COMPUTATIONAL INFRASTRUCTURE FOR BRIDGING THE GAP BETWEEN PREVIOUS AND FUTURE GENERATIONS OF CRYSTALLOGRAPHERS.

Mustapha Sadki, James Haestier, Amber L. Thompson and David J. Watkin,

Chemical Crystallography Department, University of Oxford.

Crystallographic Computational Infrastructure

One of the requirements for the next generation of small molecule crystallographers is a mathematical programming infrastructure, an easy and efficient means where crystallographers test their own ideas, construct new algorithms and make them available quickly to the whole community.
Having such an environment will allow building large and maintainable models for structure determination and analysis that can be adapted quickly to new situations.
We have made a concerted effort to this end and have started to implement the required infrastructure for computational crystallography including, but not limited to :

- Algebraic modelling language for crystallography
- Automatic differentiation
- Structure factor calculation
- Nonlinear least squares
- Fourier maps and
- Parameter refinement …

Modelling Language for Crystallography (Smx.interpreter)

A built-in algebraic modelling language designed around a familiar crystallographic notation, including an interactive command environment, to help with the formulation of problems. It includes a full-featured programming language; possesses a complete set of looping and conditional statements and allows the crystallographer to formulate algebraic models for structure determination and analyse data in a clear and concise way. The model is used then as a basis to generate a mathematical representation that can be relayed directly to the optimisation solver.

Example: Use of AD with smx.interpreter

```
{
  parameter   p, p2
  p = 0.2; p2=6
  p = f ( p2 ) * exp(p )

  differentiate()

  print "r p : " ,  p
}
```

```
#  Declares 2 parameters
#  Simply assigned values initially
#  Can be overwritten

# Invoke reverse sweep
# and reclamation of memory
# p(value, adjoint) will hold current value
  and adjoint w.r.t. last computed param_t
```

Example with C++

```
{
  smx::ADparam::param_t   p,p2;
  p = 0.2; p2=6;
  p = f ( p2 ) * exp(p );

  smx::ADparamEnv::differentiate();

  cout << p << endl;
}
```

Automatic Differentiation:

Automatic differentiation (AD) is a technique for computing derivatives accurately and efficiently. Uses include: solving nonlinear equations, Sensitivity Analysis (first order), Parameter identification, Optimization and useful in Verification and Validation. Furthermore, no limits are imposed on the length or the complexity of the code comprising the function to be differentiated. Thereby, AD becomes essential and the most important tool in mathematics and scientific programming.

AD Implementation

C++ specialised overloading operators and templates have been used to implement and employ the reverse mode AD technique, which is the best known alternative to the forward mode. Derivatives are accumulated in the reverse order of program execution. Since this mode needs runtime tracing to store intermediate quantities that are required in the backward-pass, we have integrated a specialized memory manager/allocator to our implementation.

Using the meta-programming and template models for all known expressions, we have created a building block that integrates the AD package and allows all LS refinement, while taking care of many of the details that formerly had to be specified by the user. The user is now free to concentrate on the broader aspects of their problem.

Structure Factor Computation

This generic implementation results in more computation efficiency by making use of the structure factor expression found in International Tables for all the space groups, when identified as template argument, and by generating the corresponding code, so avoiding a general form calculation.

Refinement and NLS Model Formulation

We consider a general LS model form defined by:

1. X refineable parameter n-vector, in n-space $\mathbf{R}^n$;
2. Yo observation vector
3. W weight vector
4. Y(x) estimated function, f: $\mathbf{R}^n \Rightarrow \mathbf{R}^1$;
5. D set of admissible parameters, a subset of $\mathbf{R}^n$; defined by:

- l, u explicit, n-vectors finite bounds of x (an embedding 'box' in $\mathbf{R}^n$) ;
- g(x) general nonlinear constraint functions,
  g: $\mathbf{R}^n \Rightarrow \mathbf{R}^m$. (could be empty)

Applying the notation given above, the least squares method is stated as minimisation of the objective function :

$$M(x) = \text{sum } \{ \text{overall } \} W.(Yo - Y(x))^2.$$

minimise : M(x)

s.t : x **in** $D = \{ x: \; l \le x \le u$ **and** $g(x) \le 0\}$

where vector inequalities are component-wise

Whether we use C++ or the algebraic model, this environment provides an easy and natural way to formulate general nonlinear least squares problems required by small molecule crystallography. The refinement can be performed simply by specifying the expression form of the function to be fitted to the data, the desired residual/objective, as well as constraints and restraints (if any) in an algebraic notation, without having to indicate anything about the partial derivatives that a solver might require.

Solvers

Conventional crystallographic solvers are built-in; however, the open architecture has also enabled some useful external and modern non-linear solvers to be successfully interfaced to the system.

Solvers included:

- Normal matrix / LU /QR decomposition / SVD, CGradient,
- Generalized Minimum RESidual (GMRES)
- Levenberg - Marquardt non-linear minimisation with bounds on the parameters or linear constraints
- LBFGS with bounds on the parameters
- and IpOpt with bounds and general constraints

A simple Program in Algebraic Notation :

Fit plane to data points in 3D
Conceptualisation: Minimize Perpendicular Distance Points to Plane
In a traditional informal algebraic description, the implicit equation for a plane in 3D space is : ax+by+cz+d=0.
If the plane is not vertical (i-e: c not 0) the equation reduces to:
ax + by + z +d =0
The distance of a point (x,y,z) to the plane along a normal to the plane is:
Distance = | a * x + b * y + z + d | / sqrt(a^2 + b^2 + 1)
The model minimises the sum of the squared distances.

Algebraic Implementation

```
observation      x, y, z;
parameters       a,  b, d;
//minimise  sum {overall observation } ( obs - calc)^2

residual:
    abs( a * x + b * y + z + d )/ sqrt(a^2 + b^2 + 1);
// reads data points  x y z from 'experiment.dat' file
data    "experiment.dat";

 // then we call for refinement
refine;
print a,b,c;
```

Implementation in C++ Using the Smx Library :

```
//All we need is to define the template function to use in Least Squares
template<class num_t>
num_t funcLine ( Array2d<num_t> &p , Array2D<> &data)
{   // return distance
   return abs(p[0]* data[0] +  p[1]* data[1] + data[2]  + p[2]) /
                  sqrt( p[0]*p[0] + p[1]*p[1] +1);
}
...
// instantiate the least squares object using the template function with
// optional arguments:
   nl_lsq<funcLine> lsq(data_points, m, n, need_covar, max_iter);
//  generates the lsq object, the function and its gradient
//  then we call for minimisation
   ret = lsq.minimise( Observation,  Parameters );
```

This environment enables the user to specify any kind of relationship between the conventional crystallographic variables and any novel ones they need to introduce. The user will find it easy to create crystallographic models of great complexity with only a few statements.

SF Least Squares Snippet Simple Code in C++:

```
{
  //  smx::sfls::param_anisotropic_anamalous<int sgnumber=-1>

   smx::io:cif::CifReader cif( ciffilename );
   ScattererList Atoms(cif) ;                    //   + optional Atoms settings, filter ...

   // instantiate the SFLS class using functor param_anisotropic_anamalous<>
   // and read any extra  param from cif, e-g :OverAllscalef,  xFlack_param ...

   smx::sfls:: SfLs< param_anisotropic_anamalous<> > sfls(Atoms, cif);

   bool refine_f_square = true;
   Reflection hkl(hklfile);                      //  + optional hkl settings ...

   sfls.fix_scalefactor();
   //  Special postions and adp beta-restrictions constraints are handled internally
   sfls.refine_positions();                      // all positions
   sfls.refine_uij();                            // all adps
   sfls.refine_biso("C3");                       // refine atom C3 as isotropic
   sfs.set_constraint_eval_jac( eval_jac_g);  //  eval_jac_g() functor returns the sparsity structure
    // of the   Jacobian of the constraints, or the values for the Jacobian of the constraints at the point x.
   sfls.refine( hkl, refine_f_square );
}
```

Example of implicit linear constraint within the smx.Interpreter:
Consider an atom disordered on 2 positions C1 and C2. The refinement of the site occupation factors should be subjected to the constraint:

Occ1 + Occ2 = 1 # where Occ1, Occ2 are parameters

Or by using suffix :

Sites["C1"].Occ + Sites["C2"].Occ = 1; # where **Sites** is a set of parameters

This system is designed to facilitate the solution of nonlinear least squares and to support the whole *crystallographic modelling life-cycle* (building – refining – analyzing – revising). It is by supporting C++ interfaces and the algebraic modelling for crystallography that this platform will be accessible by all users including:

Structure Analyst End-users, Research Crystallographers or Crystallographic Programmers
Once completed, it will be a fundamental building block which will facilitate high quality analyses, helping to develop and explore the full capability of crystallography upon which other researchers can build new applications

This work is funded by EPSRC grant number EP/C536282/1