**Title**
The computational crystallography toolbox: Crystallographic algorithms in a modern software framework



**Journal**
Journal of Applied Crystallography, 35(1)

**Author**
Adams, Paul D.

**Publication Date**
2001-10-10

# The Computational Crystallography Toolbox: crystallographic algorithms in a modern software framework

Ralf W. Grosse-Kunstleve, Nicholas K. Sauter, Nigel W. Moriarty & Paul D. Adams
*Lawrence Berkeley National Laboratory, One Cyclotron Road, Mail Stop 4-230, Berkeley, California 94720, U.S.A.*
E-mail: *PDAdams@lbl.gov*

## Synopsis

We present a library of reusable software components for crystallographic calculations.

## Abstract

The advent of Structural Genomics initiatives has led to a pressing need for high-throughput macromolecular structure determination. To accomplish this, new methods and inevitably new software must be developed to accelerate the process of structure solution. To minimize duplication of effort and to efficiently generate maintainable code, a toolbox of basic crystallographic software components is required. We have undertaken the development of the Computational Crystallography Toolbox (cctbx) for this purpose. In this paper we outline the fundamental requirements for the cctbx and explain the decisions that have lead to its implementation. The cctbx currently contains algorithms for the handling of unit cells, space groups, and atomic scatterers, and is released under an Open Source license to allow unrestricted use and continued development. It will be developed further to become a comprehensive library of crystallographic tools useful to the entire community of software developers.

## 1. Introduction

As a result of the near completion of the Human Genome Project and the creation of several pilot *Structural Genomics* centers (Service, 2000) in the United States, there is a real need for advanced crystallographic software. Macromolecular crystallography is one of the most powerful and general tools available to elucidate the three-dimensional structures of the proteins that correspond to the many genes that have been sequenced. However, the determination of a macromolecular structure can still be a very time consuming, labor-intensive process, even after the experimental diffraction

data have been collected. Given the large number of genes in the human genome alone, it is clear that software must be developed to accelerate this procedure.

Crystallography has often made use of the latest advances in computing *hardware*. This is dictated by the requirements of the algorithms that are typically used in crystal structure determination. Many algorithms are iterative (e.g. structure refinement) or stochastic (e.g. multi solution direct methods like Shake-and-Bake (Weeks & Miller, 1999) and SHELX (Sheldrick & Gould, 1995)) and each iteration or trial is often computationally expensive. In contrast, advances in *software* technology have been slower to propagate through the field, with the dominant programming language in crystallography still being FORTRAN (Backus, 1954).

In order to solve structures the crystallographic community has traditionally used a diverse set of software, often combined in an *ad hoc* fashion. We will explain why this approach is inadequate for the tightly integrated, large-scale projects that are needed to meet the challenges of high-throughput structure determination. In light of this, it is essential that better mechanisms be developed for the organization of computational crystallography software. This requires flexible and reusable software components that many researchers can use to accelerate the development of increasingly high-level software systems.

To better understand the possible approaches to developing new software it is instructive to look at the way large modern software systems are written. They typically feature object-oriented design, databases, graphical user interfaces, distributed computing, and platform independence. *Object-oriented programming* is a fundamental tool in the repertoire of modern computer science. However, until now its potential has not been fully realized in crystallography. In this paper we will discuss the main features of object-oriented programming and how these relate to accelerating the development of a large, maintainable software system for crystallographic applications. Based on this discussion, we describe the major design decisions and important features of the first set of libraries that we have currently released as part of the *Computational Crystallography Toolbox* (cctbx).

# 2. Methods

## *2.1. Fundamental goals*

Our first goal is to design and implement *reusable* software components for macromolecular structure determination that lend themselves to integration into large, modular, layered software systems. The availability of these reusable components will reduce the duplication of effort by different research groups writing new crystallographic applications. The fundamental requirements for a reusable software component include that it (i) performs a well defined task, (ii) communicates input and output parameters exclusively through a well defined *interface*, (iii) supports a mechanism to cleanly recover from unforeseen conditions (errors or exceptions), and (iv) otherwise imposes no unnecessary restrictions on the systems into which it is to be integrated. It must also be possible to have an arbitrary number of simultaneously active, independent instances of the software component. For example, a software component that handles a list of Miller indices and associated data such as experimentally measured diffraction intensities should allow multiple lists to coexist independently. Software components that meet all these requirements are suitable as low-level building blocks for larger systems. Ultimately, the use of reusable software components leads to software systems that even though complex in design can still evolve over time and remain maintainable. If the components are well documented, they also lead to systems that can be simultaneously developed by a diverse group of developers.

Our second goal is to permit a tight integration between reusable software components, written in a compiled language, and a flexible scripting language. Our experience with the Crystallography & NMR System (CNS) (Brunger et al., 1998) has shown that this promotes highly efficient software development. High-level algorithms such as complex refinement protocols or phasing procedures can be developed most rapidly in a scripting language. By contrast, numerically intensive low-level algorithms such as the computation of structure factors or discrete Fourier transforms must be implemented in a compiled language for performance reasons.

## *2.2. Available technology*

This section is a discussion of the abstract concepts of software design in the context of programming languages that are familiar to crystallographers. However, it should be kept in mind that the merits of a particular syntax are secondary to the fundamental concepts.

Computer programming languages have evolved from machine code to be increasingly high-level and abstract. For a relevant subset of programming languages, Fig. 1 illustrates the qualitative correlation between programmer efficiency and run-time performance, which are two essential considerations in the development of any program. As they have developed these languages have provided increased support for rational program organization in order to improve code maintainability and programmer efficiency. In the following section we discuss the implications of this by focusing on four major programming languages, three compiled and one interpreted: FORTRAN, C, C++, and Python. The considerations in this section are the basis for the design decisions that have led to the current implementation of the Computational Crystallography Toolbox.

### 2.2.1. The first compiled language: FORTRAN

Fig. 1 illustrates that Machine code is very efficient at run-time, but requires a long development time. FORTRAN (The IBM Mathematical FORmula TRANslating System) (Backus, 1954) was the first high-level programming language created to reduce this development time and was specifically designed for mathematicians and scientists. Interestingly, a crystallographer (D. Sayre) was a member of the early FORTRAN development team at IBM (Lohr, 2001). It is perhaps therefore no coincidence that FORTRAN has built-in support for two high-level data types that are useful in crystallography: complex numbers and multi-dimensional arrays. To this day the vast majority of crystallographic software packages are written in FORTRAN-77, and new FORTRAN-77 programs are still being developed. Therefore we have carefully considered whether this existing collection of source code can be readily integrated into a comprehensive structure determination system, and have arrived at the conclusion that these programs and low-level libraries written in FORTRAN cannot be easily used for this purpose.

The main problem is that FORTRAN-77 offers only very weak support for writing software components that are reusable in the sense outlined in section 2.1. A major limitation is that FORTRAN-77 provides no portable solution for dynamic memory management. Equally problematic is that FORTRAN-77 does not support user-defined data types (a simple example for a user defined type is a type for Miller indices; an example for a more complex user-defined type is a type for the handling of space group symmetry). However, dynamic memory management and user-defined data types are technical prerequisites for the implementation of software components that support multiple, independent, simultaneously active instances. Consequently, most existing FORTRAN programs support only a limited, hard-coded number of active instances of a given data type, such as a list of Miller indices, symmetry operations, or atomic coordinates (see also appendix section A.1). In addition, the only built-in (as opposed to hand-crafted) mechanism that FORTRAN-77 offers for the handling of run-time errors is the STOP statement. There is no formal way for high-level software components to handle such conditions. The lack of user-defined data types and other mechanisms for structuring a large system leads to implementations that are very hard to maintain as they grow. An anecdotal piece of evidence is that a significant number of the existing software packages in crystallography are written and maintained by single individuals.

FORTRAN programs are notoriously inflexible in accommodating evolving requirements. The most frequently encountered example is a hard-coded limit on the maximum number of atoms or reflections a program can handle. This is a direct result of the lack of portable dynamic memory management. A more complex and serious inflexibility arises from the lack of user-defined types: associated data elements (for example the three elements of a Miller index or the six components of an anisotropic temperature factor) must be passed individually in subroutine calls. In large systems this leads to very long parameter lists. Adding or extending data items therefore involves changing large portions of a program system, since often many subroutines are affected. These changes are both time-consuming and error-prone. In practice, problems like this often make the exploration of new ideas very difficult. Even systems that have been well designed within the constraints of the programming language become difficult to maintain as they evolve. For example, as a result of the advances in experimental technologies, high-resolution macromolecular data sets are more common and thus anisotropic atomic temperature factor refinement is a required tool for structure refinement. Unfortunately, adding this feature to the CNS program (Brunger et al., 1998), which

currently implements only isotropic atomic temperature factor refinement, would require changes to a very large number of subroutine calls throughout the source code. The amount of work involved makes such a change unlikely and illustrates the difficulties that are encountered with any FORTRAN based system as algorithmic requirements change over time.

### 2.2.2. Dynamic memory management and user-defined data types: C

The C programming language supports dynamic memory allocation and user-defined data types. However, there is no formal way of grouping data and algorithms such that the source code clearly reflects the underlying concepts. Memory management in C is notoriously difficult because the programmer can only rely on low-level allocation and deallocation calls. The language has no mechanism that could significantly ease the burden that dynamic memory management puts on the programmer. This is compounded by the lack of a formal mechanism for the handling of errors. Consequently, C code tends to be cluttered with *if* statements for the handling of errors (see e.g. SgInfo (Grosse-Kunstleve, 1995)) and special code for handling dynamic memory management under error conditions. In addition, the lack of a formal mechanism for the handling of errors often leaves the programmer resorting to the use of global variables for reporting error conditions (e.g., SgInfo (Grosse-Kunstleve, 1995); even the standard C library relies on this approach (errno.h)). Software components that use global variables are not suitable for inclusion into modern multi-threaded applications, such as database servers or graphical user interfaces.

### 2.2.3. Object orientation and exception handling: C++

C++ extends the support for user defined types beyond the mere grouping of data (see appendix section A.1). The grouping can also include the associated algorithms. Using this language feature naturally leads the programmer to a source code organization that reflects the underlying concepts. In recent years, *exception handling* has been included in the language to provide comprehensive support for the handling of errors or other unforeseen conditions (see appendix section A.2). Another important feature of ISO C++ (International Standardization Organization et al., 1998) that has been added to the language in recent years is the support for the parameterization of types: C++ *templates*.

The use of templates alleviates some of the difficulties that are associated with statically typed compiled languages such as C and FORTRAN. For example, many FORTRAN and C libraries

(such as BLAS (http://www.netlib.org/blas/) or LAPACK (http://www.netlib.org/lapack/)) explicitly implement the entire set of library functions for two or more different floating-point precisions (e.g. real, double precision, real*16). This large degree of redundancy hampers further development. Other systems use compile-time defines (e.g. FFTW (http://www.fftw.org/)) or require the use of non-portable compiler options to select between different floating-point precisions (e.g. FFTPACK (http://www.netlib.org/fftpack/)). In this case it is impossible to use the library with more than one floating-point precision in the same program. In C++, the floating-point type can be parameterized, and the compiler automatically generates the required type-specific code as needed. There is no need to hand-craft function names to reflect the various floating point precisions and multiple type-specific versions of the code can be used simultaneously without a run-time penalty. The parameterization of types in combination with exception handling also opens an avenue for high-level dynamic memory management that is completely different from the scheme familiar to C programmers. If the C++ features are used to the full extent, dynamic memory management is typically entirely automatic for the vast majority of software components in a system.

An important difference between a modern compiled language such as C++ and older languages is its broad flexibility in terms of run-time performance and programmer efficiency (see Fig. 1). The programmer is permitted to decide how much effort is devoted to optimizing algorithms and data structures for run-time performance, or how the high-level features of C++ are used to reduce the time required for code development. When necessary C++ allows a more labor intensive programming style to be used that leads to machine code that runs as fast as a program written in FORTRAN (Veldhuizen & Gannon, 1998). It is a significant practical benefit that this type of low-level programming and relatively high-level programming is possible within the framework of one language.

It should be noted that FORTRAN-95 removes many shortcomings of the language (Redwine, 1995). For example, FORTRAN-95 adds support for dynamic memory management, user-defined types and limited support for object-oriented design. However, support for exception handling and the parameterization of types is missing. In addition, FORTRAN-90 and FORTRAN-95 have not gained wide acceptance in computational crystallography.

### 2.2.4. An advanced scripting language: Python

C++ meets all the requirements for writing reusable software components that we presented in the previous section. The main disadvantage of C++ is a syntax that requires a significant time investment to learn. As with any compiled language it is also time-consuming to develop platform-specific procedures for building executable components from the source code. There are situations where the performance benefits of a compiled language are not important, for example for very high-level applications such as a complex structure solution and refinement procedure that communicates over a network connection with a graphical user interface. In such cases a scripting language is the most appropriate computational framework. The example of CNS, and many systems outside of crystallography, shows that the flexibility of a platform independent, dynamically typed, interpreted scripting language significantly accelerates the development of very high-level applications.

We have carefully evaluated a variety of scripting languages that could be used in combination with C++. We found Python (http://www.python.org/) to be outstanding because it is a mature language with an object model that is similar to that of C++ and also supports exception handling. Python is interpreted, dynamically typed and generally regarded as a tool for the rapid development of maintainable applications. This is facilitated by a clear syntax that is easy to learn and support for both object-oriented design and modular program architecture. In addition the Boost.Python Library (http://www.boost.org/) is available for conveniently integrating C++ and Python. It is used to directly connect C++ classes and functions to Python without obscuring the C++ interface.

Boost.Python significantly reduces the effort required to convert from a Python to a C++ implementation. This situation arises frequently in the development of scientific algorithms where a scripting language such as Python is the most efficient tool for rapidly exploring a new idea. When the script-based algorithm is mature, performance considerations often make the move to a compiled language necessary. Since both Python and C++ are object-oriented, in most cases a computationally efficient C++ implementation can be quickly derived from an existing Python implementation, and Boost.Python is available to easily re-establish an identical Python interface. In this way the time spent prototyping a software component is not lost, and the advantages of

object-oriented design are used throughout, in the Python layer and the compiled layer, at all stages of development.

### 2.2.5. Software development infrastructure

A modern software development infrastructure goes far beyond the traditional compiler and the operating system. For example, the master copy of the cctbx source code is hosted at *SourceForge* (http://sourceforge.net) and is accessible via the Internet. SourceForge is a free service to *Open Source* (http://www.opensource.org/) developers. The concept behind Open Source is very simple. When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves. Developers improve the code, adapt it to new areas, and correct errors. This can happen at a speed that, compared to the typical pace of conventional software development, seems astonishing.

SourceForge offers easy access to a rich set of tools that foster collaborative software development, including a *Concurrent Versions System* (CVS). This enables a group of developers to work on the same code base simultaneously without requiring extensive effort to merge the updated code. The CVS setup also facilitates multi-platform development because it is very easy to install a *local copy* of the master CVS repository on a variety of platforms. This environment where a group of collaborators develop a system simultaneously on a variety of platforms naturally leads to source code that is both reusable and platform independent.

The use of a system for automated documentation generation further enhances the reusability of the cctbx. The bulk of the documentation is directly embedded as comments in the source code. This ensures that updating the documentation is a natural part of changing the source code. The embedded comments are structured by using a non-intrusive mark-up syntax that has similarities with the HTML syntax (http://www.w3.org/). The free Doxygen (http://www.stack.nl/~dimitri/doxygen/) program is used to extract these structured comments and to convert them into various formats that are easy to navigate. Supplementary documentation, such as an introductory section or a tutorial section, is automatically linked to the reference documentation. A number of examples are also included; these are discussed in the following section.

Another important part of the cctbx is the automatic multi-platform build system (that is based on Python and the traditional *make* facility). This system serves two main purposes: ease of installation, and expediting the development. On the supported platforms, the entire cctbx package can be installed with a single command once the distribution files are unpacked, or the local CVS copies are complete. Depending on the platform, the installation takes between approximately 10 minutes and 30 minutes. Clearly it is impractical to recompile all source code files each time a change is made during development, therefore the build system keeps track of the compilation dependencies and only rebuilds the components that are affected by a change. Typically, the time required for rebuilding the cctbx after a change is only a fraction of the time required for a new installation.

# 3. Results

## 3.1. cctbx overview

This section is an overview of the features of the cctbx. Examples illustrate the intended use of selected algorithms. A summary of the cctbx features is given in appendix section A.3. Detailed documentation for all cctbx features is available online (http://cctbx.sourceforge.net/).

At the point of this writing the cctbx provides three main modules. The *eltbx* (element toolbox) is a collection of tables of various X-ray and neutron scattering factors, element names, atomic numbers, atomic weights, ionic radii, and characteristic X-ray wavelengths. Associated with each table are procedures for accessing the tabulated data, e.g. by using interpolation.

The *uctbx* (unit cell toolbox) provides tools for the description and manipulation of unit cells, and is organized around the `UnitCell` class. An instance of this class is initialized with the six unit cell parameters or a metrical matrix (Boisen & Gibbs, 1990) (also known as metric tensor). The metrical matrix, orthogonalization matrix, fractionalization matrix and the unit cell parameters of the reciprocal cell are pre-computed when the `UnitCell` class is used to construct an object (see appendix section A.1). Repetitive computations involving the UnitCell instances, e.g. the

computation of d-spacings for a list of Miller indices, are therefore highly efficient. (Where possible, the C++ `inline` keyword is used to achieve maximum performance.)

The *sgtbx* (space group toolbox) is the most comprehensive module of the cctbx. A large variety of space-group symbols can be used to derive the corresponding symmetry operations. The reverse process is also supported: given a group of symmetry operations, the sgtbx can be used to compute a space-group symbol. This is facilitated by an algorithm for the determination of the space-group type (see appendix section A.3.3.2). Other algorithms include the characterization of symmetry matrices (A.3.3.4), the handling of basis transformations, the manipulation of Miller indices including the handling of systematic absences and phase restrictions, the manipulation of fractional coordinates including the assignment of Wyckoff positions, the handling of Euclidean and affine normalizers (also known as *Cheshire groups* (Hirshfeld, 1968); see appendix sections A.3.3.2 and A.3.3.3), and the handling of asymmetric units, both in direct space and in reciprocal space.

The sgtbx algorithms are almost entirely general. A technical restriction is that the symmetry operations and transformation matrices are internally represented as integer matrices and base factors. For example, the default base factor used for the translation part of a symmetry operation is 12, and a translation part ($x$,$y$,$z$) is internally represented as three integer numbers (12$x$,12$y$,12$z$). The use of integer matrices greatly simplifies the implementation of the sgtbx algorithms. In addition, integer based algorithms are typically significantly faster than floating-point based algorithms. The restriction to certain fractions is not normally a significant limitation and the advantages clearly outweigh the disadvantages.

## 3.2. Discussion of examples

The examples discussed here can be viewed and executed through a web browser by visiting http://cci.lbl.gov/cctbx/, or by installing the files on a local web server. Each example consists of two files, a HTML file with the user interface and a Python script that is executed by a web server. The source code is available in the *examples* directory of the cctbx. The source code shown in the following figures is derived from the online examples and has been partially rearranged for the purposes of presentation. The source code fragments are also useful as examples for C++

developers. With a few minor modifications (e.g. addition of variable declarations, semi-colons and curly braces) the source code fragments can easily be converted into the corresponding C++ code.

### 3.2.1. Browse alternative space group settings

The *browse_settings* example uses an internal table, with 656 space group settings, that is based on Table 4.3.1 in the International Tables for Crystallography, Volume A (1983). Via the web interface the user specifies a space group symbol. The Python script determines the space group number corresponding to the given symbol, and then lists all tabulated settings for that space group number. Recognized symbols include space group numbers, Hermann-Mauguin symbols, and Schoenflies symbols. For space groups that are tabulated with two origin choices in the International Tables, these symbols can be followed by a colon and the character "1" or "2", indicating origin choice 1 or origin choice 2, respectively. For rhombohedral space groups, the symbols can be followed by a colon and the character "H" or "R", indicating hexagonal axes or rhombohedral axes, respectively. For the monoclinic space groups, both the short (e.g. *P* 2) and the long (e.g. *P* 1 2 1) Hermann-Mauguin symbols can be used. For higher symmetry space groups, only the short Hermann-Mauguin symbol is recognized (the long symbols are inconvenient and not commonly used).

The processing of Hermann-Mauguin symbols is exclusively based on table lookup. No attempt is made to derive symmetry operations from the Hermann-Mauguin symbols directly. While it is natural to derive a Hermann-Mauguin symbol for a given list of symmetry operations, it is problematic to derive the symmetry operations from a Hermann-Mauguin symbol. In particular, there are no established rules for the selection of the location of the origin with respect to the symmetry elements, or the selection of alternative origins. Unconventional Hermann-Mauguin symbols are therefore both difficult to handle algorithmically and prone to misunderstanding. For a more in-depth discussion of this topic see Grosse-Kunstleve (1999). Unconventional space group settings are best encoded as Hall symbols (Hall & Grosse-Kunstleve, 2001), which are also supported by the web interface. Unlike Hermann-Mauguin symbols, Hall symbols can be used to unambiguously represent any arbitrary space group setting.

Using the cctbx, processing the input space group symbol to obtain symmetry operations involves only two Python statements (or alternatively similar C++ statements) that are shown in Fig. 2(a). The next step in the example script is to determine the space-group type from the group of symmetry operations. The corresponding statements are shown in Fig. 2(b). This step is required for the general case where a Hall symbol is given as the input and the space group number is therefore not known from a table lookup. The determination of the space-group type typically only takes a small fraction of a second. Therefore it is not a problem to use the algorithm even if the space group number is available by different means. The advantage is that the script is both general and simple.

The structure of the Python loop for generating the list of alternative settings is shown in Fig. 2(c). The six source code lines shown in Fig. 2 form the functional core of the example script. The entire script is 83 lines long. The 77 lines that are not shown are for processing the input from the web browser, formatting the HTML output, and the handling of errors.

### 3.2.2. Explore symmetry

The *explore_symmetry* example reports a number of space group properties given a space group symbol or symmetry matrices, or a combination of the two. In addition to the input field for a space group symbol there is an input field for symmetry operations in Jones-Faithful notation (e.g. -*z*+1/2,-*x*,*y*+0.5). The example script processes the input space group symbol in the same way as shown in Fig. 2 (if no symbol is given "P1" is implicitly substituted). An example result of a symbol lookup is shown in Fig. 3(a).

The functional core of the loop for processing the additional symmetry operations is shown in Fig. 4. While the additional symmetry operations are processed, a table like the one shown in Fig. 3(b) is generated. After all symmetry matrices are added, the resulting space group is characterized as shown in Fig. 3(c), and the space-group type is determined in the same way as shown previously in Fig. 2(b). Also shown is a parallelepiped containing an asymmetric unit in direct space (Fig. 3(d); see also appendix section A.3.3.7), a table with all symmetry operations of the space group similar to the table shown in Fig. 3(a), a table of Wyckoff positions (Fig. 3(e); see also appendix section

A.3.3.6), and the additional generators of the Euclidean normalizer (Fig. 3(f); see also appendix section A.3.3.3).

### 3.2.3. Change space group setting

It is common that a certain crystal structure is published in the literature in two or more different settings of the same space group. A typical example is that of a rhombohedral space group (e.g. *R* 3) where either a hexagonal basis system or a rhombohedral basis system is used. Other examples are space groups with two origin choices (e.g. *P n n n*), or orthorhombic space groups where the basis vectors are permuted (e.g. *P* 2 2 $2_1$, *P* 2 $2_1$ 2, *P* $2_1$ 2 2). Unusual settings can also arise from group-subgroup or relations (e.g. the monoclinic subgroup of space group *P* 3 1 2 which is generated by the two-fold axis parallel [-1, 1,0]). The *change_setting* example of the cctbx can be used to determine the change-of-basis matrix between two settings of the same space group. Optionally, this change-of-basis matrix is used to transform unit cell parameters and atomic coordinates.

The main inputs to the example are two space group symbols that correspond to alternative settings of the same space-group type. In the first step, the symmetry operations for the two input settings are generated, and the space-group type along with a change-of-basis matrix to the reference setting is determined for each (the source code is equivalent to that in Fig. 2). Let $C_{\text{old}}$ be the change-of-basis matrix that transforms atomic coordinates in the first input setting to coordinates in the reference setting, and $C_{\text{new}}$ the matrix that transforms coordinates in the second input setting to coordinates in the reference setting. The change-of-basis matrix $C_{\text{old-new}}$ that transforms coordinates in the first setting to coordinates in the second settings is then obtained as the product:

$$C_{old-new} = C_{new}^{-1} C_{old}$$

The Python source code for computing $C_{\text{old-new}}$ and reporting the result is shown in Fig. 5(a). The unit cell parameters are transformed using the `ChangeBasis` method of the `UnitCell` class. The source code for transforming atomic coordinates is shown in Fig. 5(b). An example output is shown in Fig. 6.

### 3.2.4. Assign Wyckoff positions

When studying a crystal structure it can be helpful to know the Wyckoff positions of the atoms in the structure. Wyckoff letters can be assigned with the *wyckoff* example script of the cctbx. The input form contains fields for the unit cell parameters, a space group symbol, atomic coordinates and a minimum distance. Atomic sites that have symmetrically equivalent points within the given minimum distance are moved to the exact location of the nearest special position before the Wyckoff letter is assigned. The source code for this step is shown in Fig. 7(a). Fig. 7(b) shows the source code for the actual assignment of the Wyckoff letter. The details of the algorithms are published elsewhere (Grosse-Kunstleve & Adams, 2002).

### 3.2.5. Structure factor calculation

The cctbx contains the entire infrastructure needed for efficiently carrying out a direct-summation structure factor calculation. This is demonstrated by the *web_hklf* example. The online input form contains fields for entering unit cell parameters, a space group symbol, a minimum d-spacing up to which structure factors are computed, a minimum distance between symmetrically equivalent atomic sites (see the *wyckoff* example), and for entering atomic coordinates.

An initial step is the assignment of Wyckoff positions to the input coordinates. This step is equivalent to the previous example. That is, atoms are moved the exact location of the nearest special position before the structure factor calculation is carried out. This provides the user with information about the input coordinates, and robustly circumvents the problems due to limited input precision (e.g. 0.33 instead of 1/3).

The standard features of Python are used to process the list of atomic sites. The information that is stored for each site is shown in Fig. 8(a). Fig. 8(b) shows how the list of Miller indices is generated. Fig. 8(c) shows the actual structure factor calculation.

## 4. Conclusion

The cctbx is a toolbox of fundamental algorithms for computational crystallography that are designed for integration into highly automated software systems, but are also suitable for smaller systems, including educational software. The choice of modern programming languages that

support a modular system design through object-orientation and exception handling ensures ease of use and a high degree of flexibility.

Some of the basic functionality of the cctbx is also present in pre-existing programs. However, these implementations are not reusable in the sense outlined in section 2.1. As a result many basic algorithms are often re-written many times over, usually in an *ad hoc* fashion and with severe limitations. For example, many programs use a static table of symmetry operations. This makes it difficult to work with other settings, such as primitive settings of centered space groups that are the most suitable for certain calculations. The cctbx offers a much more dynamic and flexible approach. Object-oriented design and the use of exception handling lead to uniform and reusable implementations of the basic software components that can be easily integrated into larger systems.

The combination of Python and C++ gives access to both the flexibility of a dynamically typed, interpreted language and the performance benefits of a statically typed, compiled language. The Boost.Python library is used to implement concise bindings between Python and C++. Unlike other systems with a comparable functionality, Boost.Python is implemented in pure C++ and therefore does not introduce the overhead of a third syntax.

The cctbx has all the important features that are typically associated with the term *reusable software*, including unrestricted availability. The cctbx source code is freely available under an Open Source license for both non-profit and commercial use at http://cctbx.sourceforge.net/. We are convinced that this is the most appropriate license type for a scientific software library. This allows all the developers in the extended crystallographic community to integrate the cctbx into other systems. Since the cctbx is hosted by an independent organization that promotes concurrent development (SourceForge), it is also easy for developers to make contributions to the cctbx for the benefit of the community. Instead of repeatedly reinventing the wheel, developers can reuse the modular, extensively tested and mature cctbx components and thus focus their efforts on the development of high-level algorithms. We hope that the cctbx is the first major step towards building a reference library for crystallographic computations which will eventually become a computational equivalent of the International Tables for Crystallography.

## Acknowledgements

We would like to thank Y. LePage and E. Kroumova for providing electronic copies of the Wyckoff tables as published in Hahn (1983). K. Cowtan contributed an initial version of the algorithms for the handling of contiguous reciprocal-space asymmetric units. Vincent Favre-Nicolin contributed initial versions of the Henke tables, the Sasaki tables, and the table of neutron bound scattering lengths and cross-sections of the element toolbox. Our work was funded in part by the U.S. Department of Energy under Contract No. DE-AC03-76SF00098 and NIH/NIGMS under grant number 1P50GM62412-010003.

## Appendix

### *A.1. The concept of classes and objects*

In the terminology of object-oriented programming, a class is to an object what a blueprint is to a building. A building is constructed based on the blueprint. Similarly, an object is *constructed* based on the class (see Figs. 9(a) and 9(b)). To facilitate this, each class contains definitions of data items and associated algorithms: *methods* in Python terminology, *member functions* in C++ terminology.

The closest FORTRAN-77 comes to the concept of a class is a common block. However, a common block only contains data. Associated functions and procedures are connected to each other only in a very loose and unobvious way: by using the same common block. Another fundamental difference is that a common block is static. That is, only one copy of the data in the common block can exist in the same program. In contrast, objects are constructed dynamically, and there can be arbitrarily many objects based on the same class (limited only by the amount of memory available at runtime). This allows for high-level expressions like the one shown in Fig. 9(c). Note that there is almost no technical or notational overhead in the expression (apart from the dot and the parentheses). This level of abstraction would be impossible to achieve with lower-level programming languages such as C and FORTRAN-77.

## A.2. Exception handling

Another principle concept of modern programming languages is that of *exception handling*. Both Python and C++ support exception handling, and the cctbx uses this for handling and reporting errors.

With exception handling, the handling of errors is cleanly separated from the actual algorithms. Consider a simple expression like `b = a.inverse() * x` to compute the inverse of a matrix a and multiply the result with a vector `x`. If the matrix `a` is not invertible, an exception is raised by the matrix inversion algorithm. This exception, and hence the error, can be detected and processed anywhere in the program. Without exception handling, the expression would have to be broken up into a sequence of atomic operations and `if` statements to test for errors. Potentially, the error status has to be propagated through a multi-level hierarchy of function calls. The resulting code is heavily cluttered with `if` statements (see e.g. SgInfo (Grosse-Kunstleve, 1995)).

Many FORTRAN programs use an approach that avoids elaborate error testing: a `STOP` statement is used to report an error condition and terminate the program. This makes it very difficult to integrate such programs into larger systems that typically require run-time status to be carefully tracked. Exception handling can be viewed as a generalization of the `STOP` statement that is compatible with modular programming. If an exception is raised but not caught, the result is identical to using a FORTRAN `STOP`. However, high-level components of a larger system have the full ability to catch and analyze the exceptions and continue in an orderly manner.

## A.3 Summary of important cctbx features

### A.3.1. Element Toolbox: eltbx

- 9-term coefficients for the analytical approximation to the scattering factor for all elements and selected ions (International Tables for Crystallography, Volume C, 1992, p. 500-502).
- 11-term coefficients for the analytical approximation to the scattering factor for all elements and selected ions (Waasmaier & Kirfel, 1995).
- Henke tables (Henke et al., 1993) for elements with Z=1-92. Each table contains about 500 points on a uniform logarithmic mesh from 10 to 30,000 eV with points added 0.1 eV above

and below absorption edges. The atomic scattering factors are based upon experimental measurements of the atomic photoabsorption cross section. The absorption measurements provide values for the imaginary part of the atomic scattering factor. The real part is calculated from the absorption measurements using the Kramers-Kronig integral relations.

- Sasaki tables (Sasaki, 1989) for elements with Z=4-83 and Z=92. These tables are valid in the energy range 4-124 keV. They have a fine step size close to the absorption edges (K,L1,L2,L3). The tables are therefore suitable for use in connection with anomalous diffraction experiments.
- Neutron bound scattering lengths & cross-sections (Neutron News, Vol. 3, No. 3, 1992, pp. 29-37).
- Characteristic wavelengths of commonly used X-ray tube target materials: Cr, Fe, Cu, Mo, Ag.
- Ionic radii (ICSD User's Manual, 1986).
- Atomic weights (CRC Handbook of Chemistry & Physics, 70th edition, 1989-1990).

### A.3.2. Unit Cell Toolbox: uctbx

- Computation of the metrical matrix (Boisen & Gibbs, 1990).
- Computation of the unit cell volume (see e.g. Giacovazzo , 1992).
- Orthogonalization and fractionalization of coordinates using the Protein Data Bank (PDB) convention. Let $\boldsymbol{F} = \{\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}\}$ be the fractional basis, and $\boldsymbol{C} = \{\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}\}$ be the cartesian (orthogonal) basis. $\boldsymbol{C}$ is defined in terms of $\boldsymbol{F}$ by the following relations:

  $\boldsymbol{i} \parallel \boldsymbol{a}$, $\boldsymbol{j}$ is in the $(\boldsymbol{a},\boldsymbol{b})$ plane, $\boldsymbol{k} = \boldsymbol{i} \times \boldsymbol{j}$

  The lengths of all vectors $\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}$ are 1.
- Transformation (change of basis) of unit cell parameters.

  Let $\boldsymbol{G}_{old}$ be the metrical matrix of the old basis system, and $\boldsymbol{R}$ the rotation part of a transformation matrix that transforms the coordinates in the old basis system to coordinates in the new basis system. The metrical matrix for the transformed unit cell is

  $$\boldsymbol{G}_{new} = \boldsymbol{R}^{-t}\boldsymbol{G}_{old}\boldsymbol{R}^{-1},$$

  where $\boldsymbol{R}^{-1}$ is the inverse of $\boldsymbol{R}$, and $\boldsymbol{R}^{-t}$ is the transposed inverse of $\boldsymbol{R}$.
- Computation of various d-spacing measures given a Miller index (see e.g. Giacovazzo, 1992).
- Computation of the maximum Miller indices for a given minimum d-spacing.

This computation is based on evaluating the lengths of the unit vectors in reciprocal space and involves the metrical matrix $\boldsymbol{G}^*$ of the reciprocal unit cell. The maximum Miller indices $\boldsymbol{H_{max}}$ are determined as:

$$H_{\max} = \begin{bmatrix} f((0,1,0),(0,0,1)) \\ f((0,0,1),(1,0,0)) \\ f((1,0,0),(0,1,0)) \end{bmatrix}$$

The function $f$ is defined as:

$$f(u,v) = \mathrm{int}\left( \frac{G^*u \times G^*v}{\sqrt{(G^*u \times G^*v)G^*(G^*u \times G^*v)}} \right)$$

The function int($x$) converts a decimal number $x$ into the next smaller integer number.

## A.3.3. Space Group Toolbox: sgtbx

### A.3.3.1. SpaceGroup

The `SpaceGroup` class is the central class of the sgtbx. Objects of this class are typically initialized with a Hall symbol (Hall & Grosse-Kunstleve, 2001). The symbol is parsed to obtain symmetry operations (represented as matrices) which are then added to the `SpaceGroup` object. It is also possible to add symmetry matrices directly. A Hall symbol does not need to be supplied if only matrices are known.

The internal structure of an `SpaceGroup` object and the optimized algorithm for carrying out the group multiplication are described in detail by Grosse-Kunstleve (1999). The `SpaceGroup` class supports about 50 methods. The principle methods include:

- Input of symmetry operations and group multiplication.
- Test for chirality. A space group is chiral if all its symmetry operations have a positive rotation-part type (1, 2, 3, 4, 6). If there are symmetry operations with negative rotation-part types (-1, -2=m, -3, -4, -6) the space group is not chiral. There are exactly 65 chiral space groups.
- Determination of a change-of-basis matrix that transforms the given setting to a primitive setting. For the conventional centring types (*P*, *A*, *B*, *C*, *I*, *R*, *H*, *F*), tabulated matrices are used

for convenience. For the general case, the matrix is determined with the algorithm of Grosse-Kunstleve (1999).

- Application of a change-of-basis matrix to the symmetry operations.
- Test if given unit cell parameters are compatible with the symmetry operations. A given unit cell is compatible with a given space group representation if the following relation holds for all rotation matrices $\boldsymbol{R}$ of the space group:

  $$\boldsymbol{R}^t\boldsymbol{G}\boldsymbol{R} = \boldsymbol{G}$$

  $\boldsymbol{G}$ is the metrical matrix for the unit cell. This formula tests if the unit cell (represented by $\boldsymbol{G}$) is invariant under the basis transformations corresponding to the symmetry operations.
- Test if a given Miller index fulfills the conditions for a systematically absent reflection.
- Determination of symmetry equivalent Miller indices and related properties (see `SymEquivMillerIndices` below).
- Determination of the point-group type and the Laue-group type.
- Construction of derived groups (Patterson space group, point group, Laue group)

### A.3.3.2 SpaceGroupInfo

The `SpaceGroupInfo` class includes an implementation of the algorithm for the determination of the space-group type (Grosse-Kunstleve, 1999). The input to the algorithm is a group of symmetry operations (given as a `SpaceGroup` object). The result consists of a space group number corresponding to the International Tables for Crystallography (Hahn, 1983) and a change-of-basis matrix that transforms the given symmetry operations to a reference setting. This change-of-basis matrix can be used to transform certain space group properties that are easy to tabulate, but difficult to generate *ab initio*, to the given space group representation. This technique is in principle similar to computing a normal form of a matrix. The normal form is advantageous because certain properties of the matrix can be easily derived from it. Permutation matrices are used to relate these properties back to the original matrix. In the space-group-type algorithm, the reference setting corresponds to the normal form, and the change-of-basis matrix and its inverse correspond to the permutation matrices.

The principle methods of the `SpaceGroupInfo` class are listed below. The class is also used in the determination of Wyckoff letters (A.3.3.6) and for the handling of contiguous reciprocal-space asymmetric units (A.3.3.7).

- Building of space group symbols *given only symmetry matrices as the input*.

  For conventional space group representations, Hermann-Mauguin symbols and Schoenflies symbols are obtained via table lookup. For the general case, the tabulated Hall symbol for the reference setting of the given space-group type is combined with a change-of-basis matrix that is obtained with the algorithm for the determination of the space-group type. To ensure a reproducible Hall symbol for any given space group representation, the given change-of-basis matrix is combined with the operations of the affine normalizer in order to select a "canonical" change-of-basis matrix. Each product of the given change-of-basis matrix and an operation of the affine normalizer is an alternative change-of-basis matrix. The selection of the canonical change-of-basis matrix is based on a set of rules which ensure that the selected matrix is independent of the order in which the alternative matrices are generated.
- Access to generators of the Euclidean normalizer (Koch & Fischer, 1983, Hirshfeld, 1968). The combined use of this method and the `StructureSeminvariant` class (section A.3.3.3) gives a complete description of Euclidean normalizers.
- Test for the 22 (11 pairs) enantiomorphic space groups. A space group $\boldsymbol{G}$ is enantiomorphic if $\boldsymbol{G}$ and $-\boldsymbol{IG}(-\boldsymbol{I})$ have two different space-group types. $\boldsymbol{I}$ is the unit matrix.
- Determination of a *change-of-hand matrix*. This matrix can be used to transform the given symmetry operations to obtain the enantiomorph symmetry operations, and to transform fractional coordinates to the enantiomorph space group.

### A.3.3.3. StructureSeminvariant

Structure-seminvariant vectors and moduli are a description of "permissible" or "allowed" origin shifts. These are important in crystal structure determination methods (e.g. direct methods) or for comparing crystal structures. An introduction to structure-seminvariant vectors and moduli is given in chapter 2.2.3 of the International Tables for Crystallography, Volume B (Shmueli, 2001). The `StructureSeminvariant` class of the sgtbx is an implementation of the algorithms in section 6

of Grosse-Kunstleve (1999). These algorithms are executed when the class is instantiated. The vectors and moduli are accessed through member functions of the class.

Allowed origin-shifts are also a part of the Euclidean normalizer symmetry and listed in the International Tables for Crystallography Volume A, (Hahn, 1983), Table 15.3.2, column "Translations." The other generators listed in the "Additional generators" column of the same table are accessible through the `SpaceGroupInfo` class (see section A.3.3.2).

### A.3.3.4. RotMxInfo & TranslationComponents

The algorithms described by Grosse-Kunstleve (1999) are use to determine the following properties of symmetry operations:

- The rotation-part type (1, 2, 3, 4, 6, -1, -2=m, -3, -4, -6).
- Axis direction (Eigenvector) of the proper rotation matrix corresponding to the rotation matrix of the symmetry operation. (Improper rotation matrices have a determinant of -1. The proper rotation matrix with determinant 1 is obtained by multiplying all elements of the matrix with -1.)
- Sense of rotation (clockwise or counter-clockwise) with respect to the axis direction. (The sense of rotation is only defined for rotation part types 3, 4, 6, -3, -4, -6).
- Determination of the intrinsic (screw or glide) part of the translation part.
- Determination of the location part of the translation part.
- Determination of a fixed point (origin shift) of the Eigenvector of the proper rotation matrix corresponding to the rotation matrix of the symmetry operation.

For example, if applied to the symmetry operation $-y,z+1/2,-x+1/2$, the algorithms produce the results:

- rotation part: 3
- axis direction: [-1, 1, 1]
- sense of rotation: positive (counter-clockwise)
- intrinsic part: (-1/3, 1/3, 1/3) (this is 1/3 [-1, 1, 1], the axis direction; i.e., the symmetry operation is a $3_1$ axis)
- location part: (-1/3, -1/6, -1/6)

- fixed point: (1/6, 1/6, 0)

### A.3.3.5. SymEquivMillerIndices & PhaseRestriction

Objects of the class `SymEquivMillerIndices` are initialized with an input Miller index that is passed to a method of an `SpaceGroup` object. A list of symmetry equivalent Miller indices is computed and stored inside the object. The methods provided by the class `SymEquivMillerIndices` include:

- Test if the reflection with the input Miller index is centric. A reflection with the Miller index ***H*** is *centric* if there is a symmetry operation with rotation part ***R*** such that ***HR*** = ***-H***.
- Multiplicity of the input Miller index. For acentric reflections and in the presence of Friedel symmetry (no anomalous signal), the multiplicity is twice the number of symmetry equivalent Miller indices. For centric reflections or in the absence of Friedel symmetry (i.e. in the presence of an anomalous signal), the multiplicity is equal to the number of symmetry equivalent Miller indices.
- Determination of the factor **ε** for the input Miller index. The factor **ε** counts the number of times a Miller index ***H*** is mapped onto itself by symmetry. This factor is used for statistical averaging (Read, 1986) and in direct methods formulae (Steward & Karle, 1976).
- Determination of a representative symmetry-unique ("asymmetric") Miller index. The selection of the symmetry-unique index is based on twelve contiguous reciprocal space asymmetric units that cover the 230 reference settings. The algorithm for the determination of the space-group type is used to derive a change-of-basis matrix for the transformation of the tabulated asymmetric units. In this way a contiguous asymmetric unit is available for any arbitrary setting.
- Determination of the phase restrictions for the input Miller index. The result is a new object of the class `PhaseRestriction`. Objects of this class provide methods for reporting the pair of restricted phases, and methods for testing if a given phase is compatible with the restrictions.

### A.3.3.6. SymEquivCoordinates

Objects of the class `SymEquivCoordinates` are containers for lists of symmetry equivalent sites in fractional space, for example sites occupied by atoms. For maximum flexibility, the class supports a variety of algorithms:

- A trivial algorithm (no treatment of special positions):

  The symmetry equivalent sites are obtained as the product of all symmetry operations with the fractional coordinates of the input site $\boldsymbol{X}$. The number of symmetry equivalent sites in the resulting list is always equal to the order of the space group. Special positions are not treated in a special way.

- A simplistic algorithm (with treatment of special positions):

  The symmetry operations are applied to the fractional coordinates $\boldsymbol{X}$. The unit cell parameters are used to compute the distances between the symmetry equivalent sites. If the distance between symmetry equivalent sites is shorter than a given a small tolerance, $\boldsymbol{X}$ is on a special position and duplicates are removed from the list. As explained in detail by Grosse-Kunstleve & Adams (2002), due to rounding errors that are inevitably associated with floating-point arithmetic, this simple algorithm is not numerically stable. As a safeguard it is asserted that the number of symmetry equivalent points in the list is a factor of the space group multiplicity. To ensure numerical stability, it is also possible to define an exclusion radius. An exception is raised if a symmetrically equivalent point is within this radius around the original site, but not within the given tolerance. This approach does not silently lead to incorrect results, but manual intervention is required if a problem is detected.

- An algorithm based on the site-symmetry group:

  The site-symmetry group is determined with the numerically robust algorithm of Grosse-Kunstleve & Adams (2002). If the input site is close to a special position, it is moved to the exact location of the nearest special position by applying a *special position operator* that is defined as the average of the symmetry operations of the site-symmetry group. A list of *unique operations* is obtained as the non-redundant list of products of the symmetry operations of the space group and the special position operator. The list of symmetry

equivalent sites is then obtained by multiplying the coordinates of the exact location of the nearest special position with the unique operations. This algorithm is slower than the simplistic algorithm outlined above, but not susceptible to rounding errors and therefore ideal for highly automated applications where the need for human intervention is prohibitive.

- An algorithm based on a table of Wyckoff positions:

  This algorithm is an alternative to the algorithm that is based on the site-symmetry group. Conceptually the algorithms are similar. However, the special position operator is obtained from a *table of representative special position operators* given a Wyckoff letter (Grosse-Kunstleve & Adams, 2002). If the Wyckoff letter is known in advance, this algorithm is both robust and fast.

### A.3.3.7. Brick

A *Brick* in the cctbx is a parallelepiped chosen to minimize the memory that has to be allocated for storing part of a map (e.g., an electron density map) covering an asymmetric unit. Asymmetric units of high-symmetry space groups are complicated shapes, not parallelepipeds. Therefore a brick will in general contain more than exactly one asymmetric unit. However, we are free to choose an asymmetric unit (which is not necessarily contiguous), and find the smallest convenient parallelepiped that will contain this volume.

Bricks for 530 conventional settings and an additional 223 primitive settings of centred space groups were computed with sginfo2 (unpublished work). Currently, the algorithm for computing the bricks is not available in the cctbx, and the bricks are therefore tabulated. However, given the large number of tabulated bricks this should hardly ever be noticeable.

For some applications it is necessary to map out exactly one asymmetric unit. A method for refining a brick is to allocate a map that contains flags indicating whether or not a certain point inside the brick is in the asymmetric unit or is redundant. It is straightforward to generate such a map of flags by looping over the symmetry operations for each grid point. One point is marked as being in the asymmetric unit, and all symmetrically equivalent points are marked as being outside.

# References


Backus, J.W. (1954). Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN.

Boisen, M.B. Jr & Gibbs, G.V. (1990). Mathematical Crystallography, Reviews in Mineralogy, Vol. 15 (revised edition). Washington, D.C.: Mineralogical Society of America.

Brunger, A.T., Adams, P.D., Clore, G.M., DeLano, W.L., Gros, P., Grosse-Kunstleve, R.W., Jiang, J.-S., Kuszewski, J., Nilges, M., Pannu, N.S., Read, R.J., Rice, L.M., Simonson, T. & Warren, G.L. (1998). Acta Cryst. D54, 905-921.

Giacovazzo C. (Ed.) (1992). Fundamentals of Crystallography, IUCr Texts on Crystallography 2, Oxford Science Pub.

Fischer, W. & Koch, E. (1983). International Tables for Crystallography, Vol. A, ch. 15. Dordrecht: Kluwer.

Grosse-Kunstleve, R.W. (1995). SgInfo - A comprehensive Collection of ANSI C Routines for the Handling of Space Group Symmetry. http://www.kristall.ethz.ch/LFK/software/sginfo/

Grosse-Kunstleve, R.W. (1999). Acta Cryst. A55, 383-395.

Grosse-Kunstleve, R.W. & Adams, P.D. (2002). Acta Cryst. A, submitted.

Hahn, T. (1983). International Tables for Crystallography, Vol. A, Dordrecht: Kluwer.

Hall, S.R. & Grosse-Kunstleve, R.W. (2001). International Tables for Crystallography, Vol. B, ch. A 1.4.2., p. 107 and pp. 112-119. Dordrecht: Kluwer.

Henke, B.L., Gullikson, E.M. & Davis, J.C. (1993). Atomic Data and Nuclear Data Tables Vol. 54 No. 2.

Hirshfeld, F.L. (1968). Acta Cryst. A24, 301-311.

International Standardization Organization (ISO), International Electrotechnical Commission (IEC), American National Standards Institute (ANSI), and Information Technology Industry Council (ITI) (1998). International Standard ISO/IEC 14882, 1st ed., Information Technology Industry Council, 1250 Eye Street NW, Washington, DC 20005 (also available at http://webstore.ansi.org/).

Lohr, S. (2001). The New York Times, Wednesday, June 13, 2001.

Read, R.J. (1986). Acta Cryst. A42, 140-149.

Redwine, C. (1995). Upgrading to Fortran 90, Springer Verlag.

Sasaki, S. (1989). Numerical Tables of Anomalous Scattering Factors Calculated by the Cromer and Liberman Method, KEK Report, 88-14, 1-136.

Service, R.F. (2000). Science 289, 2254-2255.

Sheldrick, G.M. & Gould, R.O. (1995). Acta Cryst. B51, 423-431.

Shmueli, U. (2001). International Tables for Crystallography, Vol. B, Dordrecht: Kluwer.

Steward, J.M. & Karle, J. (1976). Acta Cryst. A32, 1005-1007.

Suh, I.-H., Kim, K.-J., Choo, G.-H., Lee, J.-H., Choh, S.-H., Kim, M.-J. (1993). Acta Cryst. A49, 369-371.

Veldhuizen, T.L. & Gannon D. (1998). SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, October 21-23, 1998.

Waasmaier, D. & Kirfel, A. (1995), Acta Cryst. A51, 416-431.

Weeks, C.M. & Miller, R. (1999). J. Appl. Cryst. 32, 120-124.

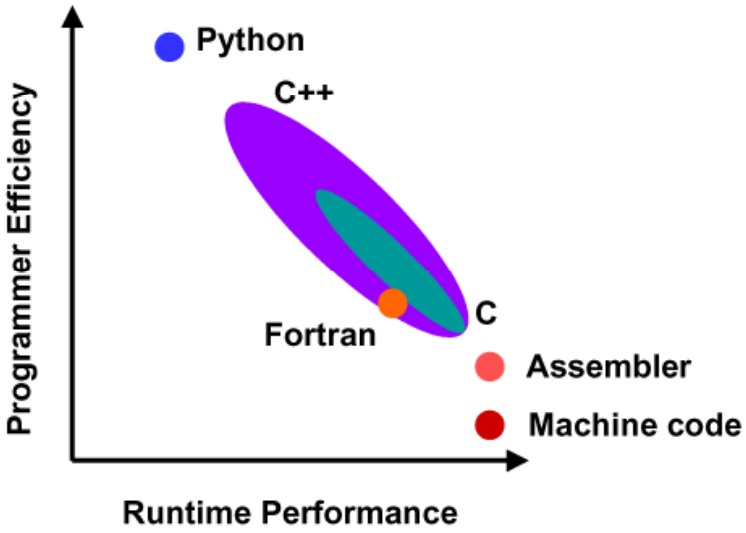



**Figure 1**: Correlation (qualitatively) between programmer efficiency and run-time performance for a selection of programming languages.

**(A high-resolution image will be provided when the paper is accepted for publication.)**

(a)

```
Symbols_Inp = sgtbx.SpaceGroupSymbols(inp.sgsymbol, inp.convention)
SgOps = sgtbx.SpaceGroup(Symbols_Inp.Hall())
```

(b)

```
SgInfo = SgOps.Info()
SgNumber = SgInfo.SgNumber()
```

(c)

```
for SgSymbols in sgtbx.SpaceGroupSymbolIterator():
  if (SgSymbols.SgNumber() == SgNumber):
    # print the HTML formatted table row
```

**Figure 2**: Functional core of the *browse_settings* example Python script.
(a) `inp.sgsymbol` and `inp.convention` are two strings as specified by the user in the input form. `Symbols_Inp` is the result of the table lookup and contains references to the tabulated Hermann-Mauguin symbol, Schoenflies symbol, space group number and Hall symbol. The latter is interpreted by the second statement to obtain a group of symmetry operations.
(b) Determination of the space group number from the group of symmetry operations.
(c) Structure of the Python loop for generating the list of alternative settings.

(a)

```
Input space group symbol: P 41
Convention: Default

Result of symbol lookup:
  Space group number: 76
  Schoenflies symbol: C4^2
  Hermann-Mauguin symbol: P 41
  Hall symbol: P 4w
```

(b)

```
Addition of symmetry operations:
```

| Matrix | Rotation-part type | Axis direction | Screw/glide component | Origin shift |
|---|---|---|---|---|
| -y,-x,-z+1/4 | 2 | [-1,1,0] | (0,0,0) | (0,0,1/8) |
| y,-x,z+3/4 | 4^-1 | [0,0,1] | (0,0,3/4) | (0,0,0) |

(c)

```
Number of lattice translations: 1
Space group is acentric.
Space group is chiral.
Space group is enantiomorphic.
Number of representative symmetry operations: 8
Total number of symmetry operations: 8

Symmetry operations match:
  Space group number: 91
  Schoenflies symbol: D4^3
  Hermann-Mauguin symbol: P 41 2 2
  Hall symbol: P 4w 2c
```

**Figure 3**: Partial output of the *explore_symmetry* example.

(a) Result of the symbol lookup (see section 3.2.1).

(b) Table of additional symmetry operations with characterization of the rotation and translation parts (see appendix section A.3.3.4).

(c) Characterization of a space group (see appendix sections A.3.3.1 and A.3.3.2).

(d)

```
Parallelepiped containing an asymmetric unit:
  0<=x<=1/2; 0<=y<=1/2; -1/8<=z<=3/8
```

(e)

```
List of Wyckoff positions:
```

| Wyckoff letter | Multiplicity | Site symmetry point group type | Representative special position operator |
|---|---|---|---|
| d | 8 | 1 | x,y,z |
| c | 4 | 2 | 1/2*x+1/2*y,1/2*x+1/2*y,3/8 |
| b | 4 | 2 | 1/2,y,0 |
| a | 4 | 2 | 0,y,0 |

(f)

```
Additional generators of Euclidean normalizer:
  Number of structure-seminvariant vectors and moduli: 1
    Vector    Modulus
    (0, 0, 1) 0
  Inversion through a centre at: 1/4,0,0
  Further generators:
```

| Matrix | Rotation-part type | Axis direction | Screw/glide component | Origin shift |
|---|---|---|---|---|
| y,x,-z | 2 | [1,1,0] | (0,0,0) | (0,0,0) |

**Figure 3** (*cont.*): Partial output of the *explore_symmetry* example.

(d) Parallelepiped containing an asymmetric unit (see appendix section A.3.3.7).

(e) Table of Wyckoff positions (see appendix section A.3.3.6).

(f) Additional generators of the Euclidean normalizer of space group $I4_1$ (see appendix section A.3.3.3).

```
for s in inp.symxyz:
  M = sgtbx.RTMx(s) # parse the Jones-Faithful notation
  SgOps.expandSMx(M) # add matrix to the group of symmetry operations
```

**Figure 4**: Functional core of the loop for processing the additional symmetry operations in the *explore_symmetry* example Python script.

(a)

```
M = SgInfo_new.CBOp().InvM() * SgInfo_old.CBOp().M()
CBOp = sgtbx.ChOfBasisOp(M)
print "Change-of-basis matrix:", CBOp.M()
print "               Inverse:", CBOp.InvM()
```

(b)

```
if (inp.coor_type == "Fractional"):
  c = CBOp(coordinates)
else:
  c = UnitCell_old.fractionalize(coordinates)
  c = CBOp(c)
  c = UnitCell_new.orthogonalize(c)
```

**Figure 5**: Functional core of the *change_setting* example Python script.

(a) Computation of $C_{\text{old-new}}$ = `CBOp`.

(b) Application of the change-of-basis operator to fractional or Cartesian coordinates.

```
Old unit cell parameters: 18.497 13.677 12.607 90 90 90

Old space group: (63) A m m a
New space group: (63) C m c m

Change-of-basis matrix: y,z,x
               Inverse: z,x,y

New unit cell parameters:  13.677 12.607 18.497 90 90 90

Fractional coordinates:

T1 0.1126 0.0369 0.1664
T2 0.1128 0.7712 0.9394
T3 0.7733 0.9042 0.0521
```

**Figure 6**: Partial example output of the *change_setting* Python script.

(a)

```
SnapParameters = sgtbx.SpecialPositionSnapParameters(UnitCell, SgOps,
                                                     1, MinMateDistance)
SP = sgtbx.SpecialPosition(SnapParameters, coordinates, 0, 1)
print SP.SnapPosition() # Exact location of the nearest special position
print SP.getPointGroupType() # Site-symmetry point-group type
```

(b)

```
WyckoffTable = sgtbx.WyckoffTable(SgInfo)
WyckoffMapping = WyckoffTable.getWyckoffMapping(SP)
WyckoffPosition = WyckoffMapping.WP()
print WyckoffPosition.M() # Multiplicity
print WyckoffPosition.Letter() # Wyckoff letter
```

**Figure 7**: Functional core of the *wyckoff* example Python script.

(a) Determination of the site-symmetry group. Atomic sites that have symmetrically equivalent points within the given minimum distance (`MinMateDistance`) are moved to the exact location of the nearest special position (`SP.SnapPosition()`). The point-group type of the site-symmetry group is also reported.

(b) Assignment of the Wyckoff letter. This involves the determination of a Wyckoff mapping. The mapping consists of an index into a Wyckoff table (which corresponds directly to a Wyckoff letter) and a symmetry operation. The symmetry operation is applied to the input coordinates to obtain coordinates that are compatible with the particular tabulated special position operator. This is explained in more detail by Grosse-Kunstleve & Adams (2001). See also appendix appendix section A.3.3.6.

(a)

```
Site.Sf = CAASF_WK1995(ScatteringFactorLabel)
```

The analytical approximation to the scattering factor (Waasmaier & Kirfel, 1995). `ScatteringFactorLabel` is a string as read from the web form.

```
Site.Coordinates = UnitCell.fractionalize(site.Coordinates)
```

The fractional coordinates. Shown here is how Cartesian coordinates are converted to fractional coordinates. If the input coordinates are already in fractional units, the values are simply assigned.

```
Site.Occ
```

The occupancy factor. The value is read directly from the web form.

```
Site.Biso
```

The isotropic temperature factor (B-factor). The value is read directly from the web form.

```
Site.WyckoffMapping
```

The Wyckoff mapping facilitates the efficient computation of symmetrically equivalent atomic sites as needed in the structure factor calculation. The procedure for the determination of the Wyckoff mapping is identical to that shown in Fig. 7(b).

**Figure 8**: Functional core of the *web_hklf* example Python script.

(a) Information that is stored for each atomic site. The standard features of Python are used to process the list of atomic sites entered in the input form.

(b)

```
def BuildMillerIndices(UnitCell, SgInfo, Resolution_d_min):
  MIG = sgtbx.MillerIndexGenerator(UnitCell, SgInfo, Resolution_d_min)
  MillerIndices = []
  for H in MIG: MillerIndices.append(H)
  return MillerIndices
```

(c)

```
def ComputeStructureFactors(Sites, MillerIndices):
  FcalcDict = {}
  for H in MillerIndices: FcalcDict[H] = 0j
  for Site in Sites:
    SEC = sgtbx.SymEquivCoordinates(Site.WyckoffMapping,
                                    Site.Coordinates)
    for H in MillerIndices:
      stol2 = UnitCell.Q(H) / 4.
      f0 = Site.Sf.stol2(stol2)
      f = f0 * math.exp(-Site.Biso * stol2) * Site.Occ
      FcalcDict[H] += f * SEC.StructureFactor(H)
  return FcalcDict
```

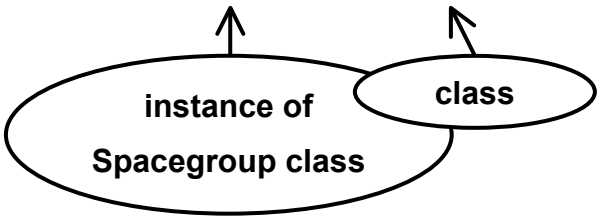
**Figure 8** (*cont.*): Functional core of the *web_hklf* example Python script.

(b) Generation of Miller indices given unit cell parameters, a group of symmetry operations, and a high-resolution limit.

(c) Direct-summation structure factor calculation given a list of sites containing information as shown in (a), and a set of Miller indices as generated in (b).

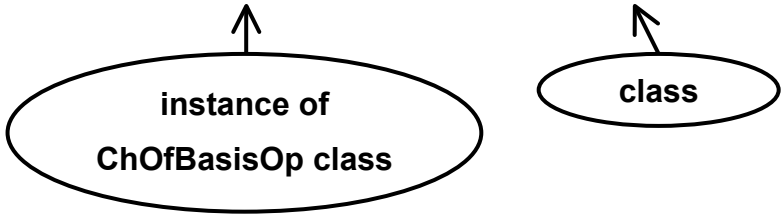
(a)

```
Symmetry = SpaceGroup("P 41")
```




(b)

```
ChangeOfBasisOperator = ChOfBasisOp("-x,-y,-z")
```




(c)

```
TransformedSymmetry = Symmetry.ChangeBasis(ChangeOfBasisOperator)
```

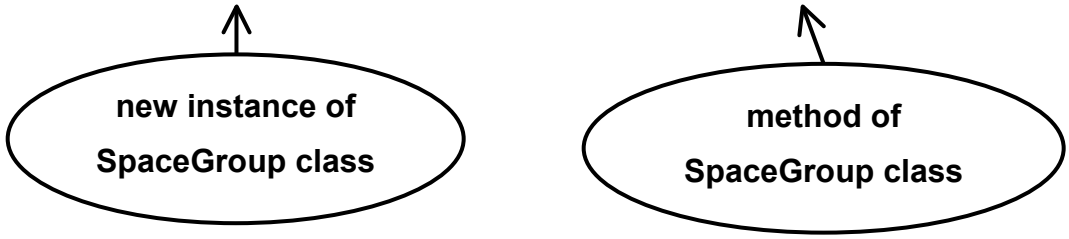



**Figure 9**: Example (using Python syntax) for expressions in an object-oriented programming language. (a) The `Symmetry` object is an *instance* of a class for the handling of space group symmetry (`Spacegroup`). (b) `ChangeOfBasisOperator` is another object that is an instance of a class for the handling of change-of-basis matrices (`ChOfBasisOp`). (c) `ChangeBasis` is a method of the space group class that applies the given change-of-basis matrix to transform the symmetry operations stored in the `Symmetry` object. The result is returned as a new, dynamically allocated instance of the space group class: the object `TransformedSymmetry`. Both `Symmetry` and `TransformedSymmetry` exist simultaneously after the expression is evaluated and can be independently manipulated.