Acta Crystallographica Section D
**Biological Crystallography**
ISSN 0907-4449

# The new *CCP*4 Coordinate Library as a toolkit for the design of coordinate-related applications in protein crystallography

**E. B. Krissinel,[a]* M. D. Winn,[b] C. C. Ballard,[b] A. W. Ashton,[b] P. Patel,[b] E. A. Potterton,[c] S. J. McNicholas,[c] K. D. Cowtan[c] and P. Emsley[c]**

[a]European Bioinformatics Institute, Genome Campus, Hinxton, Cambridge CB10 1SD, England, [b]Daresbury Laboratory, Warrington WA4 4AD, England, and [c]Structural Biology Laboratory, University of York, York YO10 5YW, England

Correspondence e-mail: keb@ebi.ac.uk

Received 8 January 2004
Accepted 25 October 2004

The new *CCP*4 Coordinate Library is a development aiming to provide a common layer of coordinate-related functionality to the existing applications in the *CCP*4 suite, as well as a variety of tools that can simplify the design of new applications where they relate to atomic coordinates. The Library comprises a wide spectrum of useful functions, ranging from parsing coordinate formats and elementary editing operations on the coordinate hierarchy of biomolecules, to high-level functionality such as calculation of secondary structure, interatomic bonds, atomic contacts, symmetry transformations, structure superposition and many others. Most of the functions are available in a C++ object interface; however, a Fortran interface is provided for compatibility with older *CCP*4 applications. The paper describes the general principles of the Library design and the most important functionality. The Library, together with documentation, is available under the LGPL license from the *CCP*4 suite version 5.0 and higher.

## 1. Introduction

A considerable part of program development in protein crystallography deals with coordinate data, commonly represented by PDB (Berman *et al.*, 2000) or mmCIF (Bourne *et al.*, 1997) coordinate files. A typical application of this type includes a PDB/mmCIF parser, an internal structure for keeping all or part of the coordinate data, more or less sophisticated selection, search and edit tools, and routines for writing an output coordinate file. Naturally, different authors employ different approaches to the implementation of these basic elements, which results in a noticeable functional redundancy across the *CCP*4 suite. Because of this, even a minor change in the file format requires a reconsideration of a significant part of the suite. In order to ease the maintenance, the `RWBrook` library (written in Fortran) was added to the suite. `RWBrook` represents essentially a PDB reader/writer with limited coordinate transformation tools and employs a sequential access to coordinate data, which is in line with the design of most of the older *CCP*4 applications.

Although it served its purpose quite well, `RWBrook` has a limited value for the development of modern applications. It does not provide a versatile data structure, which could be used by any application, and as a result it cannot provide a variety of tools for manipulating the data.

The new *CCP*4 Coordinate Library project was initiated in 2000 with the goal of collecting a maximum of coordinate-related functionality, common for the majority of existing and prospective applications, in one package. This should drastically reduce the cost of application design, eliminate the

© 2004 International Union of Crystallography
Printed in Denmark – all rights reserved

functional redundancy in the coordinate-related sections of new developments, and ease the maintenance.

In the present paper, we describe the basic concepts of the new Coordinate Library and outline the functionality it delivers. We stress the benefits, such as reading/writing various coordinate formats, that a developer can automatically enjoy by basing the coordinate-related part of applications on the new Library. A detailed technical description and numerous examples, as well as demonstration applications, are supplied with the Library.

## 2. General concepts

The Library represents a hierarchy of C++ classes, accommodating all data found in the coordinate (PDB or mmCIF) files. A schematic of this hierarchy is shown in Fig. 1. The hierarchy is controlled by a class manager, which provides most of the Library's functionality, exposed in the form of the class's public functions as well as a separate layer of Fortran interface replicating `RWBrook`.

The data, which are read from external files or set up by the application, are shelved by the class manager into respective classes, which roughly correspond to individual PDB records. For example, data from each `ATOM` PDB record are put into a separate instance of the atom class. However, the atom class also includes data from PDB records `HETATM`, `ANISOU`, `SIGATOM` and `SIGUIJ`, which are naturally associated with the atom object. Atoms, forming a residue of an amino-acid chain, are linked to the corresponding residue object, each of which is linked to its chain object. Chain objects are linked to models, a table of which is maintained by the class manager. In this hierarchy, models are identified with NMR models as defined by the PDB format. However, models are made the topmost elements of the coordinate hierarchy, which does not guarantee their chemical identity. This guarantee was sacrificed, firstly because not all PDB files strictly follow the rules, and secondly because placing the models on top adds additional flexibility to the data structure. For example, models may be used for keeping additional molecules, or as a (temporary) storage for generated symmetry mates *etc.*

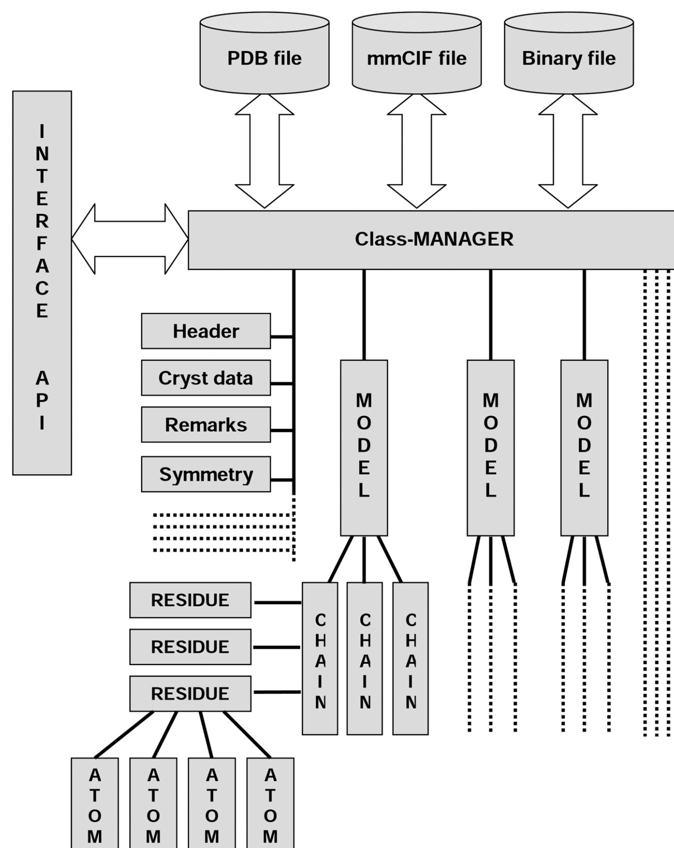


**Figure 1**
Schematic of the coordinate hierarchy realised by the Library classes. Each rectangle denotes one or more C++ classes that keep their internal data and reference tables and provide all the functionality of the corresponding objects. See text for more details.

Each object of the coordinate hierarchy (atom, residue, chain, model) keeps only its own data. For example, secondary structure is a property of the whole molecule, and therefore it is stored in model objects. Data with no relation to a particular coordinate object are stored in additional classes linked directly to the class manager. Examples include the annotation (PDB records `HEADER`, `REMARK`, `AUTHOR` *etc.*), symmetry and crystallographic information (`CRYST`, `SCALE`, `MTRIX` *etc.*), and others.

The coordinate hierarchy is not pre-dimensioned and unfolds dynamically when reading a coordinate file or in the course of editing operations initiated by the application. It is important to stress that all data are kept in RAM, which allows for a random and fast access to all the information. From the application's point of view, the coordinate hierarchy represents a run-time database, accompanied by numerous tools for efficient retrieval, searching and editing of the data. We describe the major functionality, associated with the hierarchy, in the next section.

## 3. Functionality

### 3.1. Parsing coordinate files

The Library provides built-in parsers for reading the PDB and mmCIF coordinate files. Correspondingly, the data from the coordinate hierarchy may be exported in either PDB or mmCIF format. In general, an mmCIF coordinate file contains bits of data that are not found in the PDB format. These data are lost if an mmCIF file is rewritten in PDB format. The hierarchy, in its turn, may contain data that are not found in mmCIF specifications (such as user-defined data, see below). Therefore the hierarchy is equipped with a facility of reading/writing machine-independent binary files, which can keep a snapshot of all data fields. Binary files preserve compatibility with higher versions of the Library. All the Library's parsers automatically recognize the format of input files, although reading of a particular format may be enforced. All library I/O functions allow for automatic reading/writing of compressed files, with the encoding format determined by file extension (such as .gz, .Z).

The Library does not require a file to contain all the information that a particular format may deliver. Missing data are reported by the Library's query functions, and analysis of the completeness of the data for a particular purpose is left up to the application. Missing chemical element names, often found in older PDB files, are automatically calculated from atom names. It is important to notice that the Library provides all the reading/writing functionality so that an application does not need to know the format of the input file. If necessary, the type of input file may be obtained from the Library after the file has been read.

The mmCIF reader/writer, included in the Library, may be used for reading/writing an arbitrary mmCIF file. The reader creates a hierarchy of C++ classes, shown in Fig. 2, which may be queried for the presence of data blocks, categories/loops and data fields. The data are retrieved by data block name, category/loop name, and the name and (in case of a loop) index of the data field. The mmCIF hierarchy may be created by the application and output into a valid mmCIF file.

In addition to the above, the Library includes a generic reader/writer of XML files. The XML reader/writer is based on the same principles as that of the mmCIF, although the XML data structure is much simpler and represents the encapsulation of same-type classes corresponding to the nested XML elements. The Library includes a translator for automatic conversion of mmCIF data hierarchies into those of XML.

### 3.2. Surfing the coordinate hierarchy

The Library provides two methods for accessing the data stored in the coordinate hierarchy: (*a*) directly, using specialized functions of the class manager, and (*b*) through a retrieval of the object (such as atom) that contains the data, with subsequent access of the data as properties of that object. Method (*a*) is preferable for data not directly related to the coordinates (such as annotation, remarks, cell parameters, secondary-structure description *etc.*), while surfing the coordinates is done more conveniently using method (*b*).

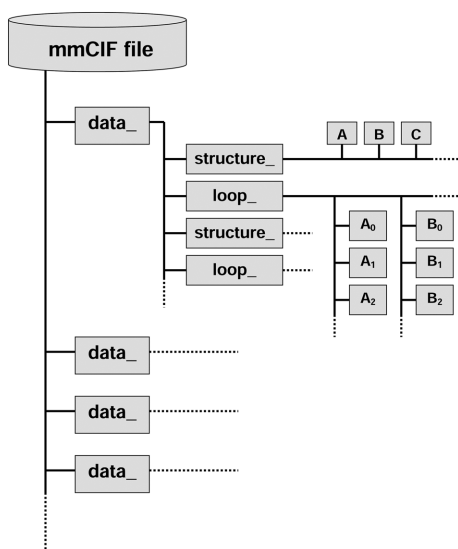


**Figure 2**
Schematic of the mmCIF data hierarchy created by the Library's mmCIF reader. Classes `data_` and `loop_` correspond to the same-named mmCIF objects, `structure_` is a class for representing a non-loop mmCIF category, `A, B, C` *etc.* denote data fields.

**Table 1**
Identification of coordinate objects.

All identification elements correspond to PDB specifications (Berman *et al.*, 2000). CID denotes the corresponding parts of coordinate ID, equation (1).

| Object | CID | Identification |
|---|---|---|
| Model | mdl | Model number 1, 2 ... |
| Chain | chn | Chain ID $A, B, \ldots, Z$ |
| Residue | seq(res).ic | Sequence number (se), insertion code (ic) and (optionally) residue name (res) |
| Atom | atm[el] : al | Atom name (atm), chemical element name (el) and alternative location indicator (al) |

As a basic principle, each coordinate object provides a set of functions for accessing all objects up and down the hierarchy. For example, each residue has functions for getting pointers to all its atoms, as well as pointers to the chain and model that contain the residue, and a pointer to the class manager. While moving up the hierarchy is a straightforward procedure, access to the encapsulated objects requires an addressing. The Library provides the following methods of addressing:

(i) using the PDB specifications as shown in Table 1;

(ii) using the objects's serial numbers (*e.g.* 5th atom in 3rd residue of 2nd chain in 1st model);

(iii) by a coordinate ID (CID).

The Library provides access functions for arbitrary combination of the first two methods (*e.g.* to get a pointer on the 1st atom in the residue 2*B* of the 3rd chain in model 1). Coordinate ID has the following syntax, inspired by that of the UNIX file system,

$$/\mathrm{mdl}/\mathrm{chn}/\mathrm{seq(res).ic}/\mathrm{atm[elm]} \ : \ \mathrm{al}$$

where all terms are explained in Table 1. The Library access functions may take a full or partial CID, according to a convention of defaults and the value of 'current CID'. CID may include a wildcard ('*'), in which case it addresses a set of objects. This feature is used in the Library's selection functions.

### 3.3. Selecting coordinate objects

Quite often, applications operate only on a subset of atoms or other objects. Thus, many tasks require only coordinates of the backbone $C_\alpha$ atoms. The Library offers a rich variety of tools for the selection of coordinate objects. Once selected, the objects can be retrieved as an array of pointers for further manipulations.

Each subset of selected objects is identified by a selection handle, which must be obtained from the class manager before the selection. Any number of selection handles may be issued, any number of selection operations on the same object type (*e.g.* atoms) may be performed using one handle, and each

object may participate in any number of selections simultaneously.

The objects to be selected may be specified by:

(i) a range of PDB specifications (shown in Table 1);

(ii) a wildcarded CID;

(iii) geometrical restrictions (sphere, cylinder, slab or closeness to previously selected objects);

(iv) a range of user-defined data (see below).

Selection sets may be combined using logical `OR`, `AND`, `XOR` and inversion. The selection may be propagated up and down the coordinate hierarchy (*e.g.* to select all atoms of selected chains or select all residues having at least one selected atom).

### 3.4. Editing coordinate hierarchy

Each coordinate object provides functions for reading and modification of its data fields such as atom coordinates. Any object may be removed from the hierarchy or added to it. The object is removed by deleting the corresponding instance of the C++ class (*e.g.* that of atom). That may be done either through the specialized functions of the class manager, which address the to-be-deleted object in the same way as the access functions (*cf.* §3.2), or directly using the class's pointer.

Each coordinate object provides `Add` and `Insert` functions for addition and insertion of the objects it encapsulates. These functions allow an application to build a coordinate hierarchy instead of reading a coordinate file. The hierarchy may be built either from top to bottom (allocate the class manager, add one or more models to it, add chains to each model, residues to each chain, and atoms to each residue), or from bottom to top (allocate atoms, add them to residues, residues to chains *etc.*), or in any combination of these methods.

Each coordinate object, as well as the whole hierarchy, may be copied into another object of the same type. Any part of one coordinate hierarchy may be moved or copied into another hierarchy. A combination of edit tools, provided by the Library, allows for arbitrary manipulations on the coordinate data.

### 3.5. Coordinate transformations

All or any separate part of the coordinate hierarchy may be transformed using a $4 \times 4$ rotation–translation matrix. These basic transformation functions are supplemented with routines for calculating the rotation matrix using either the Euler (in *CCP*4-accepted convention) or polar systems of rotation. Using the inverse routines allows one to obtain the rotation angles from rotation matrices.

The Library provides an exhaustive set of functions for symmetry transformations. The symmetry operations are loaded automatically from the *CCP*4 symmetry library, once the symmetry space group name has been read from a coordinate file or supplied by the application. Alternatively, symmetry operations may be set up by the application directly through the class manager. Either definition of symmetry operations automatically results in the generation of the corresponding rotation–translation matrices, which may be obtained from the class manager or applied directly to the coordinate hierarchy. Thus, a symmetry mate or the whole unit cell may be generated in a single call to the class manager.

Once the class manager obtains a symmetry group name and cell parameters, it automatically calculates the fractional-orthogonal transformation matrices (six orthogonalization codes are available). The actual fractional-orthogonal transformations are performed by a separate set of functions. The Library supports non-crystallographic symmetry (NCS) transformations using NCS matrices found in coordinate files or supplied by the application.

### 3.6. Calculation of contacts

Many applications include calculation of contacts, or finding pairs of atoms that are separated by a certain distance range. The Library provides several functions that allow one to look for contacts between pre-selected sets of atoms. The functions allow for pairwise or multiple contacts (that is, between two or more sets of atoms). Besides the distance range, the functions also control the minimal sequence separation, or the minimal number of residues along the amino-acid sequence between the contacting atoms, with broken chains or gaps of missing residues taken into account. The contact functions employ a bricking algorithm, which is much faster than searching all *versus* all. The brick size is variable, and the number of searched bricks is calculated automatically using the contact distance and brick size. The bricking structure is optimized such that it does not need to be recalculated at repeat searches, and it is available to the application for arbitrary use. For example, the same bricking structure is used for fast selection of objects in the vicinity of those already selected (*cf.* §3.3). The contacts are returned in a special data structure and may be sorted by distance or sequence.

### 3.7. User-defined data and objects

In many cases, it is convenient to use the coordinate hierarchy also for storage of application-specific data that are not found in either PDB or mmCIF coordinate files. For example, the *CCP*4 Molecular Graphics Viewer (Potterton *et al.*, 2002) has colour assignments for each atom, residue and chain. While in most instances the type of additional data is known *a priori*, there may be situations where it becomes known only at run time.

A routine way for accommodating known types of data is through deriving new classes of coordinate objects from those provided by the Library. The newly derived classes must be registered with the Library in order to replace all default factory functions for allocation, disposal, copying, reading/writing *etc.* of the substituted classes. The registration is done by invoking a special macro, which generates a set of factory functions for the new classes, and a single call (for each new class) to the class manager that actually installs the new factory.

Additional data of a type that is unknown at the time of compilation, may be accommodated in the coordinate hierarchy at run time, using the facility of user-defined data (UDD). UDD may be of either real, integer or string type and

must be identified by a unique name. Once the type, name and accommodating object (*e.g.* a residue) of the data become known, they must be registered with the Library using a special function of the class manager. The function returns a unique UDD handle, which is a key for further manipulations with the data. Using this handle, the data may be stored in the registered Library's class, retrieved, modified or deleted. There is no limit in principle on the number and size of different UDD data an application can store in the coordinate hierarchy.

The additional data cannot be output into either PDB or mmCIF coordinate files as it would require extension of PDB keywords or mmCIF dictionaries, which is not encouraged. Therefore, the only way to keep the additional data on a disk is through the Library's binary files, which were mentioned in §3.1. The files are machine-independent and may be used for porting data across different platforms. UDD are stored automatically, while the user-derived classes should follow a few simple steps for their data to be stored in the file. A good example of using the extendability of the Library classes and user-defined data is given by `MMUT` classes developed for the *CCP*4 Molecular Graphics Project (Potterton *et al.*, 2002).

### 3.8. Monomer database

The Library comes with a database of monomers found in PDB files as residues. The database represents a subset of the publicly available EBI-MSD ligand database (*cf.* Boutselakis *et al.*, 2003). The primary purpose of the monomer database is to provide reference data for refinement and molecular replacement applications. Currently it contains 4902 structures and provides the following data for each entry:

(i) three-letter identification code, normally coinciding with residue name in PDB files;

(ii) annotation including full name and synonyms of the structure, its chemical formula, charge and source of coordinate data;

(iii) list of atoms, chemical bonds, angles and torsions between the bonds;

(iv) list of atoms forming a chemical bond with the main chain;

(v) for each atom: atom name, chemical element name, *CCP*4 energy type, chirality, Cartesian coordinates and their standard deviations;

(vi) for each bond: reference to bonded atoms, bond order, length and length's standard deviation;

(vii) for each angle/torsion: reference to three/four atoms forming the element, angle and its standard deviation.

The monomer database allows for direct access to all its entries by the three-letter identification code, or residue name. The database can also perform similarity searches, looking for entries that are structurally close to the query. The query may be represented by a residue class of the coordinate hierarchy, or, alternatively, by a chemical graph. The technique of similarity search is based on an original graph-matching algorithm (Krissinel & Henrick, 2004*a*), the efficiency of which is sufficient for performing a complete similarity search for a 200–300 residue protein chain in less than a second. The Library provides a function for complementing a residue with data from the monomer database, such as addition of H atoms with coordinates calculated to correspond to the conformation of the side chain.

A schematic of the monomer database is given in Fig. 3. All structures are kept in a single file, occupying approximately 46 Mb of disk space. A separate file (12 Mb) contains graphs of the structures, precompiled for the fast similarity search. Both files are indexed in a relatively small (0.5 Mb) file, which is loaded into RAM at the monomer database's initiation stage. The monomer database has its own class manager, which provides all of the database functionality. All database files are platform-independent.

### 3.9. *SWIG* interface

The Library is written in C++ and is intended for use primarily in C++ applications. It also contains APIs to C and Fortran. A special *SWIG* interface has been created for using the Library from scripting languages. This was achieved with the help of an open source program *SWIG* (http://www.swig.org), which generates bindings for a library written in C or C++ to many scripting languages, using the declarations in the library's header files. The program generates wrapper code that the scripting language needs in order to access the C++ objects and methods or C functions. It is also possible to tailor the produced wrapper code to suit one's particular needs. A suitable input file for *SWIG* has been developed to generate bindings for the Coordinate Library to the scripting language Python. It is possible with the generated wrapper code to write programs based on this library straightforwardly in Python.

### 3.10. Other functions

Other useful functions not mentioned before include:

(i) generation of chemical bonds – the bonds are generated using a built-in table of atomic radii and covalent bond lengths, and stored in the coordinate hierarchy as references between the atoms;

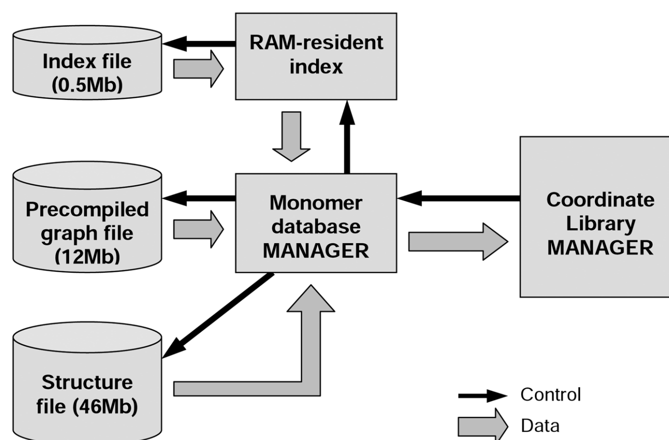


**Figure 3**
Schematic of the monomer database supplied with the Coordinate Library. See details in the text.

(ii) fast superposition of structures with known mapping between their atoms – the employed method is based on singular value decomposition of the correlation matrix (Lesk, 1986) with correction for rotoinversion, as described in detail by Krissinel & Henrick (2004*b*);

(iii) alignment of protein sequences using a dynamical programming algorithm (Smith & Waterman, 1981);

(iv) calculation of secondary structure (Kabsch & Sander, 1983);

(v) implementation of certain elements of graph theory, linear algebra, and multidimensional optimization with global search (Dennis & Schnabel, 1983).

Overall, the Library interface includes more than 1000 functions. At the same time, the number of different classes that an average application would use is relatively low and varies from five to 20. The object design of the Library allows for further expansion and the introduction of more functionality. The Library has been ported to all common UNIX platforms, as well as to Windows NT/98/2000/XP.

## 4. Conclusion

The new *CCP*4 Coordinate Library is an attempt to generalize the coordinate-related functionality, experience of which is accumulated in the *CCP*4 suite. The Library offers a novel approach to the design of coordinate-related applications in protein crystallography, which should considerably reduce the size of new developments, simplify the coding and the resulting programs, and ease the maintenance.

The Library has been used as a basis in a few major developments, particularly the *CCP*4 Molecular Graphics Project (Potterton *et al.*, 2002), molecular building for molecular graphics (Emsley & Cowtan, 2004) and the new EBI-MSD service for protein structure comparison and recognition SSM (Krissinel & Henrick, 2004*b*). The Library is widely employed in various tools serving the EBI-MSD protein database and the EBI-MSD deposition site (Boutselakis *et al.*, 2003). There is experience of using the Library in research-related applications (Nobeli *et al.*, 2003).

From *CCP*4 release 5.0, all coordinate-related Fortran applications are linked with the Library instead of to `RWBrook`. A couple of new C++ applications (`NCONT`, an analogue of older *CCP*4's `CONTACT`, and `PDBCur`, a curation tool for PDB files), which merely represent wrappers over the Library's functions, are included in the *CCP*4 suite. In the longer term, it is planned to phase out the older Fortran utilities, with their functionality being provided by new streamlined C++ applications based on the Library. It is clear from the experience of the above projects that the Library project has reached its goals.

Most of the Library functions are available only through C++ interface and therefore an object-oriented approach to program design should be adopted in order to make the most from the Library. This implies a considerable change in the field, which so far has been Fortran-dominated. We would like to note in this respect that, firstly, the computational efficiency of C/C++ is not lower than that of Fortran, and secondly, using an object-oriented style of programming is widely accepted as preferable nowadays. It delivers a greater flexibility in structuring the code and data, which allows for easier coding of more complicated tasks. Lack of such flexibility is a weak point of many applications in today's *CCP*4 suite.

EBK is grateful for support from the BBSRC Collaborative Computational Project No. 4 in Protein Crystallography (Collaborative Computational Project, Number 4, 1994).

## References

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., Weissig, H., Shindyalov, I. N. & Bourne, P. E. (2000). *Nucleic Acids Res.* **28**, 235–242.

Bourne, P. E., Berman, H. M., McMahon, B., Watenpaugh, K. D., Westbrook, J. & Fitzgerald, P. M. D. (1997). *Methods Enzymol.* **277**, 571–590.

Boutselakis, H., Dimitropoulos, D., Fillon, J., Golovin, A., Henrick, K., Hussain, A., Ionides, J., John, M., Keller, P. A., Krissinel, E., McNeil, P., Naim, A., Newman, R., Oldfield, T., Pineda, J., Rachedi, A., Copeland, J., Sitnov, A., Sobhany, S., Suarez-Uruena, A., Swaminathan, J., Tagari, M., Tate, J., Tromm, S., Velankar, S. & Vranken, W. (2003). *Nucleic Acids. Res.* **31**, 458–462.

Collaborative Computational Project, Number 4 (1994). *Acta Cryst.* D**50**, 760–763.

Dennis, J.E. Jr. & Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, p. 392. Englewood Cliffs, NJ: Prentice-Hall.

Emsley, P. & Cowtan, K. (2004). *Acta Cryst.* D**60**, 2126–2132.

Kabsch, W. & Sander, C. (1983). *Biopolymers*, **22**, 2577–2637.

Krissinel, E. & Henrick, K. (2004*a*). *Softw. Pract. Exp.* **34**, 591–607.

Krissinel, E. & Henrick, K. (2004*b*). *Acta Cryst.* D**60**, 2256–2268.

Lesk, A. M. (1986). *Acta Cryst.* A**42**, 110–113.

Nobeli, I., Ponstingl, H., Krissinel, E. B. & Thornton, J. (2003). *J. Mol. Biol.* **234**, 697–719.

Potterton, E., McNicholas, S., Krissinel, E., Cowtan, K. & Noble, M. (2002). *Acta Cryst.* D**58**, 1955–1957.

Smith, T. F. & Waterman, M. S. (1981). *J. Mol. Biol.* **147**, 195–197.