CCP4 NEWSLETTER ON PROTEIN CRYSTALLOGRAPHY

An informal Newsletter associated with the BBSRC Collaborative Computational Project No. 4 on Protein Crystallography.

Number 49 **Summer 2013**

Contents

Editors: Karen McIntyre and David Waterman CCP4, SCD Department, STFC Rutherford Appleton Laboratory, Didcot, Oxon, OX11 0FA, UK

Contributions are invited for the next issue of the newsletter, and should be sent to Karen McIntyre by e-mail at karen.mcintyre@stfc.ac.uk. Microsoft Word format is preferred.

What can electron microscopists and crystallographers learn from each other?

Chris Wood
Scientific Computing Department, STFC Rutherford Appleton Laboratory, Didcot, OX11 0FA
chris.wood@stfc.ac.uk / 01235 567864

The STFC have recently been awarded an MRC Partnership Grant to establish a Collaborative Computational Project for Electron cryo-Microscopy (CCPEM), in an effort to both support the work done by software developers within the UK EM community, and to provide assistance to EM users who require help with computational aspects of their work. The first of two developers funded by the grant (me!) started at the end of August 2012, and shares an office with the core CCP4 team in the Research Complex at Harwell. The hope is that I can use the lessons learnt by CCP4 over the years, and that the two projects can share some code. CCPEM has three principle aims:

- Build a UK community for computational aspects of cryo-EM. Provide a focus for the cryo-EM community to interact with CCP4 and CCPN (Collaborative Computational Project for NMR), and the broader international community.
- Support the users of software for cryo-EM through dissemination of information on available software, and directed training.
- Support for software developers including porting, testing, and distribution of software.

Neither I, nor Martyn Winn (Head of Computational Biology, STFC), have an EM background, so we have spent the first couple of months of the project familiarising ourselves with a range of EM software and the problems faced by both users and developers. I have visited users (Helen Saibil, Elena Orlova & Carolyn Moores at Birkbeck College, Louise Hughes at Oxford Brookes University, Kay Grünewald at Oxford, and Ariel Blocker's Group at Bristol University) and developers (Alan Roseman at Manchester, Maya Topf at Birkbeck, Sjors Scheres at the MRC-LMB, and Juha Huiskonen at Oxford) to get a feel for what CCPEM should deliver. We have also held a very useful one day community meeting in Leeds (for which Neil Ranson and Arwen Pearson need to be thanked for local organisation). We had a good attendance of 35 researchers, who between them represented users, developers, and modellers within structural biology. We also had excellent presentations from Ardan Patwardhan (EBI) on how the EBI/PDBe/EMDB and CCPEM could collaborate on validation and deposition, and David Bhella (Glasgow), Ariel Blocker (Bristol) & Ed Morris (Institute of Cancer Research) on the computational problems that they, as EM users, have encountered in their research, how they overcame the problems and how they think that CCPEM could assist them in future. As a result of the information gathered at the

meeting, Martyn and I will decide on specific outcomes for CCPEM and feed them back to the community, to ensure that we will deliver useful services; however, it seems likely that in the short term we will focus on supporting currently available software, and consolidate the software being written by UK-based developers into a single software suite that can be distributed. In the longer term, we will look at the feasibility of providing computational services on behalf of the community; this could either be in the form of a *sandbox*, whereby users can remotely test software before deciding which software would be most useful for their needs (which they would then download to a local service, and so carry out all image and data processing themselves), or a more complex setup of a small cluster, with an ability for users to upload their data. This would have the advantage that users would not need to maintain their own computational facilities. A brief set of minutes from the community meeting, and the presentations given by Martyn and me, are on the CCPEM website (www.ccpem.ac.uk/courses.php).

We have also started to support two software developers. Alan Roseman's Find-EM package – which is a set of Fortran programs to help with automated particle picking from electron micrographs, is now available from the CCPEM website (www.ccpem.ac.uk/download.php). We are assisting Alan by extending parts of the program, and helping him to rewrite parts of it so that it will be more cross-platform compatible, with the aim being that it will be able to be run on Windows, Mac OSX, and Linux computers. We are also in the early stages of assisting Maya Topf's group, who are developing a set of python libraries to help determine the structure of macromolecular assemblies using flexible fitting techniques. Initially, we aim to host the library on the CCPEM website, before developing a front end for it and incorporating it into a more generic software suite. In addition, we are involved in discussions regarding the MRC format, which is in principle

identical to the CCP4 map format. In cryoEM, the MRC format is used for images (or stacks of images), for 3D volume data, or for stacks of volumes (4D data). Due to perceived limitations in the format, a number of extensions have arisen which cause problems when files are transferred between software packages.

We are very happy to be contacted by anybody who has an interest in EM, and who thinks they could benefit from the work that CCPEM aims to do. We also have a mailing list (www.jiscmail.ac.uk/ccpem) to which anyone can subscribe. Information about future community meetings, and the minutes from working group meetings, will be distributed on the mailing list and posted on the website (www.ccpem.ac.uk).

CCP4 Package Manager

Eugene Krissinel and Andrey Lebedev
CCP4, Research Complex at Harwell, Rutherford Appleton Laboratory, Didcot OX11 0FA, UK

Starting from release 6.3.0, CCP4 introduces a new tool, CCP4 Package Manager, in the Suite. This tool facilitates installation of precompiled CCP4 Suite on Linux and Mac OSX platforms, and provides updates to the Suite on all systems. For technical reasons, Package Manager is split in two application: `ccp4sm` (setup manager) and `ccp4um` (update manager), which, however, are compiled from the same source code.

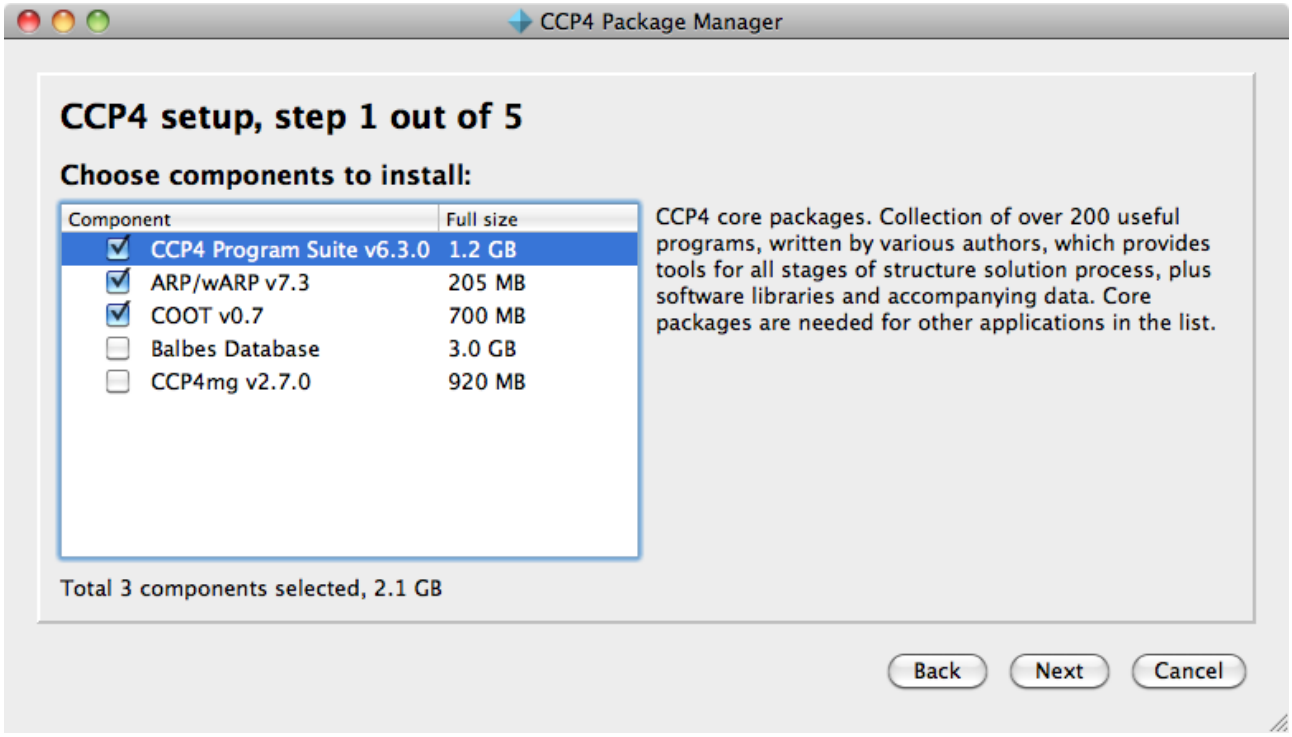
CCP4 Setup Manager

CCP4 Setup Manager, `ccp4sm`, represents a GUI front-end for the installation of CCP4 Software Suite. It does nothing else but exactly the same actions as a user would do in manual mode: identifying desired packages (CCP4 Core, Arp/wArp, Coot, Balbes database, CCP4 MG), downloading the corresponding archives (tarballs on Linuxes, or dmg images on Mac OSX platforms), unpacking, configuring and running installation scripts. Using `ccp4sm` gives extra convenience on comparison with installation in manual mode:

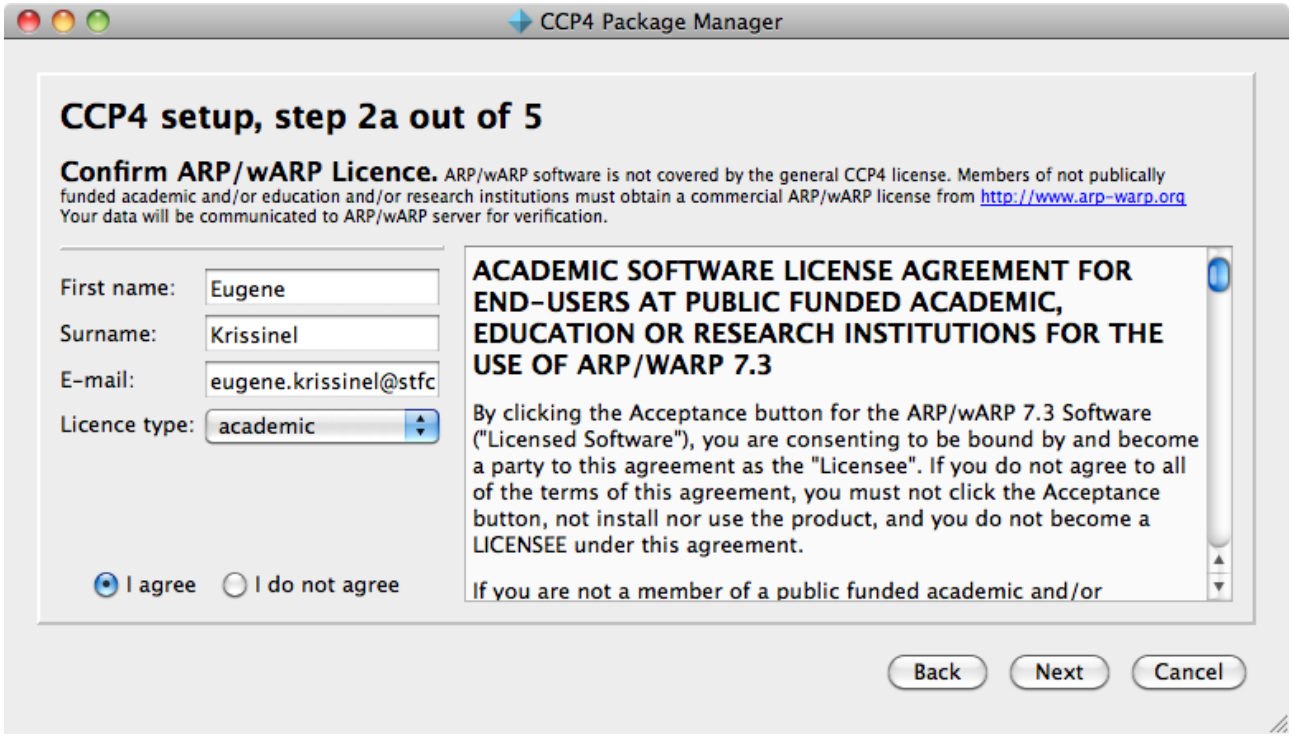
- decreased chances to install pre-compiled binaries incompatible with user's platform (`ccp4sm` will still allow, with a warning, installation of 32-bit packages on a 64-bit machine, if the corresponding libraries are available)
- download all required packages as a single action
- provide a resumable download, useful where network connection is poor
- will match specific packages to user's platform where pre-compiled binaries are not fully portable (such as Coot and, to the extent, CCP4 MG)
- does not require editing of setup files and working in Terminal
- cleans up temporary files and puts application icons (ccp4i, Coot, MG) on Linux Desktop

Before using, `ccp4sm` needs to be downloaded from CCP4 web site. It represents a light-weight graphical application (about 7MB single file on Linux platforms and a 15MB dmg package on Mac OSX), which will launch after double-clicking on it in OS's file manager (such as Nautilus or Finder). Full installation process is done in 5 steps, and is presented in 5 pages in `ccp4sm`, which are navigable back-and-forth using "Back" and "Next" buttons. At any step, installation may be cancelled by pushing the "Cancel" button.

Right after starting, `ccp4sm` establishes connection with CCP4 server. Then, as the 1st installation step, the user is presented with the choice of packages for installation:



After package selection, on the 2nd step, the user is prompted to confirm agreement with all applicable licences. In case ARP/wARP is included in the list, `ccp4sm` also queries ARP/wARP server in EMBL-Hamburg for authorisation. Certain user data: name and a valid e-mail address, is collected for this process:



Commercial users also need to specify their licence number, which should be acquired from EMBL-EM prior installation. Authorisation is usually instantaneous if network connection is stable.

On 3rd step, the user is asked about particular location (destination directory), where selected packages should be installed, as well as directory for temporary files. On Mac OSX platforms,

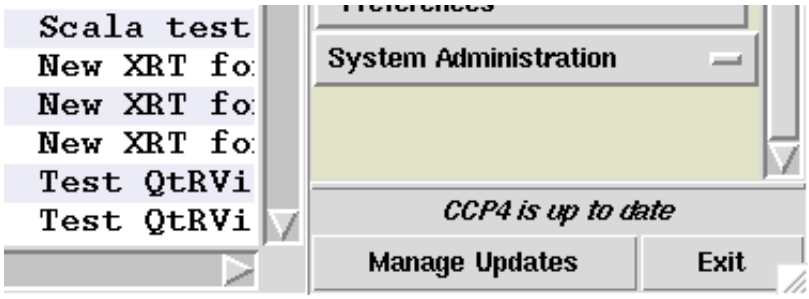
installation is limited to `/Application` folder, and `/tmp` directory is used for storing all intermediate data files. On Linux platforms, *ccp4sm* checks whether the user has sufficient privileges for writing into the destination directory, and requires a restart in administrative mode if user privileges are found insufficient.

Step 4 includes download of all selected packages. This may be a relatively long process, varying between 10-20 minutes up to a few hours, depending on the network speed, during which no further actions are required form the user. At any moment, the download may be stopped and resumed later, even if *ccp4sm* is restarted after a lengthy time period. If network connection is lost, *ccp4sm* will resume download from last recorded position, when it is started next time.

On the last step, *ccp4sm* runs installation scripts, which unpack downloaded files, put them into required location, configure and execute post-installation procedures where required. Upon completion, *ccp4sm* puts icons on user's desktop (Linux only), writes installation log `install.log` into destination directory, and removes all temporary files. After *ccp4sm* finishes, CCP4 packages are ready to use.

CCP4 Update Manager

CCP4 Update Manager, *ccp4um*, was introduced as an add-on feature shortly after CCP4 Release 6.3.0 in July 2012. In order to install *ccp4um*, a user would need to download a pre-compiled binary (Windows and Linux) or a dmg image with *.app folder (Mac OSX), copy it into `$CCP4` location and run by double-clicking in system's file manager. At first run, the updater integrated itself into the suite by creating `$CCP4/libexec` and `$CCP4/restore` directories and setting up the "Manage Updates" button and update status line in *ccp4i*:



In December 2012, CCP4 Release 6.3.0 was re-bundled to include *ccp4um* as a standard feature, eliminating the need to install it separately.

CCP4 Update Manager represents a graphical client application, which modifies CCP4 setups by downloading patches from CCP4 server and applying them locally. A "patch" may be either a file or an action to be applied to existing files. In update mechanism for CCP4 release 6.3.0, the following operations are supported:

- replacing a file
- placing a new file
- deleting a file
- changing file permissions

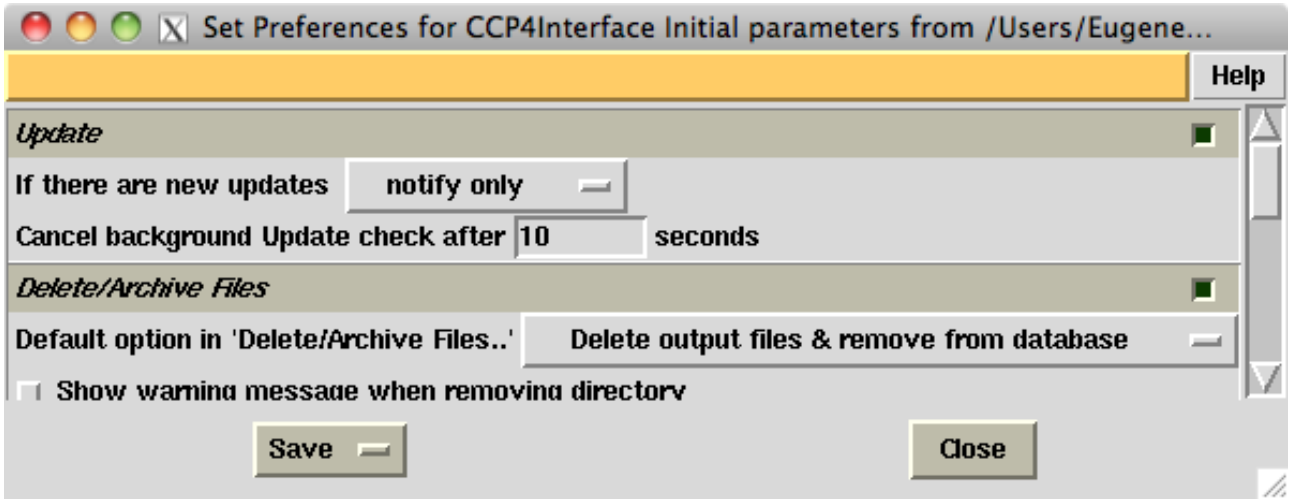
Starting from Release 6.4.0, they will be complemented with

- replacing/making/deleting a soft link
- running a process

the need of which has been identified from the experience with 6.3.0 update series.

Before applying updates, the updater makes an incremental backup of CCP4 setup, which is stored in `$CCP4/restore` directory. This allows a user to roll back, if any particular update was found harmful or unsuccessful. As an essential feature of CCP4 updates, they may be applied only consecutively, and rolled back strictly in reverse order. This restriction is imposed by the updater automatically. Even if most updates do not interfere with each other, tracking all possible dependencies in a relatively large Suite is difficult, therefore, only a single line of Suite evolution on client machines is allowed.

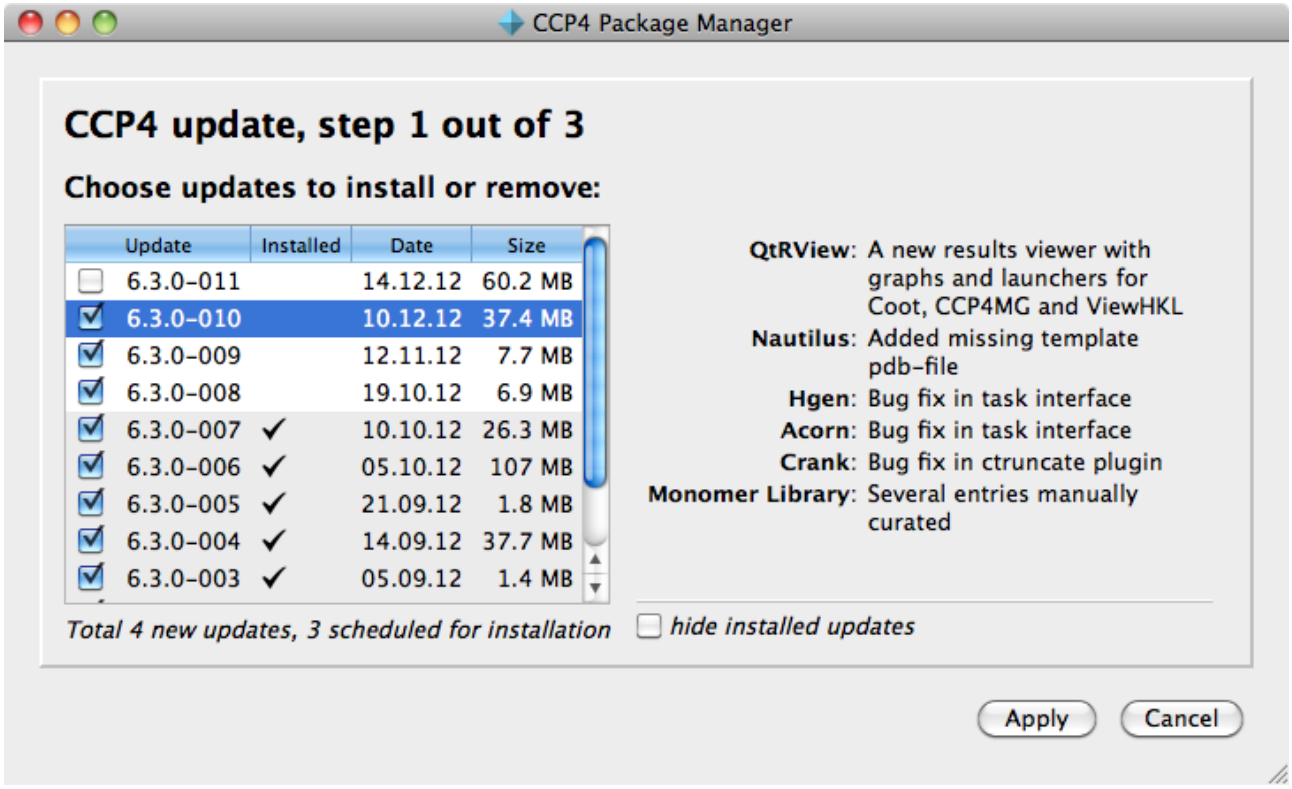
The updater may be started either from *ccp4i* (by pressing the “Manage Updates” button), or from command prompt by running *$CCP4/bin/ccp4um*. When *ccp4i* starts, it runs *ccp4um* in silent mode to check whether new updates are available. This process is controlled by *ccp4i's* Preferences, where one of three update modes may be selected:



The three modes are:

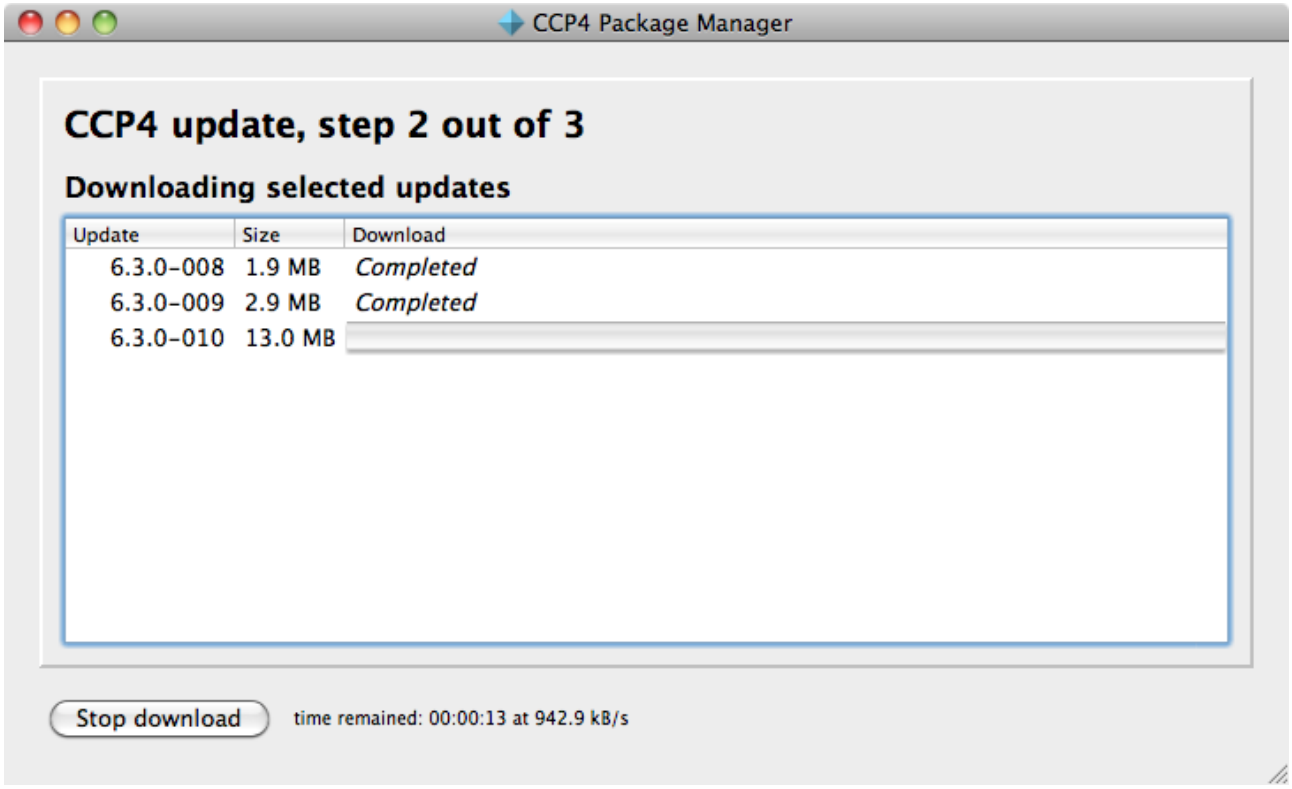
- do nothing: no checks for new updates are done
- notify only. This is the default mode, in which *ccp4um* runs silently each time *ccp4i* is started. If *ccp4um* identifies new updates on CCP4 server, a message “There are N new updates” is displayed in update status line in *ccp4i*, just above the “Manage Updates” button. At this point, the user may start *ccp4um* by pushing the “Manage Updates” button and install the updates
- start updater. In this mode, *ccp4i* acts same way as in “notify only”, however, in difference, *ccp4um* is started automatically in full interactive mode if silent check finds that new updates are available.

After starting in full graphical mode, *ccp4um* presents the user with the list of installed and available updates:

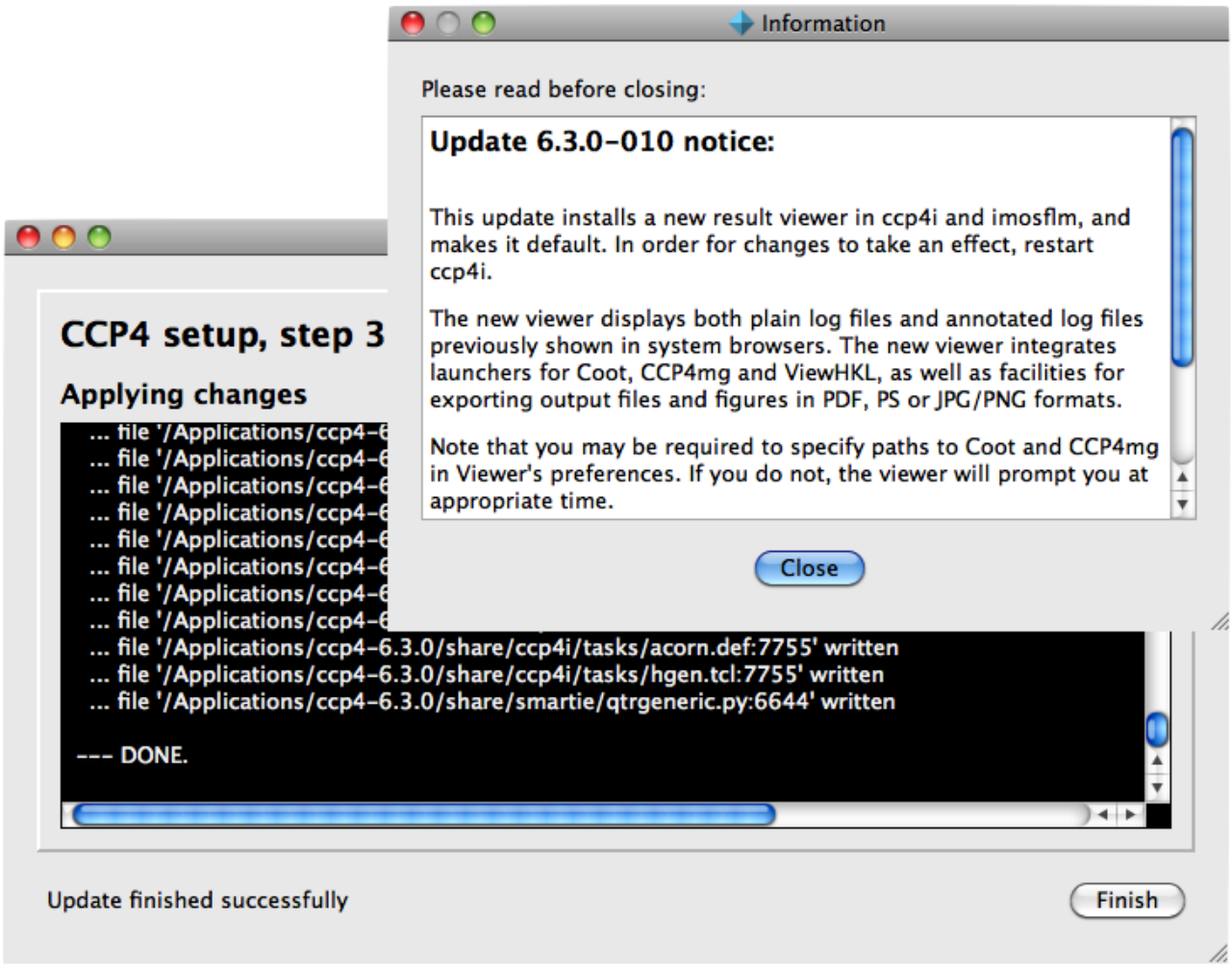


Selection of a particular update in the list will show a brief description of the update's content. In order to install update(s), a user needs to check the corresponding box(es) in the left-most column of the list, and press the "Apply" button. For uninstalling update(s), the corresponding box(es) need to be unchecked.

After pressing the "Apply" button, the rest of the update process goes automatically. First, the updates are downloaded from CCP4 server:



After downloading, *ccp4um* makes an incremental backup of changes to-be-made, and, finally, applies all patches. Optionally, *ccp4um* may display a pop-up window to advise the user of any unconventional effects the update may cause, or to re-start *ccp4i* if required:



The server-side of CCP4 update mechanism represents a set of tools for identifying updated components, packing them in archive files and placing in a designated location on the server. For this purpose, a set of template CCP4 setups (one per supported OS), representing the current state of the Suite on client machines, is maintained. Once updated component(s) are placed into the template setup, a special procedure calculates the induced difference, advances the update counter and generates the update description files (UDF). These files are retained for bookkeeping purposes and may be used to re-generate updates at any time. After generation, UDFs need some manual editing in order to add the update's annotation and optional pop-up messages to be displayed by *ccp4um*. After that, a separate process uses thus prepared UDFs to produce update packages and places them on the server. Update packages represent compressed archives containing the updated components, their descriptions and other service information for *ccp4um*.

Summary of all updates is available at http://www.ccp4.ac.uk/updates/, together with cumulative update patches. These patches may be used by those users who cannot run *ccp4um* because of, e.g., restricted network access.

During first 8 months of *ccp4um* deployment, CCP4 core team has issued 21 updates in 6.3.0 series. These updates contained mostly bug fixes, but also new versions of important components, such as Phaser 2.5.2, released in Update 6.3.0-007 in October 2012, as well as all-new applications, such as the new result viewer, QtRView (Update 6.3.0-010 in December 2012). It is

expected that, in future, most changes in the Suite will be delivered in update mode, rather than by means of full-scale releases, which, therefore, will become less frequent. The logics behind having both updates and releases is the same as that of virtually any other complex software, which normally has update and upgrade cycles. While updates are extremely efficient for delivering relatively small and local changes in the Suite, full releases are necessary when essential structural changes or new underlying technologies need to be implemented.

CCP4 web services

Ville Uski
CCP4, Research Complex at Harwell, Rutherford Appleton Laboratory, Didcot OX11 0FA, UK

Software as a service has become a popular software-delivery model with the advent of cloud computing. In this model, the software and the related data are centrally hosted on the cloud (network of remote computers), instead of installed on each user's personal computer, and they are typically accessed by using a thin client via a web browser, the so-called web application. Some of the benefits of the web applications as opposed to desktop applications include the lower cost of software maintenance and easier scalability of resources. Moreover, the ubiquity of ultra-portable devices has put forward the idea of having super-computing facilities available via the internet on the go.

Web services have recently stirred interest among macromolecular crystallographers, due to the increasing rate and volume of data collected in the experiments, and the complexity of computational setups, posing difficulties for small to medium size laboratories with limited access to full-scale IT support. There are already various web services hosted by facilities such as Diamond, which provides automatic data analysis and long-term data storage solutions with web access, and EMBL-Hamburg with its ARP/wARP model-building web service. CCP4 has recently incorporated in its strategy a focus on cloud computing. The plans include collaborations with Diamond and the new CCP4 GUI, which would include access to CCP4 web services through RESTful APIs.

CCP4 has provided some utility programs as a web service for almost 10 years now. The idea was originally to expand the service as an alternative way to run such utilities without needing to install the whole CCP4 software suite. While this service has not been actively maintained for some time now, there have still been some users. More recently, the Balbes webserver has migrated from the York Structural Biology Laboratory to the Research Complex at Harwell, and was taken over by the CCP4 core team. This webserver provides the Balbes molecular replacement pipeline (Long et al, 2008) as a web service (see figure 1). Using the web service has perhaps been the most common way to run the pipeline, and there have been many users.

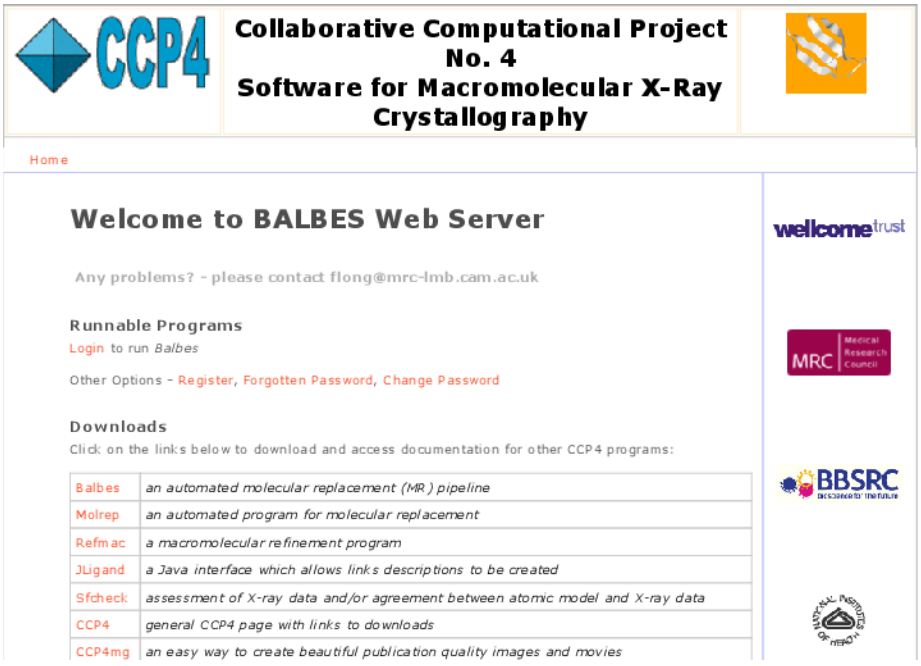


Figure 1: Web front-end of the Balbes molecular replacement pipeline.

Under the hood, as it were, the Balbes webserver is a Java servlet application, implementing the model-view-controller software architecture pattern. It runs in the Tomcat servlet container. Its development started over 10 years ago by Paul Young, was further developed by Fei Long, and recently modified by the CCP4 core team. The new webserver under development is also written in Java and implements the same architecture. However, it makes use of some of the modern Java frameworks, such as Tapestry, Hibernate and Shiro, and will implement APIs following the RESTful design model. Using the frameworks improves the scalability, security and the ease of maintenance of the web services, and makes it relatively easy to improve the user experience with the help of Ajax features.

Molecular replacement (MR) is an excellent candidate for a web service, as it requires large databases and can involve large-scale computations. MR pipelines implement computationally intensive tasks that can greatly benefit from parallelisation. The Balbes webservice is a front-end for Balbes, which runs in our Linux cluster. This pipeline has not yet been parallelised, but we are currently in the process of setting up a web service for MrBUMP (Keegan and Winn, 2008), which has been parallelised. The webservice will have a new front-end, including both Balbes and MrBUMP. Yet another computationally intensive MR pipeline, AMPLE (Bibby et al, 2012), will be included next.

There are several other CCP4 applications that are tentatively planned to become web services. These include various automatic pipelines for data processing, experimental phasing, model building and validation. Other ideas include web interfaces to CCP4's structure solution components and useful utilities.

References

- Long F., Vagin A., Young P. and Murshudov G.N. "BALBES: a Molecular Replacement Pipeline" (2008). Acta Cryst. section D64, 125-132.
- Keegan, R.M. and Winn, M.D. "MrBUMP: an automated pipeline for molecular replacement" (2008) Acta Cryst. D64, 119-124
- Bibby J., Keegan R. M., Mayans O., Winn M. D. and Rigden D. J. "AMPLE: a cluster-and-truncate approach to solve the crystal structures of small proteins using rapidly computed ab initio models" (2012) Acta. Cryst. D68, 1622-1631

Python dispatchers for CCP4

David G. Waterman
CCP4, Research Complex at Harwell, Rutherford Appleton Laboratory, Didcot OX11 0FA, UK

Introduction

Python is a popular high-level language for scripting in science, offering a high degree of portability with concise and clear source code. The CCP4 Suite contains numerous components written in Python. A typical use case consists of Python 'glue', in which a top level Python program acts as a decision-making script, controlling the execution of various lower level programs that do the intensive calculations. This is the basis behind the expert system, of which MrBump[1] and xia2[2] are two prominent examples in CCP4. Another common use of Python is as a language to write graphical user interfaces, *via* extensions such as PyQt or wxPython. The forthcoming CCP4 GUI, CCP4i2[3], is an example of this, replacing Tcl/Tk with Python/PyQt for a more modern and powerful system.

In those cases where Python is used to wrap lower level executables there is a common need to write very similar 'wrapper' code. Typically this makes use of Python's built-in subprocess module. This is relatively straightforward to use, however care must be taken to ensure that it is used in a portable manner across platforms (there are some differences on Windows compared to Unix-like platforms). There are also some limitations to basic use of subprocess. For example, there is no simple interface by which jobs can be started but control returned immediately to the Python script, which is then free to monitor the output of that job, potentially aborting an unsuccessful run before it completes, or to start new jobs to run in parallel. These problems have been addressed multiple times in the CCP4 suite, as the suite lacked a consistent, common Python interface.

From version 6.4.0, the CCP4 suite will be distributed with a Python package called CCP4Dispatchers, containing wrappers for all of the executables in the suite. These dispatchers are designed with a common, simple, but flexible interface that works in the same way across platforms. It is hoped that writing these wrappers once more lifts the need for some boilerplate code in future CCP4 python projects. In addition, the CCP4Dispatchers package contains features that makes it more powerful than basic use of subprocess.

Automatic code generation

The CCP4 suite is under active development. New programs appear, occasionally old programs are removed, change name or are replaced by versions in a different, interpreted, language. The suite also makes relatively heavy use of environment variables. These may change names and values over time (although the recent trend is to pare back the list of CCP4 environment variables, which grew rather unwieldy over the years). The CCP4Dispatchers package must cope with the need to maintain the Python interface in step with the programs and environment. The effort required to do this is minimised by generating the dispatchers automatically as one of the final steps during installation of the suite. This ensures the dispatchers are up to date with the installed programs, and guarantees that a dispatcher is available for every program present at installation. When changes are made to the suite, e.g. by the automatic updater, the dispatcher package can simply be regenerated as this operation is very quick.

The 300+ programs distributed with CCP4 are diverse. Although the majority are native executables, these are supplemented by shell scripts, interpreted languages such as Python, Tcl, Perl or Ruby and

Java bytecode. This must be detected when the dispatchers are generated to create the correct dispatch command, which avoids the need to rely on the shell to interpret the target type. This is also of benefit to the user of the CCP4Dispatchers package, who need not care what the target type is. As long as it is a CCP4 executable located in the $CBIN directory, it should have a usable Python wrapper automatically written for it.

In CCP4 6.4.0, binary distributions of the suite on Mac or Linux will have the CCP4Dispatchers package generated under $CCP4/share/python by the BINARY.setup script. On Windows the package will be distributed pre-generated, just to make things easier for the installer.

Encapsulating the CCP4 environment

An important feature of the CCP4Dispatchers package is to wrap up everything needed to run a CCP4 job, including a correct set of CCP4 environment variables. These are captured when the package is generated and are automatically set when a dispatcher is created in Python code. This has some interesting consequences.

1. It is beneficial to third-party code, which may need to run CCP4 programs but without the inconvenience of having to set up the environment explicitly. It is only required that a CCP4Dispatchers package is placed in Python's module search path, so that it can be imported. Then the environment is handled automatically prior to runtime of the program.

2. It potentially alleviates the requirement to set the CCP4 environment for command line use too. Rather than start the program directly, a user may call the programs *via* their dispatchers, handling the environment set up implicitly just prior to runtime. In fact, a set of symbolic links (or .bat files on Windows), that have the same names as the underlying executables, are generated in a separate directory along with the CCP4Dispatcher package. If this directory is placed in the PATH then no other variables need to set in the shell in order to run any CCP4 programs.

3. Multiple dispatcher packages can be generated with different environments. This provides a means to access multiple versions of the suite without their environments colliding in the shell.

The second two points above hint at more advanced usage of the dispatcher generator. It may be run outside of the suite install process in order to generate packages with custom locations, names and environment definitions.

Custom use

The binary distributions of the suite come with a canonical set of CCP4 dispatchers, located in $CCP4/share/python. However, customised sets of dispatchers may be generated at any time using Python scripts located in $CCP4/libexec. Dispatcher generation is a two stage process. First a file of environment variable definitions must be created. To simplify this procedure the script envExtractor.py can be run passing in a number of POSIX shell scripts used to set up the environment. On Windows the file of environment variable definitions must be created manually, but the syntax is extremely simple. Secondly, dispatcherGenerator.py is run, with minimal input being the file of definitions and a path to an directory of executables to write dispatchers for.

This two stage procedure makes it easy to modify sets of environment variables prior to dispatcher generation. It also means that the dispatcher generator can remain a completely general tool. In fact, not even the name of the output package 'CCP4Dispatchers' is prescribed. By using options of dispatcherGenerator.py it is possible to write a package of Python dispatchers for any set of

executables, with any valid package name, in a user-specified location (usually in PYTHONPATH), with a set of links (or .bat files) in some other location (usually in PATH).

More detailed information about the use of envExtractor.py and dispatcherGenerator.py are given in the documentation at $CCP4/html/CCP4Dispatchers.html.

A simple example

The following code shows how the example script refmac5-simple.exam may be rewritten in Python, using the CCP4Dispatchers:

```
from string import Template
import os
from CCP4Dispatchers import dispatcher_builder

cmd = Template("HKLIN $CEXAM/rnase/rnase18.mtz " + \
               "HKLOUT $CCP4_SCR/rnase_simple_out.mtz " + \
               "XYZIN $CEXAM/rnase/rnase.pdb " + \
               "XYZOUT $CCP4_SCR/rnase_simple_out.pdb")
cmd = cmd.substitute(os.environ)
keywords = """LABIN FP=FNAT SIGFP=SIGFNAT FREE=FreeR_flag
NCYC 10
END
"""

d = dispatcher_builder("refmac5", cmd, keywords)
d.call()
```

Further examples of use are given in the documentation at $CCP4/html/CCP4Dispatchers.html.

Feedback is very welcome!

References

[1] Keegan, R.M. and Winn, M.D. “*MrBUMP*: an automated pipeline for molecular replacement” (2008) *Acta Cryst*. D**64**, 119-124

[2] Winter, G. "xia2: an expert system for macromolecular crystallography data reduction". (2010) *J. Appl. Cryst.* **43**, 186-190

[3] Potterton, L. "CCP4i2 for Programmers". (2012) *CCP4 Newsletter* **49**.

The DIALS framework for integration software

David G. Waterman[a*], Graeme Winter[b], James M. Parkhurst[b], Luis Fuentes-Montero[b], Johan Hattne[c], Aaron Brewster[c], Nicholas K. Sauter[c], Gwyndaf Evans[b*]

[a]*CCP4, Research Complex at Harwell, Rutherford Appleton Laboratory, Didcot OX11 0FA, UK*

[b]*Diamond Light Source Ltd, Harwell Science and Innovation Campus, Didcot, OX11 0DE, UK*

[c]*Lawrence Berkeley National Laboratory, 1 Cyclotron Rd., Berkeley CA 94720, USA*

**Correspondence email addresses: david.waterman@stfc.ac.uk & gwyndaf.evans@diamond.ac.uk*

Introduction

The field of macromolecular crystallography (MX) has benefited greatly from technological advances in recent years. Automation, high brilliance beamlines at 3rd generation synchrotron sources and high frame-rate pixel array detectors have together enabled extremely rapid collection of most MX datasets. In tandem, large area photon-counting detection, microfocus beams and high quality X-ray optics have brought more challenging projects within reach. Thus there is a clear need for diffraction data analysis software designed to cope with the ever increasing volumes and rates of data collection, and with the developments in experimental methodology, from shutterless, fine-sliced rotation scans through to the thousands of randomly-oriented snapshots of serial crystallography. To match these technological advances it is to be expected that this new software would utilise techniques of parallel processing using multiple CPU and GPU machines, facilitating not just speed, but highly accurate analysis based on a comprehensive underlying physical model. Moreover the current state of the art now does not constitute the peak of progress in this field as there are significant changes yet to come, and modern diffraction integration software must be designed such that flexibility, extensibility and collaboration remain the core principles of the project.

To address the outlined requirements, we are developing DIALS (Diffraction Integration for Advanced Light Sources), a new software project for the analysis of crystallographic diffraction images. The main design goals of DIALS are versatility and modularity. To this end, the various components of a complete package have been identified and are being developed as independent modules where possible, which communicate *via* carefully designed APIs. This reduces bottlenecks caused by design decisions within one module affecting development in others. It also facilitates careful testing and benchmarking of the individual steps characteristic of integration programs, e.g. spot finding, centroid determination, indexing, orientation and geometry refinement, profile determination, profile fitting, integration etc. Thus future developments can clearly be assessed in terms of their impact on final data and structure quality. Furthermore the *toolkit* nature of DIALS will broaden its applicability in the long term beyond rotation method MX and serial MX, allowing excursions into Laue crystallography, neutron crystallography, small molecule crystallography and potentially even fibre diffraction.

At the broadest level, the component modules include, but are not limited to, visualisation of results, visualisation of diffraction images and graphical user interfaces, simulation software, databases for handling of large and multiple datasets, key algorithms for integration with specialisations for pixel-array detectors and for challenging data, and a core framework with an internal description of a diffraction experiment, into which various algorithm modules can be plugged. The EU project BioStruct-X (www.biostruct-x.org) has funded development to address these tasks and produce software that can unify the analysis of both synchrotron and FEL crystallography. Groups from Diamond Light Source, CCP4, the Center for Free Electron Lasers (CFEL), the European XFEL, EMBL Grenoble, the Paul

Scherrer Institute, Dectris and the Lawrence Berkeley National Laboratory in the USA are all contributing to aspects of the final deliverables of this project.

In this newsletter, we focus on one of those tasks, namely the development of the core programming framework with which a diffraction experiment can be described to the computer program. Formally marking this out, rather than letting it develop implicitly as part of an application suited to a particular task, is an important feature of our approach. It avoids the the core modules becoming overly application-oriented. Indeed, the framework is designed to be general with respect to hardware, to diffraction geometry and to choice of experiment, with the same basic elements shared for e.g. rotation method and serial femtosecond crystallography experiments.

Elements of the core programming framework

The framework consists of several elements - models for experimental components, cleanly defined interfaces for analysis steps, and data structures for the storage of analysis results, including reflection shoeboxes, background models and spot profiles. Where possible we are adopting standard experimental descriptions and file formats, for example imgCIF[1] and the model of experimental geometry contained therein, and HDF5[2] for the storage of results and processed data. Figure 1 illustrates how the elements of the framework can be arranged in a workflow for diffraction image integration.

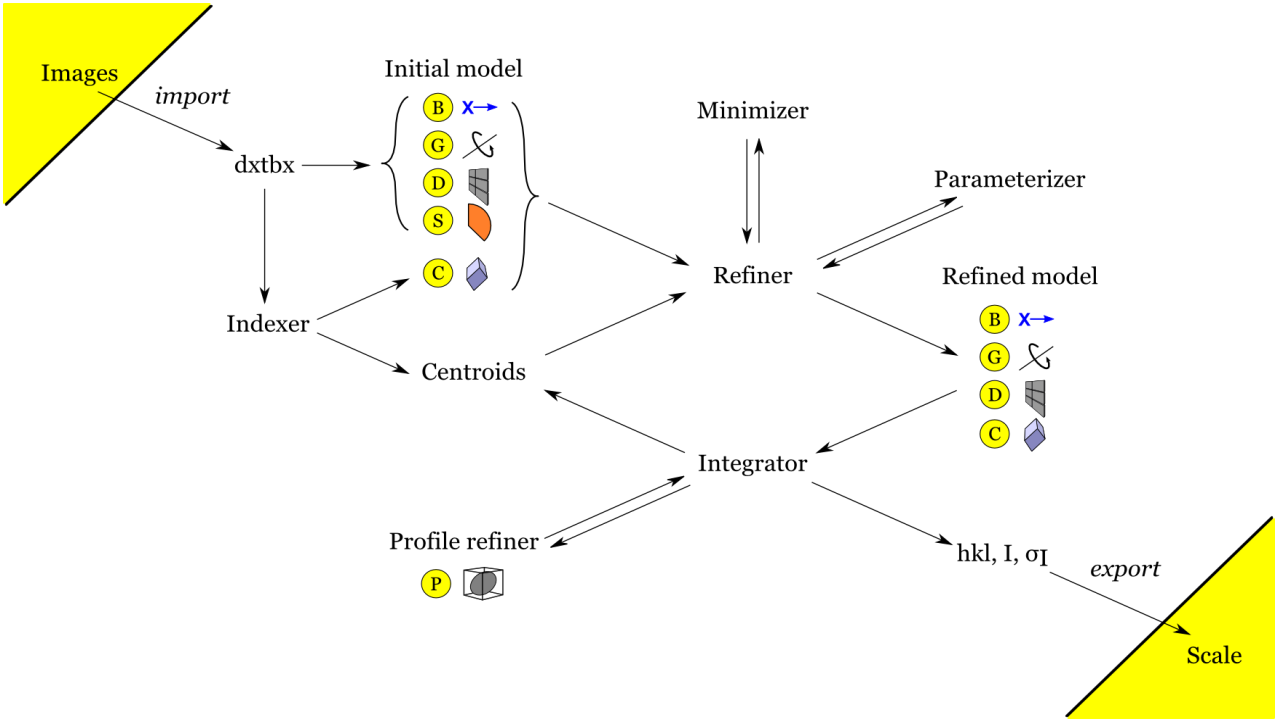


Figure 1: A possible use of DIALS framework modules to integrate diffraction images from a rotation method scan. Initial experimental models for the Beam, Goniometer, Detector and Scan (a contiguous series of rotation images) are constructed by dxtbx by analysis of the image headers and converted to the standard imgCIF frame. An indexer module accesses image data using the dxtbx and performs spot finding and indexing on the images, resulting in an initial Crystal model and calculated spot centroids. The models and centroids are passed to a refinement module. This makes a suitable parameterisation of the models, target function and choice of minimisation engine. After refinement, an improved model of the experimental geometry is passed to an integrator module. This performs spot integration using a suitable 2D or 3D method and includes a Profile forming and refinement module, which could use either empirical learning or *ab initio* synthetic methods to create the model profiles. The construction of spot profiles provides the opportunity to improve on the initial centroids, which may be used again in a second cycle through the refinement procedure. The final basic result from the integrator module is a list of reflections, their intensities and estimated intensity errors, however it is likely that additional information, such as the profile model used during integration, will also be output for potential downstream analysis.

A key objective is to model the analysis in an object-oriented manner, with a clear separation between e.g. the detector description from the sample model. As such there are four main experimental model components: the beam, the detector, the goniometer, and the sample. Each of these has a strictly defined abstract interface. For example, the detector interacts with other elements *via* abstract detector planes, accommodating any area detector that can reasonably be described by an arrangement of two-dimensional panels. Behind this abstract interface more detailed calculations may be performed, for example mapping the millimetre position on the detector surface to the appropriate pixel in the image, perhaps allowing for detector parallax in pixel array detectors or taper distortions in CCDs. Initial values for these models are generated from the headers of the raw image data from the diffraction experiment toolbox, dxtbx.

The dxtbx[3] is an extensible toolkit for providing universal access to X-ray diffraction data from a wide variety of detectors, sources and beamlines, embedding local knowledge to return a standard description of the experiment, corresponding to the imgCIF experimental geometry. In addition this toolbox also provides universal access to the three-dimensional raw pixel data to simplify access to the raw measurements and ensure that the minimum of effort is expended on straightforward elements of the implementation of a data processing toolbox. Finally the system is extensible by users and beamline scientists, such that local knowledge of how image headers should be interpreted can be added and used automatically as a "plug-in".

Collaboration and openness

We have a mandate for the free sharing of our source code, documentation and associated materials with the general public. In the spirit of the open source movement, we welcome collaboration from anyone who wishes to contribute to the project. The framework development is currently a joint effort between developers funded by BioStruct-X, Diamond and CCP4, based in the UK, and developers funded by an NIH technology project: *Realizing new horizons in X-ray crystallography data processing*, based at the LBNL, USA. Our common interests allow us to distribute effort and combine expertise effectively, greatly enhancing our joint productivity and allowing us to tackle the sizeable task of writing a new data processing package within a reasonable time frame.

To avoid "reinventing the wheel" as much as possible, the DIALS project builds on knowledge accumulated over many decades in the field of data processing for MX. We benefit greatly from the altruism of experts who contribute their ideas and advice either directly or *via* their detailed publications on existing algorithms and packages. At the heart of the DIALS framework lies a design philosophy of hardware abstraction and a generalised model of the experiment that is inspired directly by material published on the seminal workshops on position sensitive detector software[4]. Continuing in the spirit of these workshops we hold regular meetings, with talks from invited speakers, and code camps in which specific problems are addressed by intensive effort across the collaboration. Summaries of these meetings and copies of slides given as presentations are available online at http://cci.lbl.gov/dials/.

Our decision to build on existing ideas provides an obvious way to check that our developments proceed in a correct manner, as we can compare results with those from existing software. The early development of the framework code began with a series of use cases bootstrapping from data files created by processing with current software and ensuring our code reproduced acceptably similar results. We are continuing this theme by reproducing published spot finding and integration algorithms used by integration packages such as XDS[5] and MOSFLM[6] and from software used for other types of diffraction image analysis, such as ESMERALDA[7]. Nevertheless, the toolkit architecture implies that we are not limited by these initial methods. We expect new research to lead to more advanced or case-

specific algorithms in future, which may either be implemented by us or by other groups who chose to work with the toolkit.

Source code and timescales

The DIALS framework is being developed in a fully open-source, collaborative environment. We are using Python plus C++, with heavy use of the cctbx[8] for core crystallographic calculations and much infrastructure including a complete build system. Seamless interaction between the C++ and Python components of this *hybrid system* is enabled by boost.python. Python provides a useful ground for rapid prototyping, after which core algorithms and data structures may be transferred over to C++ for speed. High level interfaces of the hybrid system remain in Python, facilitating further development and code reuse both within DIALS and by third parties.

The DIALS framework has a home on sourceforge http://dials.sourceforge.net/. A beta release of DIALS is planned for late 2014 with a final release following in 2015. This complete diffraction integration package will be distributed by CCP4.

Acknowledgements

JMP and LF-M were supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under BioStruct-X (grant agreement N°283570). N.K.S., A.B., and J.H. were supported by National Institutes of Health/National Institute of General Medical Sciences grants 1R01GM095887 and 1R01GM102520, as well as by the Director, Office of Science, Department of Energy under Contract DE-AC02-05CH11231.

References

[1] Bernstein, H. J. & Hammersley, A. P. (2005). *International Tables for Crystallography*, Vol. G, *Definition and Exchange of Crystallographic Data*, 37-43.

[2] HDF Group (2010). *Hierarchical Data Format Version 5*, http://www.hdfgroup.org/HDF5.

[3] Parkhurst *et al*. (2013), in preparation.

[4] Bricogne, G. (1987). *Proceedings of the CCP4 Daresbury Study Weekend*, pp. 120-145.

[5] Kabsch, W. (2010). *Acta Cryst.* D**66**, 125-132.

[6] Leslie, A. G. W. and Powell H. R. (2007), *Evolving Methods for Macromolecular Crystallography*, **245**, 41-51. ISBN 978-1-4020-6314-5.

[7] Fuentes-Montero, L., Cermak, P. & Rodriguez-Carvajal, J., http://lauesuite.com, in preparation.

[8] Grosse-Kunstleve, R. W., Sauter, N. K., Moriarty, N. W., & Adams, P. D. (2002). *Journal of Applied Crystallography*. **35**, 126–136.

SynchLink:

an IOS app for viewing data interactively from synchrotron MX beamlines

Helen Ginn[a,b], Ghita Kouadri Mostefaoui[a], Karl Levik[a], Jonathan M. Grimes[a,b], Martin A. Walsh[a], Alun Ashton[a], David I. Stuart[a,b]

[a]Diamond Light Source Ltd, Diamond House, Harwell, Didcot, Oxfordshire, OX11 0DE

[b]Division of Structural Biology, The Wellcome Trust Centre for Human Genetics, University of Oxford, Roosevelt Drive, Headington, Oxford, Oxfordshire OX3 7BN

Correspondence to: scisoftjira@diamond.ac.uk

ISPyB[1] (Information System for Protein crystallography Beamlines) is a Laboratory Information Management System (LIMS) developed to manage Synchrotron radiation macromolecular crystallography experiments. ISPyB provides features that allow users to register sample information for a particular experiment, manage the shipment of samples and collate data collection and processing parameters for each experiment providing a portal for users to view current past and future visits to the synchrotron. ISPyB is currently used at the ESRF and Diamond for MX experiments and, at Diamond, currently extends to presenting results of automated processing results from XIA2[2] and FastDP[3]. The system is accessible to the user remotely through a web based interface.

In January 2013 Diamond officially released V1.0 of SynchLink, an iPhone/iPad (iOS v6 onwards) app which allows users of Diamond MX beamlines to monitor, re-visit, and search information on their data collections and automated data processing results recorded in ISPyB[1].

Introduction

Smartphones and tablets have significant advantages for use in a variety of industrial sectors where there is a high level of mobility of workers. For example, the healthcare industry was an early adapter to mobile devices and uptake increased steadily from their introduction in the late nineties[5].

Despite their increased processing, data storage and graphical capabilities few use-cases have thus far allowed us to exploit the technology in the structural biology workflow. There are some lightweight applications for chemical information, 3D molecular graphics viewers and for monitoring synchrotron and beamline status, but due to either the data-heavy nature of processing images or computational intensive nature of structure solution, few applications are yet available.

However, the obvious advantages of using small easily portable mobile devices for MX synchrotron users was clear and the availability of an Information-rich database ISPyB which is populated by data acquisition software and automated data processing and structure solution pipelines brought about the requirement for an app that could exploit these past developments.

Here, we have developed an IOS v6+ app to allow MX users of Diamond to interact with ISPyB. We chose IOS in the first instance to mitigate the number of variables by limiting the number of hardware to support to iPhone/iPad (iOS v6+) and use the native Mac development tools i.e. Objective-C language and the XCode toolset as a development environment.

Methods

As with many software development projects, the user interface is often all a user will see and its reliability, intuitiveness and speed of interaction will determine the success of the project overall. Fortunately, given a populated data source, this initiative was able to focus on the three main components to achieve this: 1) the app itself on the device and how it interacts with 2) the webservices servicing the data and 3) interfacing with the data sources. Moreover, the workflows and the way the data are presented to the user were reassessed and optimised for current expected use.

GUI

The GUI was developed as a series of windows into the data exposing more and more detail on the user request whilst minimising the number of windows to navigate before reaching the data. We worked hard to limit the information displayed to what was necessary, to avoid overloading the user. Figure 1 shows a montage of some of the screens as they would appear on an iPhone, along with a schematic representation of the navigation.

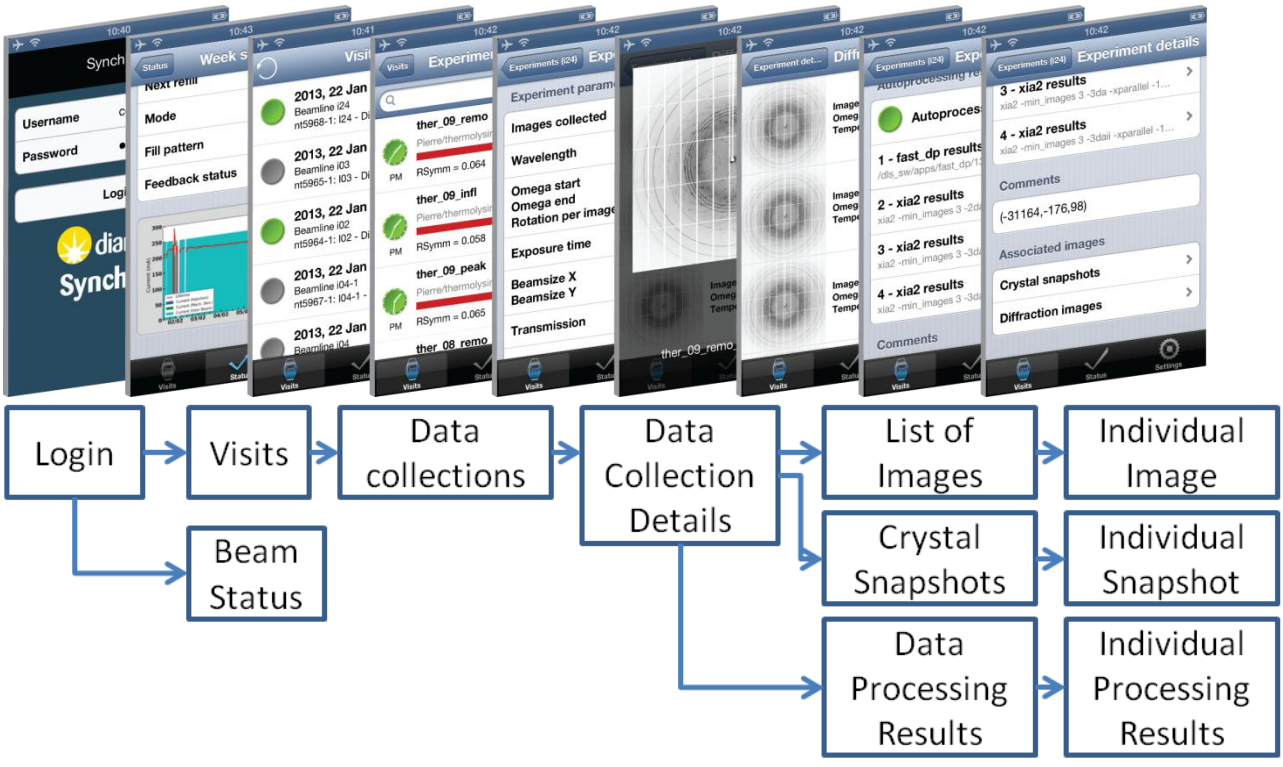


Figure 1; Screenshot's from SynchLink as shown on an iPhone and the main user workflow.

As previously outlined, displaying useful information quickly was vital to distract the operator from potentially longer running data transfers caused by either slow networks or simply volume of data. To this end the GUI was developed to prioritise the display of information for

the current screen before downloading off-screen data (figure 2). Additionally a search option was create to select only visits and data collections matching the search key.

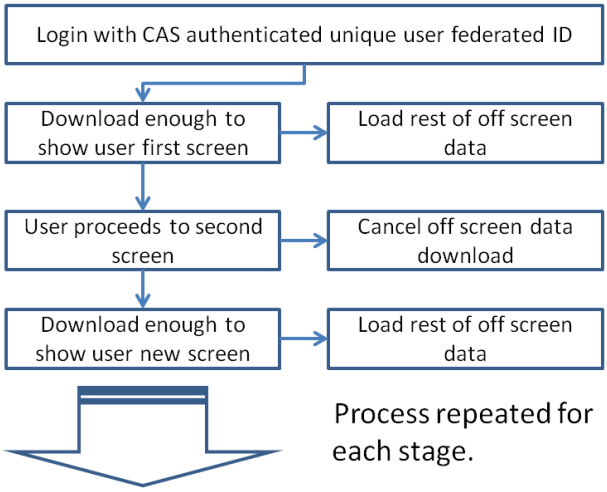


Figure 2; GUI workflow on how the app can remain responsive to user interaction.

WebServices

A number of web service technologies were explored, but eventually a JSON web service developed using Java/Axis2 and running on an Apache Tomcat server was chosen due to its limited data exchange size, thereby increasing the responsiveness of the client app. A generic structure was developed for these webservices based on the database tables and its structural linking as this ensured maximum flexibility to the application developer whilst authentication and authorisation to the data was applied server side. As a continuous session could not be held open with a mobile device (either no support or a draining of battery power) a server side session token is shared with the app upon successful login/authentication. This is valid for 24 hrs and used to authenticate the request from the device without having to fully authenticate against the facility's CAS (Central Authentication Service) mechanism.

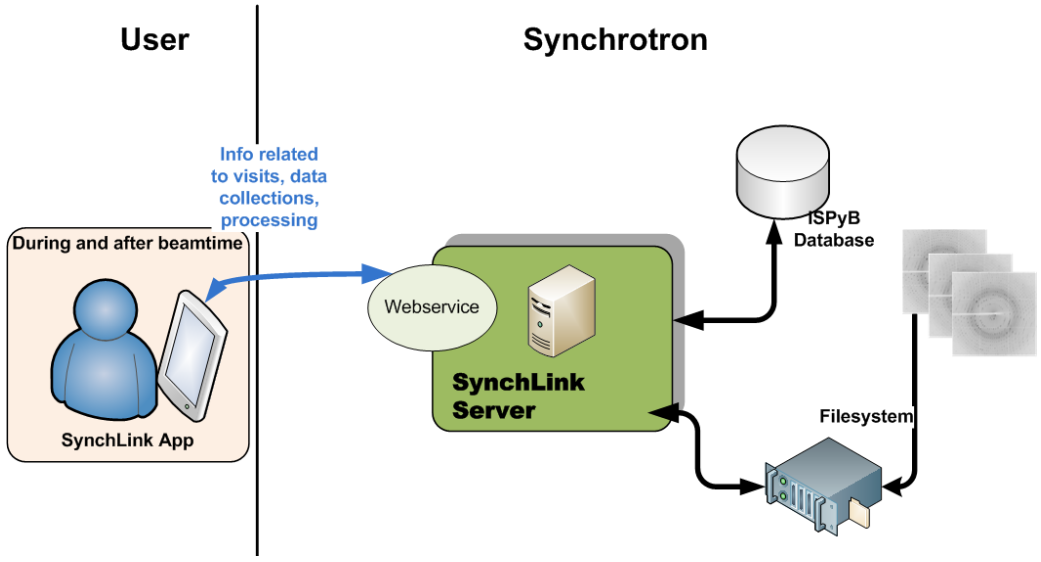


Figure 3: A schematic view of the SynchLink webservice and app Client

Message encryption between the SynchLink webservice and the app is achieved using SSL certificates. The transmission confidentiality is important given the sensitivity of data collected by some synchrotron users.

The webservice is implemented using the Apache Axis2/Java framework and the server is hosted on a machine internal to the synchrotron network. The webservice is accessible through an URL accessible externally that redirects to the internal server machine. This choice, allows more freedom in moving the webservice to other internal machines (for

maintenance or upgrade) but also to limit access to the server only to clients of the webservice.

The webservice has direct access to the ISPyB database and the synchrotron filesystem achieving a perfect encapsulation layer of the synchrotron resources (figure 3).

To aid performance, JSON data-interchange format is used and gzip compression is performed on the exchanged messages in order to reduce the size of data.

Integration with the facility site

Key to the app's success is the ISPyB[1] database (https://forge.epn-campus.eu/projects/ispyb3) and its continuous population from the user office scheduling software, data acquisition software (www.opengda.org) and Diamond's automated data processing pipelines[2,3,4]. The way ISPyB integrates within a facility is outlined elsewhere[1], but the use of Synchlink integrates seamlessly into the user experience as summarized in figure 4 below.

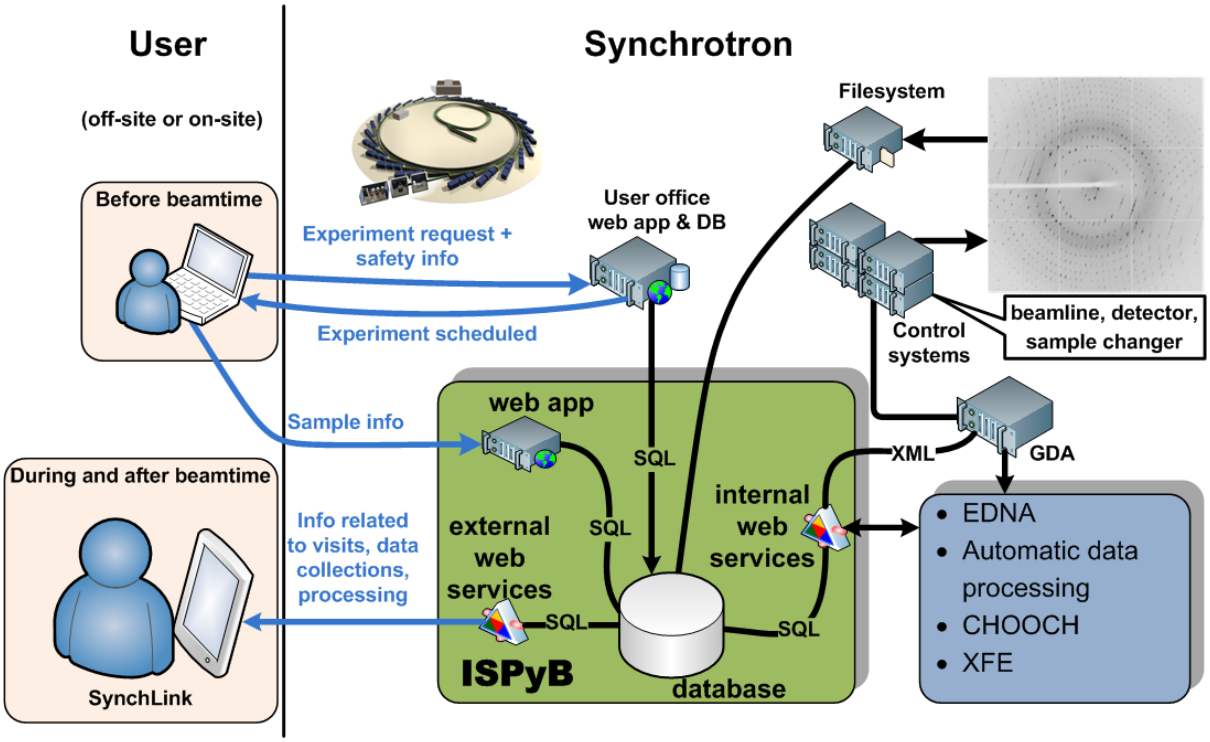


Figure 4; Schematic representation of how SynchLink integrates with facility services.The figure has been modified from Delageniere et al[1]

Availability

The app is available from the App Store and is free to download, though the data charges of your network provider will apply. Nearly 200 downloads have already been recorded. The ISPyB database and frameworks are available to participants of the ISPyB collaboration.

Future

There are many potential developments from V1.0 in increasing the functionality and science coverage of the current app, including support of other devices, such as Android-based.

Acknowledgements

The app was developed by Helen Ginn, a very talented Oxford Biochemistry undergraduate during a summer internship. Helen was supervised by Dave Stuart at Oxford and Martin Walsh/Alun Ashton at Diamond. There was also expert input from Jonathan Grimes at Oxford, and Ghita Kouadri Mostefaoui and Karl Levik at Diamond who developed the webservices and integration to ISPyB. Francoise Cesmat and Bill Pulford also provided valuable help during the licensing and registration process in releasing the app. Our collaboration with the ESRF on the ISPyB project provided the vehicle holding the information for the project.

References

1) ISPyB: an Information Management System for Synchrotron Macromolecular Crystallography (2011) S. Delageniere , P. Brenchereau , L. Launer , Alun Ashton , R. Leal , S. Veyrier , J. Gabadinho , E. J. Gordon , S. D. Jones , Karl Levik , S. M. McSweeney , S. Monaco , M. Nanao , D. Spruce , O. Svensson , Martin A. Walsh , G. A. Leonard Bioinformatics DOI: 10.1093/bioinformatics/btr535
2) xia2: an expert system for macromolecular crystallography data reduction (2010) Graeme Winter Journal of Applied Crystallography (Vol: 43, Pages: 186-190) DOI: doi:10.1107/S0021889809045701
3) Automated data collection for macromolecular crystallography. (2011) Graeme Winter , Katherine McAuley Methods PMID: 21763424
4) EDNA: a framework for plugin-based applications applied to X-ray experiment online data analysis (2009) Marie-Francoise Incardona , Gleb Bourenkov , Karl Levik , Romeu Pieritz , Alexander Popov , Olof Svensson Journal of Synchrotron Radiation (Vol: 16 (6), Pages: 872-879) DOI: 10.1107/S0909049509036681
5) Who's using PDAs? Estimates of PDA use by health care providers: a systematic review of surveys. Garritty C, El Emam K: Journal of medical Internet research 2006, 8:e7

Integrating crystallographic structure determination and visualization:
remote control of Coot

Nathaniel Echols and Paul D. Adams
Lawrence Berkeley National Laboratory, Berkeley CA
correspondence: nathaniel.echols@gmail.com

Macromolecular structure building and refinement is typically an inherently visual task, requiring at the very least interactive validation of the final model, and ideally some degree of monitoring throughout the process. Most building and refinement programs, however, have traditionally been largely decoupled from direct visualization aside from the output of files suitable for 3D display. Although a few projects have dedicated molecular graphics viewers (Langer et al. 2013, Turk 2013), the overhead for both the programmer (who must develop and maintain the code) and the user (who must learn a separate interface) is substantial. An alternative, simpler approach is to leverage an existing viewer (see for instance Payne et al. 2005).

The ubiquitous crystallography model-building program Coot (Emsley et al. 2010) has several advantages which make it an obvious and attractive frontend for automated structure determination software:

- distribution under an open-source license (GPL)
- binary distributions for the most commonly used operating systems
- Python and Scheme scripting support

The latter feature facilitates the development of plug-ins which extend the functionality of Coot, allowing manipulation and augmentation of the graphical interface using either the PyGTK or GUILE scripting APIs, which wrap the GTK GUI library (http://www.gtk.org). Python (http://www.python.org) in particular provides a rich set of built-in functions, including support for commonly used network protocols such as HTTP. Importantly, the use of scripted plugins does not require modification of the source code or separate distribution of Coot itself.

The Phenix software (Adams et al. 2010) is a Python-based crystallography suite encompassing most common procedures in structure determination (with the exception of processing of raw diffraction images). We have placed an emphasis on ease of use and automation, including the development of a modern graphical interface (Echols et al. 2012). Although we have found it convenient to use a simple Python-based OpenGL molecular viewer within the Phenix GUI, the development of a full-fledged building program is far outside the scope of the project, and the existence of Coot has made this both unnecessary and undesirable. Inspired by an existing PyMOL feature[1] and the molecular dynamics viewer VMD (Humphrey et al. 1996), we therefore

[1] G. Landrum, unpublished; http://sourceforge.net/p/pymol/code/HEAD/tree/trunk/pymol/modules/pymol/rpc.py. Note that this module also now registers most of the function calls automatically by extracting their names from the cmd module.

implemented an extension to Coot which enables direct control from external programs using the XML-RPC protocol (http://xmlrpc.scripting.com/spec.html). Although the use of TCP/IP sockets raises several practical issues, the protocol is sufficiently flexible to make fully interactive communication possible on all three common desktop operating systems (Mac, Linux, and Windows), and is easily adapted to any program capable of running an XML-RPC client, including software written in Python, C++, and Java.

Overview of XML-RPC

The XML-RPC (eXtensible Markup Language Remote Procedure Call) protocol is a simple client-server framework for executing remote code in a platform- and language-independent manner. Other similar frameworks are available, the most popular being SOAP (Simple Object Access Protocol), but we chose XML-RPC primarily because it is installed by default with Python, and because the more advanced features of SOAP were deemed unnecessary for this application. The underlying communication is handled via HTTP, with the server handling POST queries containing formatted XML (example below), which encapsulates the method calls:

```
<?xml version='1.0'?>
<methodCall>
<methodName>read_pdb</methodName>
<params>
<param>
<value><string>/Users/nat/data/lysozyme.pdb</string></value>
</param>
</params>
</methodCall>
```

The use of XML imposes some obvious limitations on the type of information which may be exchanged. Any binary content or other special characters which are unacceptable to the XML parser are prohibited; this includes Python pickles (serialized objects). In practice, if a common filesystem is available it is usually possible to use intermediate files (PDB or MTZ), with the XML-RPC calls restricted to instructions to load files.

Definition of the remotely callable methods is left to the application developer; individual methods must be registered with the XML-RPC server. Typically the method names in the application will be identical to the remote calls, although this is not a requirement. Arguments are limited to objects which may be embedded in XML, which excludes binary data. A minimal example of paired client and server Python code is shown below.

Server:

```
from iotbx.pdb import hierarchy
import SimpleXMLRPCServer

def read_pdb (file_name) :
```

```
    pdb = hierarchy.input(file_name=file_name)
    pdb.hierarchy.show()

server = SimpleXMLRPCServer.SimpleXMLRPCServer(
    addr=("localhost", 40000))
server.register_function(read_pdb, "read_pdb")
server.serve_forever()
```

Client:

```
import xmlrpclib
import sys
remote = xmlrpclib.ServerProxy("http://localhost:40000/RPC2")
remote.read_pdb(sys.argv[1])
```

For interactive applications, it is necessary to handle the remote requests without blocking the main execution thread. This can be handled in several ways depending on the design of the program being extended. If the Python interpreter itself is executed, as opposed to being embedded as a library in a monolithic C or C++ executable, the threading module may be used to handle server requests separately from the main thread. However, the programmer must still exercise caution when calling methods which may not be thread-safe, for example when updating a GUI[2]. Toolkits such as GTK and wxWidgets (or in the Phenix case, PyGTK and wxPython) allow use of timers to execute function calls at regular intervals. Therefore, instead of calling `server.serve_forever()`, one can instead call `server.handle_request()` periodically with a timer.

Application to Coot

Because Coot has the ability to execute arbitrary Python code, with a large number of functions available for loading and manipulating molecule and map objects, application of XML-RPC is relatively straightforward. However, we found it preferable to avoid the requirement of registering each of the many functions in the Coot API individually, especially since this would not automatically track with changes in Coot. A convenient solution is to simply use getattr() to retrieve methods by name from the coot module. Additionally, by specifying a class containing additional custom methods, these too can be automatically made accessible via XML-RPC.

Although Coot will not run a continuous, separate, Python thread which only listens on a socket, objects which are not garbage-collected will remain persistent in memory. Therefore, we can use the timeout function from the gobject module (part of PyGTK) to handle incoming requests.

[2] The design of PyMOL actually allows the child thread used to handle server requests to run any API function without crashing the program, but this requires additional logic coded in C, and would not normally apply to programs using Python GUI toolkits. We note in passing that the same principles described in this article may apply to other Python-enabled molecular graphics such as UCSF Chimera or VMD. However, the problem of how to handle XML-RPC requests more or less continuously without crashing the GUI must be handled differently in each case (and a preliminary attempt to write a Chimera plugin was unsuccessful for this reason).

We have found 250ms to be a reasonable interval, as it is short enough to make the latency barely noticeable during interactive control of Coot from the Phenix GUI, but does not appear to affect the performance of Coot itself. (A timeout of 10ms, on the other hand, makes Coot nearly unusable.)

The core of the implementation, shown below, subclasses the `SimpleXMLRPCServer` class, overriding the `_dispatch` method in order to extract the named methods. Failure to locate the desired function raises an exception. If an exception is caught for any other reason, it is re-raised with the original traceback embedded in the message string. This permits the client code to present full debugging information to the user and/or developer. Otherwise, the return value of `_dispatch` will be received by the client.

```
import coot
import gobject
import SimpleXMLRPCServer
import sys

class coot_server(SimpleXMLRPCServer.SimpleXMLRPCServer) :
  def __init__ (self, interface, **kwds) :
    self._interface = interface
    SimpleXMLRPCServer.SimpleXMLRPCServer.__init__(self, **kwds)

  def _dispatch (self, method, params) :
    if not self._interface.flag_enable_xmlrpc :
      return -1
    result = -1
    func = None
    if hasattr(self._interface, method) :
      func = getattr(self._interface, method)
    elif hasattr(coot, method) :
      func = getattr(coot, method)
    if not hasattr(func, "__call__") :
      print "%s is not a callable object!" % method
    else :
      try :
        result = func(*params)
      except Exception, e :
        traceback_str = "\n".join(traceback.format_tb(
           sys.exc_info()[2]))
        raise Exception("%s\nOriginal traceback:%s" % (str(e),
           traceback_str))
      else :
        if result is None :
          result = -1
    return result
```

(Note that in the quite common case that the return value is None, the server instead returns -1, due to limitations inherent to Python's implementation of the protocol.)

The second half of the implementation actually starts the XML-RPC server, and defines additional methods which will automatically become part of the remotely callable API:

```
class coot_interface (object) :
  def __init__ (self, port=40000) :
    self.flag_enable_xmlrpc = True
    self.xmlrpc_server = coot_server(
      interface=self,
      addr=("127.0.0.1", port))
    self.xmlrpc_server.socket.settimeout(0.01)
    self.current_imol = None
    print "xml-rpc server running on port %d" % port
    gobject.timeout_add(250, self.timeout_func)

  def timeout_func (self, *args) :
    if (self.xmlrpc_server is not None) :
      self.xmlrpc_server.handle_request()
    return True

  def update_model (self, pdb_file) :
    if (self.current_imol is None) :
      self.current_imol = read_pdb(pdb_file)
    else :
      clear_and_update_molecule_from_file(self.current_imol,
        pdb_file)

  def recenter_and_zoom (self, x, y, z) :
    set_rotation_centre(x, y, z)
    set_zoom(30)
    graphics_draw()
```

In this example, the method `update_model` reloads a model as desired (updating the molecule object rather than creating a new one), for instance showing the progress of automated model-building or refinement. The `recenter_and_zoom` method is used in a number of contexts in the Phenix GUI, in particular the MolProbity interface (Chen et al. 2010), to associate tables and plots of residue properties with the Coot viewport.

A more complex, fully functional implementation of this method is distributed as part of CCTBX in the `cootbx` directory, with the file name `xmlrpc_server.py`. It may be invoked at launch time as:

```
coot --script xmlrpc_server.py
```

We have found it most convenient for the user if the remote control can be disabled and resumed at any time; therefore the actual implementation also adds a button to the Coot toolbar which toggles the connection.

Once Coot is started with the XML-RPC server running, other Python code can connect using xmlrpclib.ServerProxy and call the full range of API functions (built-in or custom), e.g.:

```
import xmlrpclib
coot = xmlrpclib.ServerProxy("http://127.0.0.1:40000/RPC2")
coot.read_pdb("model.pdb")
coot.auto_read_make_and_draw_maps("maps.mtz")
```

Other programming languages will work equally well provided an XML-RPC library is available.

Practical considerations, limitations, and workarounds

Although the above description is reasonably simple and foolproof, a number of problems became immediately apparent upon testing in real-world situations. Chief among these was the requirement that Coot be in a running state before remote calls can be initiated. Using the XML-RPC protocol exactly as intended meant that buttons intended to open the results of a program run would either cause the Phenix GUI to block for several seconds while Coot was launched, or would not actually perform the desired action until clicked a second time. As neither behavior was ideal for an intuitive user interface, it was necessary to find a way to deal with the lag time transparently.

Our eventual solution leaves much to be desired from a purist perspective, as it essentially reduces the XML-RPC communication to a one-way system. The client code is invoked as soon as Coot is started, but well before the XML-RPC server is started; when the remote calls inevitably fail due to connection refusal, the resulting exception is suppressed and the request cached by the modified client. A timer function in the Phenix GUI is used to attempt to flush the cache (again, every 250ms) until successful. In practice this appears nearly seamless to the end user (aside from the delay associated with launching Coot), but as there is no guarantee that any given query will be immediately successful, the return value from the server cannot be reliably obtained.

Several issues were only partially resolved. On many systems, especially running Linux, the use of sockets was (surprisingly) unreliable despite connecting only on the local host. This led to a number of connection errors even when Coot was running, which had to be specifically ignored. Socket errors also occur when the desired TCP/IP port is unavailable, either due to use by another program, or a previous instance of Coot. Therefore our module also chooses the port number from a large range at random, and communicates this to Coot via an environment variable.

Filesystem-based alternatives

Although we believe that the flexibility of the XML-RPC approach makes it a compelling choice for inter-program communication, it is not necessarily suitable for applications where Coot must accept input from a remote host - for instance when the external software is being run on a managed cluster. In such situations, or any other application where the use of sockets is either undesirable or unnecessary, a simpler method based on monitoring of file modification times can be equally effective. Again, the timeout function is key; a minimal example is shown below:

```
import os.path

class watch_model (object) :
  def __init__ (self, file_name, timeout=2000,model_imol=None) :
    self.file_name = file_name
    self.model_imol = model_imol
    self.last_mtime = 0
    import gobject
    gobject.timeout_add(timeout, self.check_file)

  def check_file (self) :
    import coot
    if os.path.exists(self.file_name) :
      file_mtime = os.path.getmtime(self.file_name)
      if file_mtime > self.last_mtime :
        if self.model_imol is not None :
          clear_and_update_molecule_from_file(self.model_imol,
            self.file_name)
        else :
          self.model_imol = read_pdb(self.file_name)
        self.last_mtime = file_mtime
```

To use in the context of a program which regularly updates a PDB file, the calling code must simply write a script invoking the watch_model class with the appropriate file name. When run in Coot, the script will then reload the PDB file whenever the modification time changes (with a minimum frequency of every 2 seconds). As above, care must be taken when operating on NFS mounts. An example of this approach is also available in the `cootbx` directory as `watch_file.py`, including a more complex class which also reads a file containing map coefficients with standard labels. The applicability is not limited to PDB (or map) files; the target file could just as easily be a dynamically generated Python (or Scheme) script instead. Note that because this method does not require the use of any APIs in the underlying program beyond the ability to write a script file, it is suitable for use in software based on FORTRAN or other environments where the XML-RPC protocol is unavailable or unwieldy.

Availability

With the exception of features which are specific to the Phenix GUI, the majority of the implementations detailed above are part of the CCTBX, and we encourage their use and redistribution without restrictions as long as the original copyright is included (see http://sourceforge.net/p/cctbx/code/HEAD/tree/trunk/cctbx/LICENSE_2_0.txt for full license terms).

Acknowledgments

We are grateful to Joel Bard for suggesting the use of the timeout function, Paul Emsley and Bernhard Lokhamp for technical advice, and William Scott and Charles Ballard for debugging issues with Coot binary distributions. Funding was provided by the NIH (grant # GM063210) and the Phenix Industrial Consortium.

References

Adams, P. D., Afonine, P. V., Bunkóczi, G., Chen, V. B., Davis, I. W., Echols, N., Headd, J. J., Hung, L.-W., Kapral, G. J., Grosse-Kunstleve, R. W., et al. (2010). Acta Crystallographica. Section D, Biological Crystallography. 66, 1–9.

Echols, N., Grosse-Kunstleve, R. W., Afonine, P. V., Bunkóczi, G., Chen, V. B., Headd, J. J., McCoy, A. J., Moriarty, N. W., Read, R. J., Richardson, D. C., et al. (2012). J. Appl. Cryst (2012). 45, 581-586.

Emsley, P., Lohkamp, B., Scott, W. G., & Cowtan, K. (2010). Acta Crystallographica D. 66, 486–501.

Humphrey, W., Dalke, A., & Schulten, K. (1996). J. Molec. Graphics 14, 33-38.

Langer, G. G., Hazledine, S., Wiegels, T., Carolan, C., & Lamzin, V. S. (2013). Acta Crystallographica D. 69, 635–641.

Payne, B. R., Owen, G. S. & Weber, I. (2005). Computational Science - Iccs 2005, Pt 1, Proceedings 3514, 451-459.

Turk, D. (2013). Acta Crystallographica D. 69, 1–16.

Introduction to R for CCP4 users

James Foadi
Membrane Protein Laboratory – Imperial College London and Diamond Light Source Ltd
Harwell Science and Innovation Campus, Didcot, Oxfordshire OX11 0DE, UK

1. Introduction.

Protein crystallographers are used to assess their data through the analysis of specific statistics and graphics. It is, for instance, the norm to report R_{meas}, completeness, I over sigI and multiplicity as overall numbers and in plots as functions of resolution, when presenting datasets quality in research papers. In CCP4 access to such statistics is made available through log files, log graphs or annotated logs in a web browser. This is certainly more than enough for the average requirement of users. There might be several situations, though, where the analysis provided by individual programs is not sufficient. In such situations users prefer to scrutinize data by themselves rather than to rely on the available statistical analysis. While accessing data is relatively straightforward in the CCP4 suite, flexible programs that allow data analysis are mostly missing. Such analysis should be carried out with the help of *ad hoc* software which would take a considerable amount of time to code, not to mention the programming skills required. This is certainly a task beyond the possibilities and interest of the average CCP4 user.

An obvious way out is the utilization of generic and flexible platforms for statistical data analysis. Many packages have been developed for the purpose, and some of them have become well established in the communities of professional statisticians and people making heavy use of statistics [1, 2, 3, 4, 5]. Among these many packages, the *R software* [4] stands out as one of the most popular and flexible environment for statistical calculations. R has been spreading beyond the international community of statisticians in many academic disciplines. The main reason for this is the open-source nature of the platform, joined to an informative and collaborative spirit present within the community of R enthusiasts. Furthermore, the amount of documentation and freely-available training material for this software is vast and increasing [4]. Furthermore, there are hundreds of add-on packages for a large variety of tasks and disciplines available for download, free of charge.

In crystallography R appears to be timidly used as a side program to simulate or test data, and to carry out all sorts of statistical analysis. It is probably also employed to generate publication-quality graphs. For instance a quick search of the IUCr-journals website returns 9 hits for papers where R has been explicitly referenced [6]. One of the reasons why R is still not used in crystallography as abundantly as in other fields is possibly connected with the specific nature of crystallographic file formats. R can read numerous formats, but R packages-that deal with crystallography have either been not completed [7, 8] or are entirely absent from the pool of available R packages.

In this short article it will be shown how simple and involved analysis can be carried out without much effort thanks to the powerful and widely applicable R built-in functions. The main purpose of this article is to encourage crystallographers to use R for data analysis and graphical data representation.

2. R in a (very small) nutshell

R can be installed on all common platforms including Windows, Mac OS X and Linux. The examples below were generated using R version 2.12.1 installed on Linux Ubuntu 12.04LTS.
R can be started typing “R” in the terminal. A welcoming message, similar to the one shown here,

is displayed:

```
R version 2.12.1 (2010-12-16)
Copyright (C) 2010 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]
```

From this message we can learn how to use the help system. There are two options, in-line help and help via a browser. Data created during a session are stored in the computer active memory, in an area known as *workspace*. Anything created in the workspace is referred to as an *object*. Commands are more correctly known as *functions*. Function “ls” can be used to view what objects are available in the workspace. For instance since nothing was created in the current R session, the following output is obtained:

```
> ls()
character(0)
```

“ls” is followed by brackets, “()”; all functions need brackets. They can include numbers, characters or other R objects, but they must always be present, even if they do not require any argument. Function “ls” returns a vector whose elements are characters indicating the name of the objects in the workspace; in the above case there are no objects and “ls” returns a message saying that the vector character has size zero. Once objects start filling the workspace, the “ls” output changes. For instance let us create 10 gaussian deviates and the alphabet lowercase letters up to letter “h”:

```
> obj1 <- rnorm(10,mean=1,sd=2)
> length(obj1)
[1] 10
> obj2 <- letters[1:8]
> length(obj2)
[1] 8
> length(letters)
[1] 26
```

“rnorm” is a function to generate gaussian random deviates. R has several built-in random generators for a variety of probability distributions. In the above example the 10 numbers created by this function are stored in the object “obj1” using the *assign* operator “<-” (a “less” symbol followed by a “minus” symbol). The “obj2” object is a character vector containing the first 8 lowercase alphabet letters; these are extracted by the global constant “letter”, which is a character vector containing the 26 lowercase alphabet letters. This time the “ls” function will return a

character vector with 2 names, as the current workspace has been filled with two objects:

```
> ls()
[1] "obj1" "obj2"
```

Many built-in functions exist to load data inside the workspace. One of the most used data readers is the “read.table” function. This is suitable to read equal-length columns of numbers or characters. In the following example we read in the content of an mtz file, previously converted into an ascii file with the CCP4 program “mtz2various”:

```
> mtzdata <- read.table("insulin.dat")
> str(mtzdata)
'data.frame': 7007 obs. of  8 variables:
 $ V1: int  0 0 0 0 0 0 0 0 0 0 ...
 $ V2: int  1 1 1 1 1 1 1 1 1 1 ...
 $ V3: int  3 5 7 9 11 13 15 17 19 21 ...
 $ V4: num  873 4896 1384 5135 438 ...
 $ V5: num  12.8 44.4 11.8 42.7 4.8 ...
 $ V6: num  295 700 372 717 209 ...
 $ V7: num  2.17 3.17 1.58 2.98 1.15 ...
 $ V8: int  2 19 17 8 9 11 18 4 11 13 …
```

Data are loaded in the object named “mtzdata”. The content of this object can be explored and summarised with the function “str”. The object is of the type known as *data.frame*. This is the typical container for statistical datasets. A data.frame is very similar to a matrix with some annotations included. In other words, a data.frame is a matrix with mixed data types. Each column in the “mtzdata” data.frame has a name. All column names can be easily visualised:

```
> colnames(mtzdata)
[1] "V1" "V2" "V3" "V4" "V5" "V6" "V7" "V8"
```

These are generic names, as no headers were included in the original file. We know the correct names from the CCP4 original mtz file. Column labels can, thus, be assigned as follows:

```
> colnames(mtzdata) <- c("H","K","L","IMEAN","SIGIMEAN","F","SIGF"," FreeR_flag" )
```

The first 5 rows of the data.frame bears some similarity with part of the mtz content, as displayed by the program “mtzdump”:

```
> mtzdata[1:5,]
   H  K  L     IMEAN    SIGIMEAN         F     SIGF    FreeR_flag
1  0  1  3  873.04822   12.79639  295.43994  2.16591  2
2  0  1  5  4895.71387  44.41380  699.62976  3.17424  19
3  0  1  7  1383.53491  11.76934  371.94446  1.58220  17
4  0  1  9  5135.15283  42.72186  716.54718  2.98121  8
5  0  1 11  437.75629   4.80070   209.21368  1.14740  9
>
```

An important feature of R is the ability to produce simple, intermediate and complicated graphics of publication quality. Let us create a simple plot using simulated data. To simulate the behaviour of a response variable “y”, function of an explanatory variable “x”, we first generate a regular grid of 50 points, say between 0 and 10, using the function “seq”:

```
> x <- seq(0,10,length=50)
> str(x)
 num [1:50] 0 0.204 0.408 0.612 0.816 ...
```

Then a linear relation is imposed on y with added gaussian noise:

```
> y <- 2*x+1+rnorm(50, mean=0, sd=1)
```

The above line, among other things, shows that operations in R are vectorised. “x” and “y” are vectors containing 50 numbers each; when “2*x” is typed all 50 numbers are multiplied by two. Furthermore, adding 1 in this context means adding 1 to all 50 numbers. The expression is completed by adding 50 gaussian random deviates with mean 0 and standard deviation 1 to the 50 modified values. The linear relation can be visualised with a plot using the “plot” function (Figure 1):

```
> plot(x, y)
```

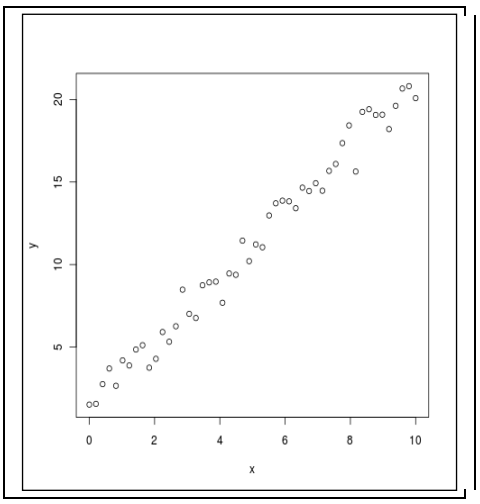


Figure 1

The “plot” command with no parameters produces the basic plot shown in Figure 1. We can enrich this plot. For instance, we can add a title using the keyword “main”, replace the open circles with full circles using the keyword “pch” and add the straight line interpolating simulated points with the “curve” function (see Figure 2):

```
> plot(x, y, main="Linear Regression", pch=16)
> curve(2*x+1, col="red", add=TRUE)
```

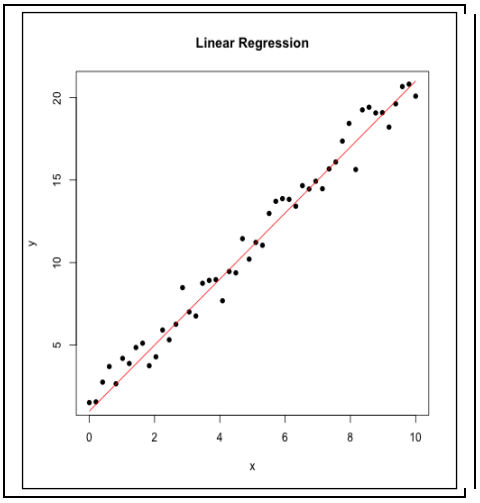


Figure 2

The “ADD=TRUE” keyword included in “curve” means that the straight line is to be added to the existing active plot. More features can be added and figures can be precisely tailored to fit any reasonable requirements. Indeed the R graphics capability is one of the main reasons why this platform has become very popular.

To conclude this very sketchy introduction to R let us illustrate the mechanism through which statistical modelling is performed. Statistical modelling implies the formulation of simple or complex relationships between explanatory and response variables that should explain data variability with a given degree of precision. The process consists of the creation of a model, the fitting of model to data and, finally, the analysis of the results. These different steps can be illustrated by a simple linear model with just one explanatory variable, x, and one response variable, y. The data previously introduced in the plotting example can also be used to illustrate linear regression. Many models can be speculated if the relationship between x and y is not known, but it makes sense to start with simple analytic forms with very few parameters and continue with an increase in analytic complexity and number of parameters until a satisfactory result is met. The definition of a model follows a “regression grammar”, where the symbol “~” is a substitute for regression, while symbols “+” and “-“ means inclusion or subtraction of terms in the regression model. Thus, for instance, the expression,

$$y \sim x$$

is a substitute for a regression where the model is a straight line with intercept, while the expression,

$$y \sim x - 1$$

is a substitute for a regression where the model is a straight line without intercept. The fitting itself is carried out by the function “lm”, in which the model is specified. The results can be saved into an R object of a class called class *lm*. Such object can, subsequently, be “queried” or used in a variety of ways for model assessment, for instance with function “summary”, for a quick report of the fitting:

```
> lm_obj <- lm(y ~ x)
> summary(lm_obj)

Call:
lm(formula = y ~ x)

Residuals:
     Min       1Q   Median       3Q      Max
-1.79056 -0.40417  0.00207  0.55003  1.47165

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.39802    0.22875   6.112 1.69e-07 ***
x            1.96340    0.03942  49.808  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.8209 on 48 degrees of freedom
Multiple R-squared: 0.981,   Adjusted R-squared: 0.9806
F-statistic:  2481 on 1 and 48 DF,  p-value: < 2.2e-16
```

The “summary” output is quite informative. For example we can read that the r-squared goodness of fit (squared correlation coefficient) is 0.981, which indicates a very good fit. Among other things we also learn that the estimated slope of the model is 1.96340, while the intercept is estimated as 1.39802. The estimated straight line can be overlapped on the correct straight line using another R function, “abline” and using the model object as an input (see Figure 3):

```
> abline(lm_obj,col="green")
```

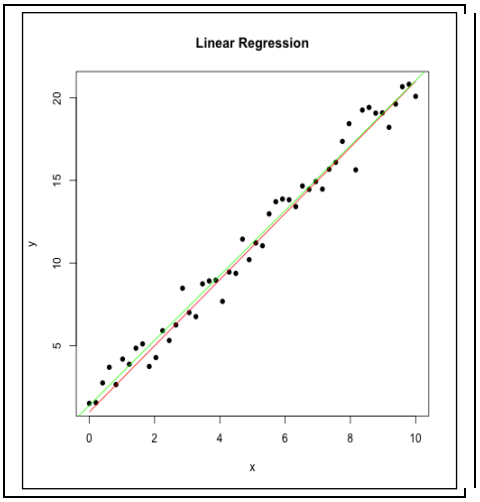


Figure 3

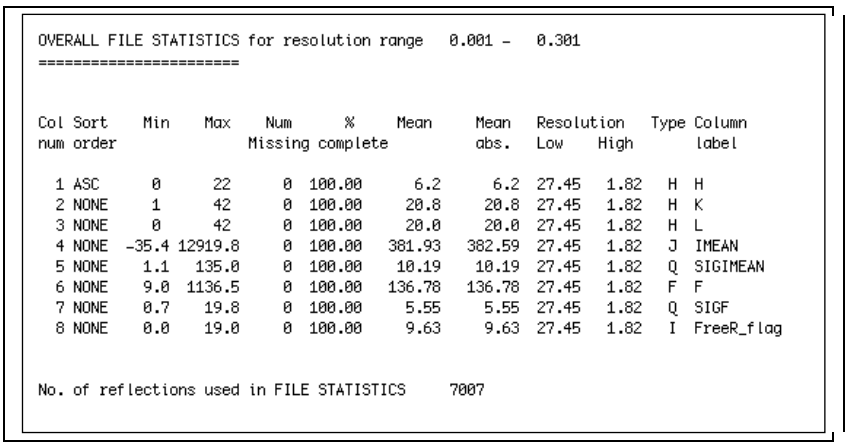
The green line in Figure 3 is the estimated line; it does not coincide with the true red line because of the random errors introduced during the simulation.

All data and typed commands created during an R session can be saved in a special file, which can be reloaded at the beginning of the next R-session. Each session file and the directory where it is stored can be associated with specific projects.

A lot of introductory material on R, including tutorials and books, can be found browsing the platform’s official web site [4].

3. Standard analysis of an mtz reflection file

In Section 2 the content of an mtz file translated in ascii form has been loaded in a data.frame called “mtzdata”. This R object will be now employed to carry out data analysis using built-in R functions. The first feeling for the data contained in the mtz file is normally acquired using the CCP4 program *mtzdump*. One section of the output is displayed in Figure 4.



```
OVERALL FILE STATISTICS for resolution range   0.001 -   0.301
=======================


Col Sort    Min    Max    Num      %     Mean     Mean   Resolution   Type Column
num order              Missing complete         abs.   Low    High       label

  1 ASC      0      22      0  100.00     6.2      6.2  27.45   1.82   H  H
  2 NONE     1      42      0  100.00    20.8     20.8  27.45   1.82   H  K
  3 NONE     0      42      0  100.00    20.0     20.0  27.45   1.82   H  L
  4 NONE  -35.4 12919.8     0  100.00   381.93   382.59  27.45   1.82   J  IMEAN
  5 NONE   1.1   135.0      0  100.00    10.19    10.19  27.45   1.82   Q  SIGIMEAN
  6 NONE   9.0  1136.5      0  100.00   136.78   136.78  27.45   1.82   F  F
  7 NONE   0.7    19.8      0  100.00     5.55     5.55  27.45   1.82   Q  SIGF
  8 NONE   0.0    19.0      0  100.00     9.63     9.63  27.45   1.82   I  FreeR_flag


No. of reflections used in FILE STATISTICS     7007
```

Figure 4

This is easily re-producible in R using the polymorphic function “summary” on the data.frame:

```
> summary(mtzdata)
H                K                L                IMEAN
Min.   : 0.000   Min.   : 1.00    Min.   : 0.00    Min.   :  -35.35
1st Qu.: 2.000   1st Qu.:14.00    1st Qu.:13.00    1st Qu.:   22.32
Median : 5.000   Median :21.00    Median :20.00    Median :   77.37
Mean   : 6.223   Mean   :20.79    Mean   :20.04    Mean   :  381.93
3rd Qu.:10.000   3rd Qu.:28.00    3rd Qu.:27.00    3rd Qu.:  304.54
Max.   :22.000   Max.   :42.00    Max.   :42.00    Max.   :12919.85

SIGIMEAN         F                SIGF             FreeR_flag
Min.   :  1.076  Min.   :   9.042 Min.   : 0.6535  Min.   : 0.000
1st Qu.:  6.276  1st Qu.:  44.042 1st Qu.: 2.6008  1st Qu.: 5.000
Median :  8.130  Median :  87.206  Median : 4.3532  Median :10.000
Mean   : 10.188  Mean   : 136.775  Mean   : 5.5494  Mean   : 9.628
3rd Qu.: 10.844  3rd Qu.: 174.432  3rd Qu.: 8.3001  3rd Qu.:15.000
Max.   :135.019  Max.   :1136.495  Max.   :19.8013  Max.   :19.000
```

Added to the minimum, maximum and mean of any data column, we can read the first quartile, median and third quartile. In general these statistics can be informative on the distribution of each quantity. For instance, the fact that first quartile and median are much closer to the minimum than to the mean for IMEAN tells that the underlying distribution is highly skewed.

What is missing from the above analysis is resolution, because this is not included in data.frame “mtzdata”. A resolution column needs, thus, to be added to this data.frame. To calculate the resolution *d* for each reflection the following well-known formula can be used:

$$1/d^2 = T/B$$

where,

$$T = 1 - \cos^2\alpha - \cos^2\beta - \cos^2\gamma + 2\cos\alpha\cos\beta\cos\gamma$$

and,

$$B = \frac{h^2}{a^2}\sin^2\alpha + \frac{k^2}{b^2}\sin^2\beta + \frac{l^2}{c^2}\sin^2\gamma + 2hk\left(\cos\alpha\cos\beta - \cos\gamma\right)$$
$$+ 2hl\left(\cos\alpha\cos\gamma - \cos\beta\right) + 2kl\left(\cos\beta\cos\gamma - \cos\alpha\right)$$

Of course this formula is not readily available in R; it needs to be defined as an R-function. The function, named “d_hkl”, reads in Miller indices and cell parameters, and returns the corresponding resolution in angstroms. The function can be typed in as follows:

```
d_hkl <- function(h,k,l,a,b,c,aa,bb,cc)
{
 # Given Miller indices and cell parameters, this function returns
 # resolution corresponding to the specific Miller indices.

 aa <- aa*pi/180
 bb <- bb*pi/180
 cc <- cc*pi/180
 top <- 1-(cos(aa))^2-(cos(bb))^2-(cos(cc))^2+2*cos(aa)*cos(bb)*cos(cc)
 b1 <- h^2*(sin(aa))^2/a^2
 b2 <- k^2*(sin(bb))^2/b^2
 b3 <- l^2*(sin(cc))^2/c^2
 b4 <- 2*h*k*(cos(aa)*cos(bb)-cos(cc))/(a*b)
 b5 <- 2*h*l*(cos(aa)*cos(cc)-cos(bb))/(a*c)
 b6 <- 2*k*l*(cos(bb)*cos(cc)-cos(aa))/(b*c)
 d2 <- top/(b1+b2+b3+b4+b5+b6)
 return(sqrt(d2))
}
> class(d_hkl)
[1] "function"
```

The application of this function to all Miller indices of data.frame “mtzdata” shows once more the ability and convenience of using R to carry out a same operation in parallel fashion:

```
> resos <- d_hkl(mtzdata$H,mtzdata$K,mtzdata$L,77.63,77.63,77.63,90,90,90)
> length(resos)
[1] 7007
> summary(resos)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.823   2.018   2.313   2.768   2.930  27.450
```

In the R code just typed “resos” is a new object vector of length 7007 containing resolutions for all Miller indices of the “mtzdata” data.frame, following its same ordering. “resos” stores the parallel application to all triples h, k, l of the function “d_hkl”. This function takes in only one value for h, one value for k and one value for l. Given that the expression “mtzdata$H” and similar expressions for K and L contains all 7007 values for h, k and l, R is clever enough to work out that the operations within function “d_hkl” have to be carried out in vectorised fashion across all triples h, k, l, keeping fixed cell parameters. Finally, “summary” returns resolutions statistics. From the result it is easy to check that minimum and maximum resolutions coincides with that shown in Figure 4.

4. Non-standard analysis of an mtz reflection file

The availability of data in data.frame format enables us to carry out non-standard analysis for the kind of data available in an mtz file. Let us consider, for instance, the issue with FREERFLAG. As it is known, each reflection of an mtz file is tagged with a random integer number in a given range. This has the purpose of dividing data in smaller, random sets to be used, for instance, for cross validation. Independent sampling guarantees that any observed reflection has the same probability to be selected for any given statistical operation. We could, for instance, decide to test the distribution of flags across reflections to make sure that specific integer values are uniformly spread across all resolutions.

Having mtz data available in a data.frame within R allow them to be easily separated using conditional statements. For example, to extract from “mtzdata” all reflections with free R flag equal to 10, we simply type (in the mtz file used here the free R flag has all integer numbers between 0 and 19):

```
> data10 <- mtzdata[mtzdata$FreeR_flag == 10,]
> str(data10)
'data.frame': 336 obs. of  8 variables:
 $ H         : int  0 0 0 0 0 0 0 0 0 0 ...
 $ K         : int  3 4 5 5 6 10 11 12 12 13 ...
 $ L         : int  15 30 5 33 4 28 37 0 28 5 ...
 $ IMEAN     : num  249.8 158.9 92.5 21 22.5 ...
 $ SIGIMEAN  : num  4.86 7.31 3.62 6.89 1.42 ...
 $ F         : num  158 125.9 96.1 43 47.3 ...
 $ SIGF      : num  1.54 2.91 1.89 9.12 1.51 ...
 $ FreeR_flag: int  10 10 10 10 10 10 10 10 10 10 …
```

The new data.frame obtained, which we named “data10”, is composed, of 336 reflections whose “FreeR_flag” column only displays value 10, as it should be. Are quantities included in this data.frame statistically different from the others? For example, if we selected a different data.frame using a free R flag equal to 5, would we notice any major difference with data in “data10”? The answer is, obviously, no. To see this let us, first, create the new data.frame, which we call “data05”:

```
> str(data05)
'data.frame': 333 obs. of  8 variables:
 $ H         : int  0 0 0 0 0 0 0 0 0 0 ...
 $ K         : int  7 7 7 12 12 13 15 16 18 18 ...
 $ L         : int  9 21 35 4 16 15 25 12 12 26 ...
 $ IMEAN     : num  1495 255 1017 1260 157 ...
 $ SIGIMEAN  : num  22.98 7.39 14.76 11.23 16.91 ...
 $ F         : num  387 160 319 355 125 ...
 $ SIGF      : num  2.97 2.32 2.32 1.58 6.81 ...
 $ FreeR_flag: int  5 5 5 5 5 5 5 5 5 5 …
```

Now let us compute summary statistics, for some of the observations, for example for the structure factors amplitudes:

```
> summary(data10$F)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  12.42   40.58   79.25  121.90  133.10  943.90
> summary(data05$F)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.98   43.89   90.23  123.00  156.10  599.10
```

There seem to be no major differences between the two sets of reflections. The maximum is larger for structure factors related to “data10”, but this can normally happen because individual structure factors carry atomic structural information of differing strengths at different resolutions. The important fact to register in this comparison is that all other measures of statistical tendency are pretty close to each other. Histograms for the two distributions can be easily calculated and illustrated using the “hist” function (to find out about the various parameters used in this example, please refer to the manual pages, “help(hist)”):

```
> countsA <- hist(data10$F,breaks=seq(0,1000,length=11),main="Comparison of
frequencies",xlab=expression(list(F[A],F[B])),ylab="Frequency")
> countsB <- hist(data05$F,breaks=seq(0,1000,length=11),lty=2,add=TRUE)
```

```
> legend(600,170,legend=c(expression(F[A]),expression(F[B])),lty=c(1,2))
> countsA <- countsA$counts
> countsA
 [1] 210  69  28  11   6   6   3   1   1   1
> countsB < - countsB$counts
> countsB
 [1] 189  82  32  18   7   5   0   0   0   0
```

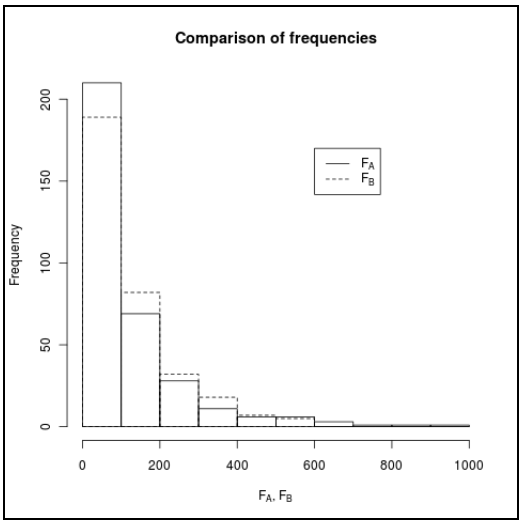


Figure 5

In Figure 5 F_A are structure factors in “data10” and F_B those in “data05”. The two distributions seem very similar. A somewhat conclusive proof that they do not differ significantly can be provided by some statistical test. We can, for example, use a well-established non-parametric test known as *Wilcoxon test* [9]. This is normally used to estimate whether two sets of numbers differ significantly or not. In our case the two sets are the number of structure factors amplitudes within each bin in the histogram. As with many other tests, the Wilcoxon test calculates a so-called *p value* to estimate confidence probabilities. The standard confidence used is the 95% one. This means that if the p value is less than 0.05 there is not enough statistical ground to consider the two sets as having the same distribution, and the null hypothesis of considering them equal is rejected. In R the test can be performed with just one command:

```
> wilcox.test(countsA,countsB)

        Wilcoxon rank sum test with continuity correction

data:  countsA and countsB
W = 57, p-value = 0.6212
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(countsA, countsB) :
```

```
cannot compute exact p-value with ties
```

As the p value is well above 0.05 we can be confident that the two different groups of structure factors follow a same distribution (ignore the warning message; this is simply telling that four of the differences are equal and will be assigned equal ranking values). This implies that the free R flag assignment does not produce any statistically meaningful difference among reflections.

5. Conclusions

The use of statistical packages is as important in crystallographic research as it is in any other scientific field. Data analysis forms the basis for quantitative investigations. It is, therefore, not surprising that statistics should be used appropriately with any scientific finding. A special session was devoted to statistical packages at the European Crystallographic Meeting in 2012 (Bergen, Norway). The first two presentations of that session focussed on the R package and on how it could be used to help out in crystallographic investigations [10, 11]. Other authors have used the package and are still using it in their research. It is foreseeable that use of R will increase in crystallography, given the very solid foundations on which it has been built and the impressive speed of diffusion among scientists and academic workers in general.

In this short article it has been shown that it is possible to use of R for statistical data analysis of crystallographic files, without resorting to the standard tools available within the CCP4 suite. Results already obtained with the CCP4 software can be easily reproduced. This provides a feasible path to control and reinforce calculations. Furthermore new types of analysis can be planned and carried out using the huge arsenal of R built-in functions.

Many crystallographic common operations, though, necessitate of specific algorithms. In general these are not implemented either in the R core on in contributed R software. There exist, accordingly, some limitations in the variety of possible crystallographic investigations within R. Additional software, possibly in the form of R packages, is needed to overcome this limitation. Promising steps in this direction have been taken with the creation of two projects dealing with diffraction images [8] and general crystallographic operations [7]. Use of the new software associated with these projects increases substantially the extent of crystallographic analysis.

Acknowledgments

The author would like to thank Andrey Lebedev for reading the manuscript and for providing useful suggestions to improve it.

References

[1] IBM SPSS Statistics 21.0 - August 2012. http://www-01.ibm.com/software/analytics/spss/

[2] StataCorp. 2011. *Stata Statistical Software: Release 12*. College Station, TX, USA: StataCorp LP. http://www.stata.com/

[3] Minitab 16 Statistical Software (2010). [Computer software]. State College, PA, USA: Minitab, Inc. http://www.minitab.com/en-US/default.aspx

[4] R Core Team, R: A Language and Environment for Statistical Computing (2013) – R Foundation for Statistical Computing - Vienna, Austria. http://www.R-project.org

[5] S-PLUS. TIBCO Software Inc. (2010). Palo Alto, CA, USA. http://www.tibco.com

[6] http://journals.iucr.org

[7] http://code.google.com/p/cry-package

[8] http://code.google.com/p/disp/

[9] G. W. Corder and D. I. Foreman (2009), *Nonparametric Statistics for Non-Statisticians*, Wiley

[10] G. N. Murshudov (2012), Statistics package *R for prototyping in macromolecular crystallography*, Acta Cryst. A**68**, s79

[11] M. W. Baumstark (2012), *R as a tool to check data quality in the context of low-resolution crystallography*, Acta Cryst. A**68**, s80