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 transform
- 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 analysis data in a clear and concise way. The model is used then as a basis to generate a mathematical representation that can be relaxed directly to the optimisation solver.

Example: Use of AD with smx.interpreter

```cpp
//  Declares 2 parameters
//  Simply assigned values initially
//  Generates the lsq object, the function and its gradient
// instantiate the SFLS class using functor param_anisotropic_anamalous::<class=num_t>
smx::sfls::SfLs< param_anisotropic_anamalous<> > sfls(Atoms, cif);  // instantiate the SFLS class using template function with any kind of parameters
ScattererList Atoms(cif);  // + optional Atoms settings, filter ...

//  functions

smx::inter::funcLine( Array2d<num_t> &p , Array2D<> &data)
int n = data.length();
int m = p.length();
return abs( a * x + b * y + z + d ) / sqrt(a^2 + b^2 + 1);
```

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 templates 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 factors expression found in International Tables for all space groups, when identified as template arguments, and by generating the corresponding code, so avoiding a general form calculation.

A simple Program in Algebraic Notation

```cpp
# SF Least Squares Snippet Simple Code in C++:

// smx::sfls::param_anisotropic_anamalous<int sgnumber=-1>
smx::sfls::SfLs< param_anisotropic_anamalous<> > sfls(Atoms, cif);  // instantiate the SFLS class using functor param_anisotropic_anamalous::<class=num_t>

//  instantiates the SFLS class using funcctor param_anisotropic_anamalous::<class=num_t>  
//  Read data points n x y z and vector inequalities are component-wise

bool refine_f_square = true;
bool refine_uij = true;
bool refine_positions = true;
bool refine_f_square = true;
bool refine_positions = true;

sfls.refine_f_square();           // refine_f_square with bounds on the parameters
sfls.refine_positions();          // all positions
sfls.refine_vo();                // all adps
sfls.refine_base("C3") ;         // refine atom C3 as isotropic
sfls.set_constraint_eval_jac( eval_jac_g); // eval_jac_g() functor returns the sparsity structure
sfls.set_consts( eval_jac_g, eval_jac_f );
// 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:

```cpp
// smx::sfls::param_anisotropic_anamalous<int sgnumber=-1>
smx::sfls::SfLs< param_anisotropic_anamalous<> > sfls(Atoms, cif);

bool refine_f_square = true;
bool refine_positions = true;

sfls.refine_f_square();           // refine_f_square with bounds on the parameters
sfls.refine_positions();          // all positions
sfls.refine_vo();                // all adps
sfls.refine_base("C3") ;         // refine atom C3 as isotropic
sfls.set_constraint_eval_jac( eval_jac_g); // eval_jac_g() functor returns the sparsity structure
sfls.set_consts( eval_jac_g, eval_jac_f );
// the Jacobian of the constraints, or the values for the Jacobian of the constraints at the point x
sfls.refine( hkl, refine_f_square );
```

This environment enables the user to specify any kind of relationships 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.

Refrainment and NLS Model Formulation

We consider a general LS model form defined by:

- l: Exp, n: vector, in: space R^n
- Yo: observation vector
- W: weight vector
- 2: estimated function, f: R^n -> R^n
- D: set of adim parameters, a subset of R^n, defined by:

- l, u: explicit, n: finite bounds of x (an embedding ’box’ in R^n)
- g: general nonlinear constraint functions, g: R^n -> R^n (could be empty)
- Applying the notation given above, the least squares method is defined as minimisation of the objective function:

  \[ \text{Minimize } f(x) \equiv \sum_{i=1}^{m} \left( w_i (y_i - f(x_i))^2 \right) \]

where vector inequilities 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 constraints 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
- Generalised Minimum RESidual (GMRES)
- Levenberg - Marquardt non-linear minimisation with bounds on the parameters or linear constraints
- LBBFGS with bounds on the parameters
- and stdOpt with bounds and general constraints

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
- Crystallographic Computational Infrastructure