Experience converting a large Fortran-77 program to C++

Ralf W. Grosse-Kunstleve, Thomas C. Terwilliger, Paul D. Adams
Lawrence Berkeley National Laboratory, One Cyclotron Road, BLDG 64R0121, Berkeley, California, 94720-8118, USA. E-mail: RWGrosse-Kunstleve@lbl.gov

Introduction

RESOLVE is a widely used program for statistical density modification, local pattern matching, automated model-building, automated ligand-fitting, and prime-and-switch minimum bias phasing (Terwilliger 2000, Terwilliger 2003). In recent years it has been developed primarily in the context of PHENIX, which is a rapidly growing software suite for the automated determination of macromolecular structures using X-ray crystallography and other methods (Adams et al., 2002).

PHENIX is a "Python-based Hierarchical ENvironment for Integrated Xtallography". The main layers of the hierarchical organization are a Graphical User Interface (GUI) written in Python (http://python.org), Python scripts implementing applications such as structure solution and refinement, and C++ extensions for numerically intensive algorithms (Abrahams & Grosse-Kunstleve, 2003; Grosse-Kunstleve & Adams, 2003).

The origins of RESOLVE predate the PHENIX project. The original implementation language is Fortran-77. Since Fortran RESOLVE makes heavy use of global data it is not suitable for tight integration as a Python extension. Therefore, Python scripts write RESOLVE input files to disk and call RESOLVE as an external program which writes its outputs to disk. These are read back by the Python scripts to continue the automated processes.

Communicating data through the file system is increasingly problematic as the number of CPUs in a machine or cluster is steadily increasing. For example, if several hundred processes write large density-modified maps or reflection files to a network file system simultaneously, the I/O tends to become a bottleneck. In our development work we found it necessary to mitigate the I/O congestion by writing intermediate files to a local disk (such as /var/tmp).

Another general problem of systems starting external processes is the vulnerability to improper configuration or limitations of the shell environment, for example a very long PATH, or LD_LIBRARY_PATH leading to surprising conflicts. Unfortunately, such problems are reported regularly and they tend to be time-consuming to debug since, by their very nature, they depend heavily on the environment and are difficult to reproduce.

Another set of problems arises from the mixing of Fortran and C++. Using C++ libraries from Fortran is very difficult, although it is not impossible. Fortran RESOLVE uses the latest CCP4 libraries (Collaborative Computational Project, Number 4, 1994) including indirectly MMDB (http://www.ebi.ac.uk/~keb/cldoc/) which is implemented in C++; the required Fortran interface layer is provided by CCP4. A major disadvantage of the C++/Fortran mix is that the build process becomes much more complex than a pure C++ or pure Fortran build.

Mixing Fortran and C++ and starting external processes has a number of other practical drawbacks, for example the external process calls are different under Windows and Unix, and under Windows and Mac OS X it is cumbersome to install any Fortran compiler. Any particular problem by itself is relatively small, but in a system as large as PHENIX the small problems add up to a significant permanent stream of distractions hampering long-term progress. Therefore it was decided to make PHENIX as a whole more uniform - and by implication easier to maintain, develop and distribute - by converting the RESOLVE implementation from Fortran to C++.

Outline of the RESOLVE conversion work

Largely due to the size of the RESOLVE Fortran code, the route taken in the conversion is a very conservative one. The Fortran code is divided into about 700 files with ca. 85000 non-empty lines, of which ca. 72000 are non-comment lines. These sizes practically preclude rewriting RESOLVE as idiomatic C++ code from scratch. Even when producing new code at the very high sustained rate of of 100 lines per day, 50 five-day weeks per year, it would take almost 3 1/2 years to finish the work. After all this time there would be very little new still from a user's perspective. Thus, we wanted to find a more evolutionary approach that produces new results while gradually improving the underlying framework in order to accelerate future developments. Therefore we did not try to turn idiomatic Fortran into idiomatic C++, but instead concentrated our resources on the more limited task of mechanically converting Fortran syntax to C++ syntax that still resembles the original Fortran. The converted code can be evolved over time to increasingly take advantage of the much richer features of the C++ language.

The company Objexx (objexx.com) was hired to help with the mechanical conversion work. The company guided us in tidying the Fortran code and writing a set of automatic tests that together call all subroutines at least once. After this, Objexx converted the sources automatically, followed by some amount of manual editing. We did a few further manual steps to call the CCP4 libraries from C++, but temporarily still through the Fortran interfaces, to keep the manual changes as limited as possible. The test suite was modified to exercise the new C++ version in addition to the original Fortran. A significant but limited amount of time (approximately a couple weeks) was spent on finding and fixing conversion errors. The majority of errors was due to oversights in the manual changes, which underlines the importance of keeping these at a minimum until all tests are in place.

After the syntax conversion we applied an extensive series of semi-automatic source code transformations. The first major goal was to improve the initially quite poor runtime performance, which was expected. However, it was a surprise when we discovered that the performance of the single-precision exp() function in the GNU/Linux system math libraries is extremely poor compared to the Intel Fortran version which we used for comparison (see the following section in this article). The runtime of several tests is dominated by exp() calls. Therefore we substituted custom code for the exp() function, which is not as fast as the Intel version, but much faster than the math library version. The next important step was to replace all small dynamically allocated arrays with automatic arrays. These changes could be applied globally via a set of small, custom Python scripts. The only localized change was to re-write small functions for reading and writing density maps using low-level C I/O facilities (printf(), scanf()). The re-written functions significantly out-perform the original Fortran subroutines.

The second major goal was to transform the sources to be suitable for building a Python extension. For this, all non-constant global variables had to be converted to dynamically allocated variables, to be allocated when RESOLVE is called from Python, and de-allocated after the call is finished. This was achieved by building a C++ struct holding all Fortran COMMON variables and all function-scope SAVE variables. In essence this means the entire RESOLVE program was converted to one large C++ object which can be constructed and destroyed arbitrarily from Python, in the same process. The encapsulation of the entire program as a regular object was completed by using the C++ stream facilities to either read from the standard input "stdin" or a Python string, and to write all output either to the standard output "stdout" or a Python file object. (To be accurate we mention that some CCP4 library routines write output directly to the standard output, i.e. by-pass the redirections, but for our purposes this is currently not important and can be changed later if necessary.)

After finishing the large-scale source transformations we concentrated on the runtime performance of one of the most important parts of RESOLVE, a search for model fragments. Using profiling tools to find the bottlenecks we were able to manually optimize the code to be faster than the original Fortran code (using the Intel Fortran compiler). This is to illustrate that C++ performance can match Fortran performance if reasonable attention is given to the time-critical parts. Of course, similar manipulations to the Fortran code would make the Fortran speed comparable again. Overall the Fortran version is still faster than the

C++ version, up to ca. factor 1.5-2.0, depending on the platform, but this is mainly due to the use of certain approaches to ease the automatic conversion step. The following section in this article presents a systematic review of runtimes.

We like to emphasize that a fast cycle of making changes, re-compiling, and running the tests was crucial for the timely progress of the conversion work. Luckily, re-compiling and running the tests is easily parallelized. Using a new 24-core system with a very recent Linux operating system (Fedora 11), a complete cycle takes less than five minutes. This would have been unthinkable only a few years ago. Without recent hardware and software, the conversion work would have stretched out much longer and some steps may never have been completed due to developer fatigue.

Systematic review of runtimes

It is difficult to present a meaningful overview of the runtimes of a program as large and diverse as RESOLVE. In our experience each platform and each algorithm has a significant potential for puzzling observations. For this article, we have therefore distilled our observations into a self-contained test case that is small but still sufficiently complex to be representative. The test case is a simplified structure factor calculation which only works for a crystal structure in space group P1, with an orthorhombic unit cell, a constant unit form factor, and isotropic displacement parameters ("B-iso" or "U-iso"). This algorithm was chosen because it is typical for crystallographic applications and it uses the exp() library function (see the previous section).

The Fortran version of the simplified structure factor calculation is:

```
subroutine sf(abcss, n_scatt, xyz, b_iso, n_refl, hkl, f_calc)
implicit none
REAL abcss(3)
integer n_scatt
REAL xyz(3, *)
REAL b_iso(*)
integer n_refl
integer hkl(3, *)
REAL f_calc(2, *)
integer i_refl, i_scatt, j, h
REAL phi, cphi, sphi, dss, ldw, dw, a, b
DO i_refl=1,n_refl
  a = 0
  b = 0
  DO i_scatt=1,n_scatt
    phi = 0
    DO j=1,3
      phi = phi + hkl(j,i_refl) * xyz(j,i_scatt)
    enddo
    phi = phi * 2 * 3.1415926535897931
    call cos_wrapper(cphi, phi)
    call cos_wrapper(sphi, phi - 3.1415926535897931*0.5)
    dss = 0
    DO j=1,3
      h = hkl(j,i_refl)
      dss = dss + h*h * abcss(j)
    enddo
    ldw = -0.25 * dss * b_iso(i_scatt)
    call exp_wrapper(dw, ldw)
    a = a + dw * cphi
    b = b + dw * sphi
  enddo
  f_calc(1, i_refl) = a
  f_calc(2, i_refl) = b
enddo
return
end
```

The corresponding C++ version is:

```
void
sf(real1d& abcss,
   int n_scatt, real2d& xyz, real1d& b_iso,
   int n_refl, int2d& hkl, real2d& f_calc)
{
  int i_refl, i_scatt, j, h;
  float phi, cphi, sphi, dss, ldw, dw, a, b;
  DO1(i_refl, n_refl) {
    a = 0;
    b = 0;
    DO1(i_scatt, n_scatt) {
      phi = 0;
      DO1(j, 3) {
        phi = phi + hkl(j,i_refl) * xyz(j,i_scatt);
      }
      phi = phi * 2 * 3.1415926535897931f;
      cos_wrapper(cphi, phi);
      cos_wrapper(sphi, phi - 3.1415926535897931f*0.5f);
      dss = 0;
      DO1(j, 3) {
        h = hkl(j,i_refl);
        dss = dss + h*h * abcss(j);
      }
      ldw = -0.25f * dss * b_iso(i_scatt);
      exp_wrapper(dw, ldw);
      a = a + dw * cphi;
      b = b + dw * sphi;
    }
    f_calc(1, i_refl) = a;
    f_calc(2, i_refl) = b;
  }
}
```

Obviously this is not idiomatic C++ code and it needs a few lines of support code to follow the Fortran syntax this closely:

```
#define DO1(i,n) for(i=1;i<=n;i++)

template <typename T>
struct dim1
{
  std::vector<T> data;
  dim1(int n) : data(n) {}
  T& operator()(int i) { return data[i-1]; }
};

template <typename T>
struct dim2
{
  int n1;
  std::vector<T> data;
  dim2(int n1_, int n2) : n1(n1_), data(n1*n2) {}
  T& operator()(int i, int j) { return data[i-1+(j-1)*n1]; }
};

typedef dim2<int> int2d;
typedef dim1<float> real1d;
typedef dim2<float> real2d;
```

The `DO1` macro is used to emulate the very concise syntax of Fortran loops. The `dim1` and `dim2` struct templates emulate Fortran column-major arrays (as opposed to row-major C-style arrays) with 1-based indices (as opposed to 0-based C-style indices). The three typedefs lead to slightly more concise code.

In passing we note that a C++ struct is equivalent to a C++ class, but without explicit `private` and `public` keywords all struct attributes are public while all class attributes are private.

For reference, the complete source code is embedded in the file `compcomm/newsletter09/sf_times.py` in the open source cctbx project (Grosse-Kunstleve & Adams, 2003). It includes a driver function for generating a random structure and a random list of Miller indices (two versions, Fortran and C++).

Source codes that are so similar should in theory lead to identical machine code, which means identical runtime performance. However, the limited features of Fortran-77, compared to C++, make it much easier for an optimizer to generate efficient code. To find out how this plays out in practice we ran the test codes above using a selection of Fortran and C++ compilers, with eight different variations of the sources above:

```
"s" or "d": single-precision or double-precision floating-point variables
"E" or "e": using the library exp(arg) function or "max(0.0, 1.0 - arg*arg)"
"C" or "c": using the library cos(arg) function or "arg / (abs(arg)+1.0)"
```

The replacements for the exp() and cos() functions are a simple trick to separate the effects of compile-time optimizations from the runtime performance of the math library. (Obviously, when using the replacement functions the resulting structure factor values are meaningless. To ensure that the algorithm is representative of real calculations if the proper exp() and cos() functions are used, `sf_times.py` includes a numerical comparison with the results of the fully-featured cctbx direct-summation structure-factor calculation.)

The complete results are archived in the file `compcomm/newsletter09/time_tables` in the cctbx project. An example table (one of 22) in the file is:

```
current_platform: Linux-2.6.23.15-137.fc8-x86_64-with-fedora-8-Werewolf
current_node: chevy
build_platform: Linux-2.6.23.15-137.fc8-x86_64-with-fedora-8-Werewolf
build_node: chevy
gcc_static: "-static "
compiler: ifort (IFORT) 11.1 20091012
compiler: GNU Fortran (GCC) 4.1.2 20070925 (Red Hat 4.1.2-33)
compiler: n/a
compiler: icpc (ICC) 11.1 20091012
compiler: g++ (GCC) 4.1.2 20070925 (Red Hat 4.1.2-33)
n_scatt * n_refl: 2000 * 20000

  sEC    seC    sEc    sec    dEC    deC    dEc    dec
  1.67   1.31   0.73   0.46   2.26   1.62   1.25   0.84  ifort
 16.76   5.59  12.97   1.76   7.70   6.35   3.15   1.94  gfortran
 -1.00  -1.00  -1.00  -1.00  -1.00  -1.00  -1.00  -1.00  g77
  1.69   1.33   0.78   0.46   2.20   1.65   1.41   0.84  icpc
 16.66   5.54  12.87   1.69   7.59   6.36   2.95   1.86  g++
```

Each row is for a particular compiler: Intel Fortran (ifort), the current GNU Fortran development line (gfortran), an old GNU Fortran development line (g77), Intel C++ (icpc), and GNU C++ (g++), always in this order, but different versions on different platforms. If a certain compiler is not available on a given platform, the corresponding row is filled with `-1.00`. The table heading (`sEC ... dec`) is not included in the `time_tables` file but was added here do indicate each of the eight source code variations. For example, `deC` indicates "double precision", using the `max(...)` function instead of the library `exp(...)` function, and using the proper library `cos(...)` function.

The first striking observation is the very large runtime difference between the ifort and gfortran executables. For the `sEC` case the ifort executable is 16.76/1.67 = 10.0 times faster. Comparing the `seC`, `sEc`, and `sec` columns it is evident that the runtime is dominated by the extremely poor performance of the single-precision exp() and cos() library functions. For the `sec` case, which does not call the math library, the gfortran performance compares significantly better, but the ifort executable is still 3.8 faster.

The next surprise is that the double-precision `dEC` gfortran executable is 2.2 times faster than the single-precision `sEC` gfortran executable. From the other double-precision times in the gfortran column it is apparent that the relatively better performance of the double-precision math library functions is the main reasons for the observation. With both math functions replaced the single-precision version is slightly faster than the double-precision version (1.76 vs. 1.94). The performance of the single-precision ifort executables is consistently better than that of the corresponding double-precision ifort executables (factor 1.4 to 1.8).

The timings above show that, for these particular compiler versions, the icpc executables, considering small uncertainties in the times, are as fast as the ifort executables, and the g++ executables are as fast as the gfortran executables. The situation is still very similar for the latest g++ and gfortran versions. The times with gcc 4.4.2 on the same machine ("chevy", 2.9 GHz Xeon) are:

```
sEC    seC    sEc    sec    dEC    deC    dEc    dec
16.39   5.14  12.73   1.63   7.37   6.14   2.92   1.77  gfortran
16.56   5.37  13.02   1.76   7.64   6.34   3.03   1.90  g++
```

Interestingly, a different picture emerges with an older version of the Intel compilers (machine chevy):

```
1.62   1.29   0.73   0.46   2.07   1.65   1.29   0.84  ifort 9.1 20060323
2.67   2.22   1.94   1.50   3.27   2.66   2.62   1.93  icpc 9.1 20061101
sEC    seC    sEc    sec    dEC    deC    dEc    dec
```

The following table compares the Intel C++ times only:

```
sEC    seC    sEc    sec    dEC    deC    dEc    dec
2.67   2.22   1.94   1.50   3.27   2.66   2.62   1.93  icpc 9.1 20061101
1.88   1.43   0.96   0.63   2.44   1.84   1.41   1.02  icpc 10.1 20080312
1.69   1.33   0.78   0.46   2.20   1.65   1.41   0.84  icpc 11.1 20091012
```

Apparently, the Intel C++ optimizer has been improved significantly in recent years, to be on par with the Fortran optimizer.

As an aside, we want to highlight the following result in the time_tables file (machine chevy):

```
sEC    seC    sEc    sec    dEC    deC    dEc    dec
3.20   2.68   1.84   1.41   3.20   2.69   1.84   1.41  ifort 9.1 32-bit
2.06   1.62   1.29   0.84   2.06   1.62   1.29   0.84  ifort 9.1 64-bit
```

Our motivation for showing these times is to encourage the use of 64-bit systems, even on machines with less than 2 GB system memory. We attribute the speed difference to the larger number of general-purpose CPU registers available to a 64-bit application.

We also like to encourage the reader to inspect the `time_tables` file, which contains significantly more information than is highlighted in this article. The current URL to the directory with the file is:

http://cctbx.svn.sourceforge.net/viewvc/cctbx/trunk/compcomm/newsletter09/

Conversion recipe

The RESOLVE conversion process was a major learning experience. From previous smaller conversion projects (FFTPACK and L-BFGS in the cctbx project) we had learned already that a conversion to idiomatic C++ is very time consuming. We also knew that there is very little to no performance gain when converting 1-based Fortran indices to 0-based C-style indices, as is re-confirmed by the results in the previous section. However, we did not have experience encapsulating a complete large program as a Python extension. In the section outlining the RESOLVE conversion work, we already mentioned that the essential idea is to treat the entire program as one large C++ object. Here we present a more detailed recipe that should be applicable to any Fortran-77 program. We do this by way of a small example:

```
subroutine show_resolution(h, k, l)
 implicit none
 integer h, k, l
 real a, b, c
 logical first
 real ass, bss, css
 real dss
 COMMON /abc/ a, b, c
 SAVE first
 SAVE ass, bss, css
 DATA first /.true./
 if (first) then
   first = .false.
   if (a .le. 0 .or. b .le. 0 .or. c .le. 0) then
     write(5, '(1x,a)') 'invalid unit cell constants.'
     stop
   endif
   ass = 1/(a*a)
   bss = 1/(b*b)
   css = 1/(c*c)
 endif
 dss = h*h*ass + k*k*bss + l*l*css
 if (dss .eq. 0) then
   write(6, '(3(1x,i3),1x,a)')
&    h, k, l, '    infinity'
 else
   write(6, '(3(1x,i3),1x,f12.6)')
&    h, k, l, sqrt(1/dss)
 endif
 return
 end

 PROGRAM conv_recipe
 implicit none
 real a, b, c
 COMMON /abc/ a, b, c
 a = 11.0
 b = 12.0
 c = 13.0
 call show_resolution(0, 0, 0)
 call show_resolution(1, 2, 3)
 end
```

This example includes the three major Fortran features breaking encapsulation: COMMON, SAVE, and STOP. COMMON and SAVE introduce global data, which make the subroutine non-reentrant (see http://en.wikipedia.org/wiki/Reentrant_%28subroutine%29). Worse, in this example the reciprocal space parameters (`ass`, `bss`, `css`) are set only the first time the subroutine is called in the process. Clearly, this would not work well for a Python extension. Subsequent calls of a Python function need to be independent of each other to avoid confusing behavior. This is particularly important in large systems.

While it is difficult to pin-point and remove specific critical globals (the program would need to be re-organized), it is not difficult to "globally remove all globals". The first step is to organize the data:

```
struct show_resolution_save
{
  bool first;
  float ass, bss, css;
  show_resolution_save() : first(true) {}
};

struct common
{
  float a, b, c;
  show_resolution_save show_resolution_sve;
  std::ostream & out_stream;
  common(std::ostream& out_stream_) : out_stream(out_stream_) {}
};
```

The previously global data can now be initialized at an arbitrary point in the process, for example:

```
common cmn(std::cout);
```

Now the `cmn` object, holding all data and a reference to the output stream, can be passed to the converted Fortran PROGRAM which is simply a normal function in C++:

```
void
conv_recipe(common& cmn)
{
  cmn.a = 11.0;
  cmn.b = 12.0;
  cmn.c = 13.0;
  show_resolution(cmn, 0, 0, 0);
  show_resolution(cmn, 1, 2, 3);
}
```

This function assigns values to three `cmn` data members and then passes the `cmn` object on to the show_resolution() function:

```
void
show_resolution(common& cmn, int h, int k, int l)
{
  show_resolution_save& sve = cmn.show_resolution_sve;
  if (sve.first) {
    sve.first = false;
    if (cmn.a <= 0 || cmn.b <= 0 || cmn.c <= 0) {
      throw std::runtime_error(
        "invalid unit cell constants.");
    }
    sve.ass = 1/(cmn.a*cmn.a);
    sve.bss = 1/(cmn.b*cmn.b);
    sve.css = 1/(cmn.c*cmn.c);
  }
  float dss = h*h*sve.ass + k*k*sve.bss + l*l*sve.css;
  std::ostream& cout = cmn.out_stream;
  if (dss == 0) {
    cout << boost::format(" %3d %3d %3d     infinity\n")
      % h % k % l;
  }
  else {
    cout << boost::format(" %3d %3d %3d %12.6f\n")
      % h % k % l % std::sqrt(1/dss);
  }
}
```

This implementation is slightly more verbose than the original Fortran version, but it is also more readable because it is immediately clear which data are related to the `cmn` and `sve` objects. We note that it is possible to avoid the `sve.` if desired by converting `show_resolution()` to a member function of the show_resolution_save struct. In RESOLVE we preferred the approach shown for clarity.

The `cmn` object can be instantiated from a C++ main or a Python function. Currently, RESOLVE makes use of both possibilities. In the case of the Python function, as soon as the call is finished the `cmn` object goes out of scope and the space for all its data is released.

The complete `conv_recipe.cpp` can be found under the URL shown at the end of the previous section. It can be compiled on virtually any Linux system with a development environment since the boost library, which is used for formatting the output of the example, is usually included in the standard development environments.

We think following the conversion recipe shown above it is feasible to automatically convert large sections of entire Fortran programs to re-usable extensions of large integrated systems such as PHENIX. The main obstacle is the Fortran I/O system, which is not trivial to emulate. The seasoned "f2c" system (http://netlib.org/f2c/) proves that it is possible with a reasonable effort, but in practice it may be wisest to simplify the Fortran I/O first and to apply program-specific custom scripts combined with a small amount of manual changes to solve problems with complex I/O operations.

Conclusion

Based on our experience working on CNS (Brunger et al. 1998), RESOLVE, and PHENIX, we are convinced that writing an entire program system in Fortran or even C++ is a very inefficient use of the most valuable resource, human time. Within the PHENIX project, most new developments start out as Python scripts building on previously implemented methods. By converting RESOLVE to a Python extension we have added the RESOLVE functionality to the pool of methods that can easily be re-used. Our systematic review of runtimes suggests that this can be achieved without significantly compromising the performance of time-critical algorithms.

The conservative conversion methods presented above do not produce a designed "clean" system. However, they lead to a system that works today and is simultaneously open to evolutionary development. If there is an evolutionary path there is no need to replace the old. It can be modified instead. In the particular case of RESOLVE, we anticipate gradual changes that over time lead to an increasingly modular organization. Post conversion, the RESOLVE code has already been changed to make use of the extensive C++ libraries developed previously in the context of PHENIX (notably the symmetry and the fast Fourier transform libraries). We also anticipate that some RESOLVE algorithms can be re-factored for use in other contexts, to accelerate the development of PHENIX as a whole.

Acknowledgments

We gratefully acknowledge the financial support of NIH/NIGMS under grant number P01GM063210. Our work was supported in part by the US Department of Energy under Contract No. DE-AC02-05CH11231. We like to thank Stuart Mentzer (Objexx Engineering, Inc.) for invaluable help with the bulk conversion of RESOLVE. Luc Bourhis' code for the C++ stream redirection (part of the cctbx project) was critical for fully encapsulating RESOLVE.

References

Abrahams, D., Grosse-Kunstleve, R.W. (2003). C/C++ Users Journal, 21(7), 29-36.

Adams P.D., Grosse-Kunstleve, R.W., Hung, L.-W., Ioerger, T.R., McCoy, A.J., Moriarty, N.W., Read, R.J., Sacchettini, J.C., Sauter N.K., Terwilliger, T.C. (2002). Acta Cryst. D58, 1948-1954.

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.

Collaborative Computational Project, Number 4 (1994). Acta Cryst. D50, 760-763.

Grosse-Kunstleve, R.W., Adams, P.D. (2003). Newsletter of the IUCr Commission on Crystallographic Computing, 1, 28-38.

Terwilliger, T.C. (2000). Acta Cryst. D56, 965-972.

Terwilliger, T.C. (2003). Acta Cryst. D59, 38-44.