# The TOPAS symbolic computation system

A. A. Coelho[a)]
*72 Cedar Street, Wynnum, 4178 Brisbane, Australia*

J. Evans and I. Evans
*Department of Chemistry, Durham University, Science Laboratories, South Road, Durham DH1 3LE, UK*

A. Kern
*Bruke rAXS, Karlsruhe, Germany*

S. Parsons
*School of Chemistry, The University of Edinburgh, King's Buildings, W. Mains Road, Edinburgh, Scotland, EH9 3JJ*

(Received 20 October 2011; accepted 22 October 2011)

Computer algebra removes much of the drudgery from mathematics; it allows users to formulate models by using the language of mathematics and to have those models evaluated with little effort. This symbolic form of representation is often thought of as being separate to dedicated computational programs such as Rietveld refinement. These dedicated programs are often written in low level languages; they are relatively inflexible in what they do and modifying them to change functionality in a small manner is often a major programming task. This paper describes a symbolic system that is integrated into the dedicated Rietveld refinement program called TOPAS. The symbolic component allows large functional changes to be made at run time and with a relatively small amount of effort. In addition, the system as a whole reduces the programming complexity at the developmental stage. © *2011 International Centre for Diffraction Data*. [DOI: 10.1154/1.3661087]

Key words: symbolic computation, computer algebra, Rietveld refinement

## I. INTRODUCTION

The computer program TOPAS V5 (Bruker AXS, 2011) is primarily a nonlinear least-squares optimization program used in the field of crystallography. It is written in the c++ programming language and at its core is a symbolic computation system. Symbolic computation is a powerful tool as it extends program functionality at run time. Equally important is the simplicity it offers at program development stage by hiding (not eliminating) underlying complexity. Often cited against the use of symbolic computation is slowness in computation. This is largely overcome by calculating only what has been changed at the symbolic equation node level.

Optimization programs generally receive parameter input in the form of numeric values, and these values are modified during the course of optimization. Invariably input flags are used to convey information such as whether to treat a particular parameter value as a constant or whether to modify the parameter value during optimization.

TOPAS is similar, except that information to the optimization routines is passed through symbolic algebraic equations which we will call the TOPAS symbolic system. These equations can describe parameter dependencies or they can define functionality at key points in the model function. The extent to which functionality can be extended is dependent on the design of the model function, written in c++, and the aspects of the model function that are exposed to the symbolic system.

Consider the case of the generalized model function in TOPAS called "fit_obj." It is written in c++ code and the only symbolic variable it manages is the abscissa x; this type of variable will be called a multivalued variable.The symbolic system defines the model itself at run time. For example, a normalized Gaussian peak profile P(x) can be defined by writing in the symbolic form (1).

$$\mathrm{P(x)} = a\,(2\ \mathrm{Sqrt(Ln(2)}/\pi)/f)\ \mathrm{Exp}(-4\mathrm{Ln(2)} \times ((\mathrm{x} - x_\mathrm{o})/f)^2). \quad (1)$$

Here, there are three parameters: the area $a$, the full width at half maximum $f$, and the peak position $x_\mathrm{o}$. These parameters and the Gaussian itself are defined symbolically at run time, and as such they can be redefined. Say, for example, that the full width at half maximum is redefined to vary with respect to x (2).

$$f = f_1\mathrm{x} + f_2\mathrm{x}.^2 \quad (2)$$

The symbolic system recognizes this and forms a dependency tree where $f$ is now dependent on $f_1$ and $f_2$, with $f_1$ and $f_2$ becoming independent parameters. At the calculation stage, the optimization routines substitutes $(f_1\ \mathrm{x} + f_2\ \mathrm{x}^2)$ for $f$ and performs the desired calculation. Without the symbolic system, a new peak type would need to be defined at the programming stage and the program recompiled and debugged.

There are limitations to the TOPAS symbolic system; it does not operate on arrays nor does it allow for the creation of storage arrays. This simplicity, however, allows for speed of calculation whilst still providing a great deal of flexibility.

## II. THE TOPAS SYMBOLIC SYSTEM

Symbolic equations are represented in computer memory as equation trees. Nodes of an equation tree comprise operators (such as the plus operator) and leaves, which can

a)Author to whom correspondence should be addressed. Electronic mail: AlanCoelho@bigpond.com

be variables, constants or pointers to other equations. In addition, parameter dependency trees are formed and independent parameters are identified. For small problems, where there are not many equations such a system will perform the symbolic equation evaluations within an acceptable time period. For larger problems, equations can comprise hundreds of millions of nodes when expanded, where expanded means inserting the symbolic form of an equation into its corresponding equation node. For example, the two equations in Eq. (3a) expands to those shown in Eq. (3b).

$$a = b + c; \quad b = c + d; \tag{3a}$$

$$a = (c + d) + c; \quad b = c + d. \tag{3b}$$

For computational speed, it is desirable not to continually recalculate equation nodes that have unchanged dependents. The TOPAS symbolic system accomplishes this by assigning a storage value and a recalculation flag at the equation node level for nonleaf items. A change in an independent parameter value results in a signal being sent to corresponding equation nodes to inform that a recalculation is necessary. In example of Eq. (3a), a change in the value of $c$ leads to $c$ informing the corresponding nodes in equations $a$ and $b$. Note only the nodes in $a$ and $b$ that need recalculation are signalled. This signalling seems computationally intensive, however, in practice $c$ signals to $b$ only if it has not already been flagged for recalculation and $b$ signals to $a$ only if it has not already been flagged for recalculation. The signalling also considers that during a derivative calculation of a nonindependent parameter equation with respect to an independent parameter, the number of equation nodes signalled lies at the intersection of two sets: one set defined as all parameters that are dependencies of the nonindependent equation and the other set defined as all parameters that have the independent parameter as a dependent. The net effect is that the traversing of dependency trees is a small part of the computation.

A subtle point that is easy to miss in the "Recalculate If Necessary" paradigm is that calculation is essentially performed on simplified equations. In the case of the Gaussian defined in Eq. (1), the optimization routines effectively see the equation shown in Eq. (4) where # denotes equation node storage values. In different problems, each # value could correspond to a lengthy calculation.

$$\mathrm{P(x)} = \# \exp(\#((\mathrm{x} - \#)\#)^2). \tag{4}$$

### A. Equation simplification

Simplifying symbolic equations is the process of manipulation equations to reduce the computational effort. There are many facets to simplifying symbolic equations; TOPAS is not exhaustive in this respect with complex operations such as factorization not performed. It does, however, reduce the equations according to approximately 100 rules and in addition it looks for numeric equivalent forms that reduce the number of equation nodes. For example, the equation $a/2$ can be manipulated to $0.5*a$ which remove the divide; thus a "rule" can be made such that a variable divided by a constant should be changed to a constant times the variable. A more complicated rule is $-\mathrm{Min}(\#a, \#b) = \mathrm{Max}(\#a, \#b)$ where # corresponds to constants. An unsimplified normalized Pseudo-Voigt, comprising 39 nodes, reduces to 18 during calculation as shown in Eq. (5) with two expensive divides being eliminated.

$$\mathrm{P(x)} = \# \exp((\mathrm{x} - \#)^2) + \#/(\# + (x - \#)^2). \tag{5}$$

Some of the simplifications are not rule based and are instead hard coded to increase speed. Speed is of the essence, as simplification occurs at run time during each nonlinear least squares iteration for each independent parameter, and often during derivative calculations. Attention is given to the fact that simplification can take as long as the calculation itself and in these cases the Recalculate if Necessary approach is relied upon. Equations, the size of normalized Pseudo-Voigts, can be simplified on present 2 GHz desktop computers at the rate of over 300 000 per second.

Equations are also analyzed globally and new equations are created from duplicate patterns; trivial patterns are not duplicated as the function call overhead negates the advantage of having fewer nodes. Multivalued independent variables, such as x in the Gaussian example above, are isolated as much as possible. For example, Eq. (6a) could be stored in a binary tree representation as either Eq. (6b) or Eq. (6c).

$$\mathrm{P(x)} = a * \mathrm{x} * b, \tag{6a}$$

$$\mathrm{P(x)} = (a * \mathrm{x}) * b, \tag{6b}$$

$$\mathrm{P(x)} = a * (\mathrm{x} * b). \tag{6c}$$

In Eq. (6b) or Eq. (6c), not having to recalculate $a$ and $b$ would not decrease the number of operations performed in calculating P(x). P(x) is therefore rearranged as shown in Eq. (7).

$$\mathrm{P(x)} = (a * b) * x. \tag{7}$$

In another example, $x/a$ becomes $(1/a)*x$ which during multiple calculations, as a result of the multivalued variable x changing, becomes #*x which removes the expensive divide.

Equation trees are balanced to reduce the number of operations performed. For example, a change in $a$ in the unbalanced equation shown in Eq. (8a) requires three multiplies for recalculation, whereas only two are required for the balanced equation shown in Eq. (8b). A change in $b$ results in three multiplies for Eq. (8a), two for Eq. (8b), etc. Changing each parameter individually, as is the case in derivative calculations, and recalculating, results in one less multiply for Eq. (8b) than Eq. (8a). This gain in efficiency is magnified for larger problems and for the case where multivalued variables are the only variables changing over many calculations.

$$(((a * b) * c) * d), \tag{8a}$$

$$(a * b) * (c * d). \tag{8b}$$

### B. Recalculate if necessary and penalties

In the optimization of penalties (also known as Restraints), where a penalty is a user defined symbolic representation of a



https://doi.org/10.1154/1.3661087 Published online by Cambridge University Press

model function, a particular penalty is calculated many times during derivative calculations with just one independent parameter value changing. The Recalculate If Necessary approach greatly speeds up these calculations, as only equation nodes that need recalculation are calculated. On complex problems, an increase in speed greater than 10 is often achieved.

Simplifying penalties with all parameters treated as constants except for the one for which the numerical derivative is being sought is largely impractical, as the act of simplification often takes longer that the calculation itself; hence the importance of Recalculate If Necessary.

The only way to better the Recalculate If Necessary approach in speed (and not by much) is to code separate simplified c++ functions for each independent parameter. For problems with hundreds of equations and parameters this means writing hundreds of c++ routines and compiling/debugging each time the user defines a new set of penalty functions.

### C. The implementation

A basic understanding of present day compilers and CPUs helps in understanding how speed can be maintained with a symbolic system. For the past 20 to 30 years, CPU speed has been advancing at a rate faster than the speed of regular memory. CPU's therefore have fast cache memory for program data (data cache) and program code (instruction cache). Modern compilers analyze program functions and "inline" them for speed depending on a cost benefit analysis; this saves a function call to regular memory which, depending on memory access, can take a number of clock cycles. Inlining also gives the compiler an opportunity to optimize the inlined code. On the negative side too much inlining increases the amount of code placed in the instruction cache, which may result in a cache miss (which results in the CPU having to wait for instructions to be loaded).

To speed up the calculation of the symbolic equations, Topas performs (inlines) up to three operations in a single function call, for example, in the equation (a+b) there are, without inlining, three function calls; one to the Plus operator and two for the loading of the variables. The inlining performed in Topas reduces this to one function call. This is accomplished by generating 315 small functions using c++ templates. To avoid large code, the number of functions necessary has been minimized by arranging equations in as few patterns as possible. For example, 3*$a$, where $a$ is a variable, can be arranged as $a$*3 and hence only one function is necessary to perform a constant multiplied by a variable rather than two.

With the performing of up to three operations in a single function call only N−1 function calls are necessary for a binary system comprising N leaves. Without inlining the number of function calls would be 2N−1 with N−1 of them being the binary operators such as Plus etc. With the use of the Register Calling Conventions (a compiler switch that exists in most compilers that passes/returns arguments to functions in registers) the speed of the symbolic system is optimized.

Apart from specialized CPUs, the maximum speed in present day desktop CPUs is around one double precision floating point addition or multiplication per clock cycle. Performing a test in c++ (using Microsoft's Visual Studio c++ compiler) on the summation of an array of double precision numbers $\sum_{i=1}^{N} a_i$ results in a speed rating of slightly less than one Gigaflop on a 2 GHz pc. Note that in performing timing tests, it is important to check the assembler code to ensure that the compiler is not optimizing out the code being timed. In performing the test on a two dimensional array $\sum_{i=1}^{N} a_{i,0}a_{i,1}$ the c++ version gave exactly the same time as $\sum_{i=1}^{N} a_i$. This is due to the fact that whilst the coprocessor is performing the addition on $a_{\mathrm{i},0}$ the CPU is fetching $a_{\mathrm{i},1}$ and placing it on the coprocessor stack independently of the addition; thus the speed rating is around two Gigaflops. A further sum of $\sum_{i=1}^{N} a_{i,0}a_{i,1}a_{i,2}a_{i,3}$ does not increase the number of Gigaflops. Performing the same loops using the symbolic system results in eight to ten times less speed; this is expected due to the N−1 function calls performed by the symbolic system. Performing the test on a loop comprising a divide however of $\sum_{i=1}^{N} 1/a_i$ results in the symbolic system being 1.8 times slower. The times for the symbolic system include the loading of the $a_i$ variable with random values before actually performing the symbolic computation.

More demanding trigonometric or power functions result in a closer match between the compiled and the symbolic times. This again is due to the fact that the coprocessor processes the floating point operation independently of the CPU. Note the time taken for many floating point operations is dependent on the arguments provided; for example, 1/2 takes fewer clock cycles than 1.0/1.234; thus the timing tests were performed on floating point numbers that are similar for both the c++ case and the symbolic case.

These results indicate that the symbolic system is not too far off compiled speed for operations more elaborate than addition or multiplication. A comparison of fitting a single Pseudo-Voigt comprising 1200 data points using the Topas symbolic fit_obj and the equivalent function hard coded in Topas gave, for four independent parameters refining over 5000 iterations, 2.34 and 1.48 s for the symbolic and hard coded equivalent, respectively. The optimized hard coded equivalent uses storage arrays such that during a particular refinement iteration the actual Pseudo-Voigt is calculated only twice. To ensure the same number of recalculations of the Pseudo-Voigt, the only nonlinear parameter (the full width at half maximum) was refined with times of 1.38 and 1.01 s for the symbolic and hard coded equivalent, respectively. It is surprising that the symbolic system is so close to the hard coded equivalent; the latter does, however keep, storage arrays and in addition it treats the Gaussian and Lorentzian parts of the Pseudo-Voigt separately. Thus in an attempt to speed up the hard coded equivalent in a general sense overheads are often required.

## III. LARGE RIETVELD REFINEMENT EXAMPLE

Rietveld refinement by Wood *et al*. (2008) comprised seven neutron Time of Flight data sets and one single crystal data set; a total of 29 structural phases, 2707 symbolic



https://doi.org/10.1154/1.3661087 Published online by Cambridge University Press

equations, 22941 equation nodes, and 276 independent parameters. Were the equations expanded they would yield 1,706,390,373 equation nodes. Without the Recalculate If Necessary paradigm there would be tens of millions of unnecessary equation evaluations and refinement time would be orders of magnitude longer. With Recalculate If Necessary, the refinement takes six nonlinear least squares iterations and a total of 6.1 s. The percentage of time spent on the symbolic equation evaluations is about 20% of the total refinement time with the other 80% spent in c++ code calculating structure factors, reflection peaks, etc. In more typical Rietveld refinement problems, the percentage is less than 10% and in problems where there are only penalties then its 100%.

To perform this refinement with the symbolic equations written in c++ would be very time consuming as it would mean coding and debugging hundreds of c++ routines and determining the dependencies for the independent parameters. It is, therefore, not practical to examine the effects of not using the TOPAS symbolic system in this particular example.

## IV. CONCLUSION

Programs comprising model functions written in low level languages, such as c++, that expresses key aspects of functionality in a symbolic form can greatly increase overall program functionality at run time. The logic of the low level code is often simplified and at the same time repetitive program logic is hidden from the user. A balance between the amount of low level and symbolic code is necessary in order to maintain computational speed. In the TOPAS symbolic system, the user can typically experiment with many crystallographic scenarios in a short time period and with a relatively small amount of effort.

Bruker AXS (**2011**). TOPAS, V5.0. (Computer Software), Bruker AXS, Karlsruhe, Germany.

Wood, P. A., Francis, D., Marshall, W. G., Moggach, S. A., Parsons, S., Pidcock, E., and Rohl, A. L. (**2008**). "A study of the high-pressure polymorphs of L-serine using ab initio structures and PIXEL calculations," CrystEngComm **10**, 1154–1166.



https://doi.org/10.1154/1.3661087 Published online by Cambridge University Press