Legacy Codes
Do They Have Any Value?

David Watkin, Chemical Crystallography Laboratory
OXFORD

Siena, 2005

Legacy code can loosely be defined as any code written at 'some time in the past', and can be divided into two categories – *Obsolescent* and *Obsolete* code.

Obsolescent code includes programs which clearly have a limited future dynamic life span and will include almost all programs in current use in small molecule crystallography. Their development and maintenance is endangered because they are the product either of a single author, or of a small localised group of authors. In addition, most current small molecule programs are written in FORTRAN, a language which itself will either die out, or transform beyond recognition.

Obsolete code includes program which are no longer in active use, and includes forerunners of current programs, programs which are unsupported, and programs addressing issues which have gone away.

Because of the decline in do-it-your-self programming in most crystallography laboratories, legacy code now interests a very limited audience. Outside of the LINUX community, a decreasing number of crystallographers are able to compile old codes. This means that legacy code really only has a value to those small groups of crystallographers still actively developing new programs.

Potential uses for old codes include:

1. Ability to perform computations not in current active codes. Acta Cryst regularly publishes notes or full papers about programs which contain algorithms useful for special purposes, yet which fail to gain common use. If the original author can be contacted, it is sometimes possible to obtain copies of the code. If it is available and can be compiled, the problem is solved.

2. If part of a computation being carried out by a new program is available in an old program, it may be possible to use the old program to validate the new one.

3. As a source of information at a more detailed level than that found in published papers. An example of this is the ability to model electron density distributed throughout some shape other than the usual sphere or ellipsoid. In 1950 King and Lipscomb reported the mathematics for some simple shapes. In 1975 Bennett, Hutcheon and Foxman reported use of a program they had written to carry out these computations, and similar code was again reported in the 1990's. Sadly, the sources of neither the Hutcheon nor the 1990's programs were available by the late 1990's, so that problems which those programmers had at one time already solved had to be solved again. Legacy code would have been helpful.

4. To help in the design of new programs. The more issues that can be foreseen at the design stage, the less re-designing that will be needed later. For example, many structure analysts will be very pleased to be able to input space group information simply be giving the standard symbol, but occasionally users may wish either to use non-standard settings of space groups, or even invent space groups for which there is no symbol. Old programs may help broaden the horizons of new programmers.

5. Data representations. While the external representation for, say, the unit cell parameters is fairly uncontroversial, there is plenty of scope for innovation in the representation of the 'Matrix of Constraint'. A flexible program will probably offer the user a GUI for more routine tasks, and some kind of command–line alternative for more special purposes. Reviewing old programs may suggest alternative representations.

6. Finding solutions to singularities and instabilities. Careful analysis of the mathematics before the coding begins should reveal latent singularities, and it may be evident how to deal with them. Instabilities in the face of unusual data or problems are more difficult to predict, and old codes may highlight otherwise unforeseen issues.

7. Sources of efficient algorithms. While over-optimisation can hinder future development of a program, over-generalisation through lack of a proper understanding of a problem can lead to very inefficient programs. If a procedure is run frequently, a speed increase of a factor of two in what would otherwise be a 60 minute job is useful, and will still be useful even if processors themselves speed up by a factor of ten.

8. Uncovering the users *real* needs. Programs are generally written either for the authors own special needs, or in response to a need expressed by the community. In either case it is important to understand the real need, which may go beyond the immediately perceived problems.

9. Determining the scope of a new program. Writing code costs time/money. Examining similar legacy codes may help in the definition of the scope of a new program, and perhaps help with the prioritisation of the implementation of features. Leaving 'hooks' for features on a wish-list will simplify future development.

10. Assessing the cost of funding the maintenance of legacy code. Procedures such as COCOMO may help in working out the cost of producing new code to replace older codes, but they are probably not very useful for assessing the cost of maintenance and development. In these cases the design quality and documentation of the old code are probably the major influencing factors.

Finding Legacy Code.

Armel Le Bail has set up a Crystallography Source Code Museum in which he has simply archived FORTRAN and C programs, with their documentation if this exists in machine readable form. This is a good first-stop for software archaeology.

Yves Epelboin looks after the SinCris site, which lists alphabetically over 600 software sources. Some of these are to old codes. Unfortunately, since this is only a website of pointers to other sites, it is not uncommon for URLs to be changed without notification.