std dev = 27.7929,
  min = 43.192, max = 228.321
In [4]: i=EMImage(img)
In [5]: img*=-1
```

Fig. 2. A sample of an interactive text-based session in SPARX. This simple example, reads a file from disk, displays various image properties, renders the image in a new window, then inverts the contrast in the image.

The SPARX interpreter is an enhanced Python shell, and provides access to three distinct types of callable functions:

- Level 1: Direct access to the C++ core image processing routines from Python. Calling syntax is almost identical between Python and C++. This includes operations such as image i/o, basic image processing, Fourier transforms, filters, and so on.
- Level 2: Python scripts built using Level 1 functions, written as a shell command. For example, performing a 3-D reconstruction from a set of projection images could be implemented in a command-line program called ‘reconstruct.py’. This program could then be used directly from the system shell, or imported into the interactive interpreter and then called directly as a Python function.
- Level 3: Higher level Python scripts consisting of Level 2 or level 1 functions. These scripts can be written by hand or be constructed using the graphical programming tool described below. Once tested and built through the graphical programming interface, such algorithms may also be called from the interactive interpreter.

Now, consider the simple example of an interactive SPARX session in Fig. 2. This script will (1) create an image object, (2) read ‘img001.spi’ from disk, (3) display basic image information using a user-defined utility function, (4) display the image in an window on the screen with a control panel permitting adjustment of brightness, contrast, …, and (5) invert the contrast of the image data. Note step 5 will also produce an immediate screen update in the displayed image. Any operation applied in-place to img will be immediately reflected in the image display window.

In writing level 2 Python scripts, the concept is similar, but the style has to be slightly more expansive. Fig. 3 has an example of a command-line program that reads a stack of images, filters each image with a low-pass Gaussian filter, and writes the filtered images to a new stack file. The ‘sigma’ parameter specifies the full width of the low-pass Fourier filter expressed in units where Nyquist frequency is

```
#!/bin/env python
from EMAN2 import *
from sparx import *
from optparse import OptionParser
import sys

def main():
    progname = os.path.basename(sys.argv[0])
    usage = progname + "inputfile outputfile --sigma=<sigma>"
    parser = OptionParser(usage,version="1.0")
    parser.add_option("--sigma", type="float", help="sigma")
    (options, args) = parser.parse_args()
    if len(args) != 2:
        print "usage: " + usage
        print "Run '" + progname + " -h' for detailed options"
        sys.exit(1)
    filt(args[0],args[1],options.sigma)

def filt(infile,outfile,sigma):
    imgs=EMData.read_images(infile)
    for i in imgs:
        i.process("filter.lowpass.gauss",{"sigma":sigma})
        i.append_image(outfile)

if __name__ == "__main__": main()
```

Fig. 3. An example of a level 2 SPARX program. This program can be used directly from GUI, from the command prompt, or imported into Python as a callable function. This simple example will read a set of images from a file, low-pass filter them, then write the results to a new file, which may be in any supported file format.

0.5. The definition of any of the named parameters, like 'sigma' are part of the built-in documentation system available both in the manual and via introspection. Note that the actual script functionality is embodied in the 'filt()' function, and the remaining code simply deals with parsing command-line options.

If this program were in a file called 'filter.py', one could simply execute:

```
filter.py input.spi output.hdf -- sigma=0.5
```

from the command-line. Alternatively, one could achieve identical results from Python/SPARX with:

```
from filter import filt
filt("input.spi", "output.hdf", 0.5)
```

The script could also be used from within the SPARX GUI and take advantage of all of the features this interface provides.

4. The SPARX interface

4.1. The design

The main motivation for the development of the SPARX GUI is to provide an interface with the ease-of use of a graphical programming environment as well as a range of capabilities usually found only in language-based systems. First, the environment provides full retention of all intermediate values in a user-controllable way. Second, it supports parallelism and disconnected operation, meaning the user interface can be exited and restarted while a computational job is being executed. Third, it supports loops and conditionals with automatic dependency checking.

A few of these features have long been available in tools such as AVS (www.avs.com) and Iris Explorer (www.nag.co.uk/welcome_iec.asp), allowing non-programmers to write simple programs graphically without having to learn the syntax of an actual programming language. These tools represent specific tasks as boxes. A network is formed by interconnecting boxes representing data flow between different tasks. For example, one box might represent reading an image from a file, and a second box might represent displaying an image on the screen. The two boxes are connected to form a network that reads a file from disk and displays it on the screen. In general, when the network is executed, data interdependencies are checked, and all of the individual boxes perform their respective tasks. However, certain constructs needed in tasks like single particle reconstruction, such as data-dependent (dynamic) loops, are difficult to implement in such strict data-flow systems.

In order to address specific needs for EM data processing, a new language called L3 was designed for SPARX. L3 is a full programming language (manuscript in preparation) with numerous features, but briefly, L3 embeds Python, providing Python syntax and allowing direct access to all of Python's functions, including the SPARX library. The design of L3 adds certain capabilities such as retention of computed values and variables, the ability to resume an interrupted computation, the ability to execute only the portion of a program where dependencies have changed, and other refinements necessary to build the GUI. The lowest level tasks in the GUI, such as filtering an input image, are simple calls to the functions in the underlying SPARX

library. More complex tasks can then be constructed graphically by combining existing low-level tasks, naturally forming a hierarchy of tasks. Further, there is no distinction between user-defined tasks and tasks distributed with SPARX, so users can build their own custom task collections. All tasks have dual graphical and textual representation, regardless of how they were created. Thus, sharing assembled tasks with others can be as simple as e-mailing the textual representation. All of the image processing capabilities in the SPARX core are provided as visual tasks, so the end-user will rarely need to directly write L3 code.

The programming environment provides full retention of all intermediate values, in a user-controllable way. Every datum computed during execution is archived in a local database structure. In effect, a full history of the script execution is available for later investigation. For example, if a script representing an iterative reconstruction process is created and then executed for five iterations, the intermediate results produced during each iteration will be available. Since storage of all intermediate files in a large reconstruction would be prohibitive, the user can specify intermediates for which only the most recent value should be available. This mechanism also provides persistent sessions. That is, if the user builds a (visual) script for iterative reconstruction, executes four iterations, then exits, he or she can return later and reload the same script. Everything will be in the same state as when he or she last exited, including the full history of intermediate values for each of the four iterations.

To support parallelism and disconnected operation, the graphical interface is separated from execution. The user starts the GUI, builds a script visually, specifies which tasks can be executed in parallel, then executes the script. The actual execution is then handled by an independent process, which executes the script on the appropriate parallel resources, such as nodes of a computational cluster. The GUI then interactively displays the progress of the running script. Even if the GUI were terminated, the task would continue executing and the user could later restart the GUI and monitor the task's progress or access intermediate results.

To support long-term projects in which scripts evolve over time, tasks are executed only when data or parameters they depend on change. For example, if a long-running script containing ten iterations were interrupted during the sixth iteration, ordinarily, the entire script would have to be re-executed. Using L3, execution resumes after the last successful command, with all previously computed results preserved. If some parameters were changed before restarting the job, only intermediate values relying on the changed parameters would be recalculated.

4.2. The GUI

A screenshot of the current GUI is shown in Fig. 4. The main window consists of two panes. On the left is the library of available tasks. On the right is the canvas in

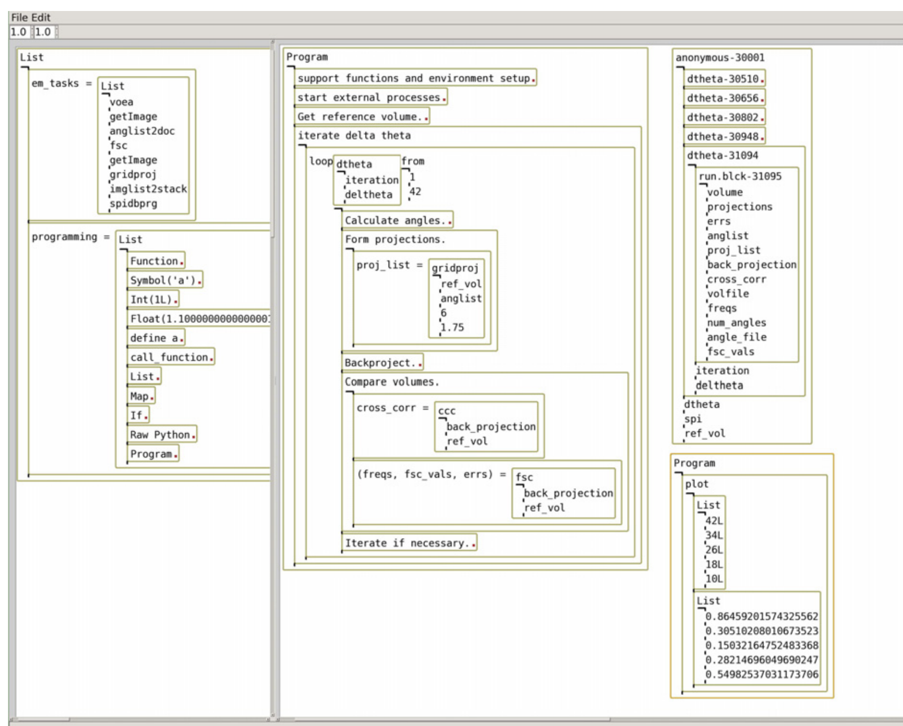

Fig. 4. A snapshot of the current SPARX GUI. The interface in the current version is still somewhat minimalistic, but it is undergoing continual improvements. The frame on the left contains available tasks and programming constructs that may be copied and used to form scripts. The frame on the right is used to construct scripts using the library tasks from the left. Shown in the right of this snapshot is the sample problem described in the text: the initial script to compute projections and backprojections (on the left), the hierarchy of computed data (on the upper right), and a second script constructed using data (delta theta and the cross-correlation coefficient) accumulated by the initial script.

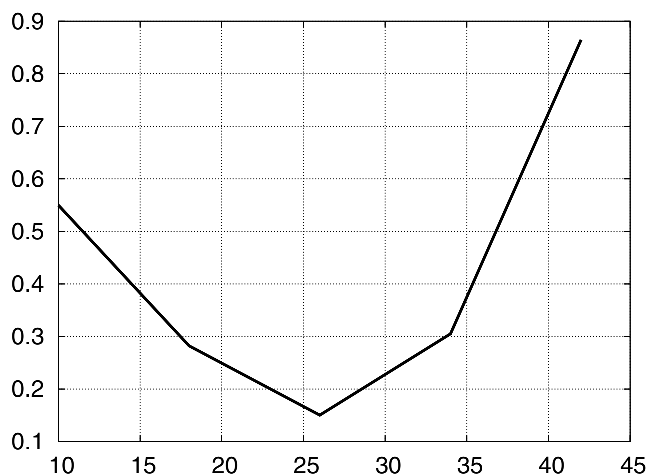

Fig. 5. The plot produced when the script in the lower right of Fig. 2 was executed. The current version lacks axis labels and other refinements, which will be added in future releases.

which the user constructs their program(s) from the tasks in the library. The canvas holds multiple scripts; each script is constructed vertically and is hierarchical. For example, a loop is represented by a box. The tasks that are executed multiple times as part of the loop are boxes *inside* the loop box. Each task has associated parameters/values. Analogous to the way a directory structure is browsed, parameters and contained elements can be displayed or hidden from view.

Note that the library portion of the display can also be expanded. The user can build a small collection of tasks to perform a specific function, name it and move it back into the library for later use in other scripts. Because tasks can be nested, whole programs can be contained in single task, and added to the library in the same way.

We demonstrate the workings of the GUI by following the simple example session shown in Fig. 4. Inside the loop, projections of a 3-D model are generated and then used to reconstruct a model. Begin by considering the left-most 'Program' box in the canvas on the right. This example begins with a 3-D model, generates a set of projections with a specific angular step, reconstructs the model from the projections, then compares the reconstructed model with the original. The loop in this example executes this overall process multiple times for different angular steps

```
list "support functions and environment setup":
    ...

list "start external processes":
    spi = init_spider()

list "Get reference volume.":
    ref_vol = getImage("volext.spi")

list "iterate delta theta":
    loop dtheta(iteration, deltheta) from (1, 42):
        list "Calculate angles.":
            anglist = voea(deltheta)
            num_angles = len(anglist)/3
            if (num_angles > 200):
                return 0
            else:
                0

        list "Form projections.":
            proj_list = gridproj(ref_vol,
                                 anglist,
                                 6,
                                 1.75)

        list "Backproject.":
            projections = imglist2stack(proj_list)
            angle_file = anglist2doc(anglist)
            volume = spi.bprg(projections, [1, len(anglist)/3], angle_file, "*")
            volfile = volume.save()
            back_projection = getImage(volfile)

        list "Compare volumes.":
            cross_corr = ccc(back_projection, ref_vol)
            (freqs, fsc_vals, errs) = fsc(back_projection, ref_vol)

        list "Iterate if necessary.":
            if (iteration <= 4) and (deltheta > 8):
                return dtheta(iteration + 1, deltheta - 8)
            else:
                return 0
```

Fig. 6. The text corresponding to the upper left visual program in Fig. 2. This script can be executed independently of the GUI while maintaining continuous data and execution tracking.

(deltheta). One could use such a simple script in order to examine the possible artifacts introduced by projection/reconstruction algorithms, and to establish a reasonable choice for angular sampling when performing a single particle reconstruction.

Once the script has been executed, the results are examined through the same interface—the script itself. For example, in order to inspect the reconstructed 3-D objects for each possible angular step, one simply goes to the 'back-project' box, selects the output image, and chooses 'display' from its menu. This will then display the list of images produced by all loop iterations. The full results are shown partially expanded in the 'anonymous' box in the upper right of Fig. 4.

Instead of just displaying a value, the value can itself be used as part of a new script. This introduces a convenient way of examining one's data. Continuing the example, by selecting the `cross_corr box` and choosing `'insert values'`, a list of values is put on the canvas. Then, selecting the `deltheta` box and again choosing 'insert values', a second list of values is put on the canvas. Now, a 'program' task and a 'plot' task (not visible in the figure) are added to the canvas, and a plot script is assembled (Fig. 4, lower right). This derived script is then executed to produce the plot in Fig. 5.

As mentioned above, a program constructed in the GUI is equivalent to an L3 script, which is what is actually executed. In Fig. 6, the textual script corresponding to the main visual script in Fig. 4 is shown. These scripts, whether typed by hand or produced by the GUI, can be executed as stand-alone programs outside the SPARX GUI, and still maintain history tracking capabilities, because even in the GUI, they are executed by the underlying L3 interpreter, not by the GUI itself.

4.3. Integration with other software

The SPARX GUI is basically a user-friendly wrapper for Python. To bring other (non-graphical) software into the SPARX environment and gain its benefits only requires writing a custom Python wrapper. The core library already supports multiple file formats and several data structuring conventions used by other packages, so conversion to/from an internal SPARX format by the wrapper is greatly simplified. This approach provides seamless integration of a variety of tools. As concrete example, a Python-SPIDER connection called pyspi is included with SPARX. This experimental module makes ~40 SPIDER commands available in Python and the SPARX GUI.

This capability will be gradually expanded as SPARX matures to allow direct mixing of algorithms between EMAN2, SPIDER, FREALIGN and possibly other TEM software packages. The process of integrating standalone programs into the GUI is usually straightforward. One of the great benefits of development within SPARX is the ability to take advantage of an existing package by incorporating it into a structure determination project.

5. Conclusions

SPARX will be an attractive environment for those users who wish flexibility in their image processing, but do not wish to learn a full programming language or to those with some programming skills who wish to integrate algorithms from multiple packages. It combines the ease of graphical programming as used in packages such as IRIS Explorer or AVS with process-flow features such as loops, historical data tracking and inspection, and parallelization capabilities. SPARX is not designed to supplant existing software, such as SPIDER and EMAN, but to unify access to such existing tools in a common environment, including novel algorithms developed in SPARX itself.

The core infrastructure of SPARX is largely complete and tested, and a usable prototype of the GUI has been completed. The first beta version of the complete package which will incorporate SPARX, the EMAN2 core, *cctbx* and all dependencies is expected in late-2006. Full documentation, details and contact information is available from the SPARX web site at http://blake.bcm.tmc.edu/SPARX. Both EMAN2 and the *cctbx* are also independently available at http://blake.bcm.tmc.edu/EMAN2 and http://cctbx.sourceforge.net, respectively.

Acknowledgments

This research has been supported by NIH Grant P01GM064692 and by the US Department of Energy under Contract No. DE-AC02-05CH11231. Support for the core EMAN2 libraries is provided by R01GM080139 and P41RR02250.

References

Abrahams, D., Grosse-Kunstleve, R.W., 2003. Building hybrid systems with Boost Python. CC Plus Plus Users Journal 21, 29–36.

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.

Adams, P.D., Gopal, K., Grosse-Kunstleve, R.W., Hung, L.W., Ioerger, T.R., McCoy, A.J., Moriarty, N.W., Pai, R.K., Read, R.J., Romo, T.D., 2004. Recent developments in the PHENIX software for automated crystallographic structure determination. J. Synchrotron. Radiat. 11, 53–55.

Baker, T.S., Cheng, R.H., 1996. A model-based approach for determining orientations of biological macromolecules imaged by cryoelectron microscopy. J. Struct. Biol. 116, 120–130.

Carragher, B., Whittaker, M., Milligan, R.A., 1996. Helical processing using PHOELIX. J. Struct. Biol. 116, 107–112.

Crowther, R.A., Henderson, R., Smith, J.M., 1996. MRC image processing programs. J. Struct. Biol. 116, 9–16.

Frank, J., Radermacher, M., Penczek, P., Zhu, J., Li, Y., Ladjadj, M., Leith, A., 1996. SPIDER and WEB: processing and visualization of images in 3D electron microscopy and related fields. J. Struct. Biol. 116, 190–199.

Gordon, R., Bender, R., Herman, G.T., 1970. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography. J. Theor. Biol. 29, 471–481.

Grigorieff, N., 1998. Three-dimensional structure of bovine NADH:ubiquinone oxidoreductase (complex I) at 22 Å in ice. J. Mol. Biol. 277, 1033–1046.

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.

Grosse-Kunstleve, R.W., Afonine, P.V., Sauter, N.K., Adams, P.D. 2005. cctbx news: Phil and friends. Newsletter of the Commission on Crystallographic Computing, International Union of Crystallography, 5.

Hegerl, R., 1996. The EM program package: a platform for image processing in biological electron microscopy. J. Struct. Biol. 116, 30–34.

Heymann, J.B., 2001. Bsoft: image and molecular processing in electron microscopy. J. Struct. Biol. 133, 156–169.

Kremer, J.R., Mastronarde, D.N., McIntosh, J.R., 1996. Computer visualization of three-dimensional image data using IMOD. J. Struct. Biol. 116, 71–76.

Lakshminarayanan, A.V., Lent, A., 1979. Methods of least squares and SIRT in reconstruction. J. Theor. Biol. 76, 267–295.

Liang, Y., Ke, E.Y., Zhou, Z.H., 2002. IMIRS: a high-resolution 3D reconstruction package integrated with a relational image database. J. Struct. Biol. 137, 292–304.

Penczek, P.A., Renka, R., Schomberg, H., 2004. Gridding-based direct Fourier inversion of the three-dimensional ray transform. J. Opt. Soc. Am. A 21, 499–509.

Pettersen, E.F., Goddard, T.D., Huang, C.C., Couch, G.S., Greenblatt, D.M., Meng, E.C., Ferrin, T.E., 2004. UCSF Chimera—a visualization system for exploratory research and analysis. J. Comput. Chem. 25, 1605–1612.

Sanner, M.F., Stoffler, D., Olson, A.J. 2002. ViPEr, a visual Programming Environment for Python, Paper presented at: Proceedings of the 10th International Python conference.

Schroeter, J.P., Bretaudiere, J.P., 1996. SUPRIM: easily modified image processing software. J. Struct. Biol. 116, 131–137.

Sorzano, C.O., Marabini, R., Velazquez-Muriel, J., Bilbao-Castro, J.R., Scheres, S.H., Carazo, J.M., Pascual-Montano, A., 2004. XMIPP: a new generation of an open-source image processing package for electron microscopy. J. Struct. Biol. 148, 194–204.

Vainshtein, B.K., Penczek, P.A., 2006. Three-dimensional reconstruction. In: Shmueli, U. (Ed.), International Tables for Crystallography. Kluwer, Dordrecht.

van Heel, M., Harauz, G., Orlova, E.V., Schmidt, R., Schatz, M., 1996. A new generation of the IMAGIC image processing system. J. Struct. Biol. 116, 17–24.

Yang, C., Ng, E.G., Penczek, P.A., 2005. Unified 3-D structure and projection orientation refinement using quasi-Newton algorithm. J. Struct. Biol. 149, 53–64.