Appendices

Appendix I. Estimation of the number of molecules and pressure of the molecular beam at the interaction region.

This approximation is based on experimental measurements of the residual background gas (CO_2) taken in early September 2005. The logic behind the approximation is that this number can be calculated using readily measured experimental parameters. The pressure of the residual background gas is known from the ion gauge in the scattering chamber. The number of electrons scattering from the molecular beam can be measured, as can the number of electrons scattering from the residual background gas by moving the molecular beam out of the path of the electron beam. The ratio of scattering events is proportional to the number of molecules in the two regions.

The electron beam is measured to have 33,000 electrons/pulse. The pressure of CO_2 in the scattering chamber (residual background gas pressure) is 1.3×10^{-4} torr. The image intensifier gain is set at 825 V, the camera pixels are binned 2×2 (512×512 pixels

effective camera dimensions), and 4 minute exposure times are being used (240000 electron pulses).

With the molecular beam centered in the path of the electron beam the mean intensity measured in 5537 ADU (arbitrary digital units). With the molecular beam far to the left of the electron beam, the mean intensity measured is 1368 ADU. The dark current background of the detector is 490 ADU. Using the calibration for ADU/electron, 93.7 ADU/electron/pixel at 825 V, (the number reported in Section 3.2.1 must be divided by 2 to adjust for the difference in image intensifier gain and multiplied by 4 since binning is 2×2) the two molecular beam conditions amount to 1.24×10^{7} and 2.46×10^{6} scattered electrons, respectively. Note that to obtain the number of scattered electrons from the molecular beam half the number of electrons scattered from the background gas must be subtracted (half, since the electrons must traverse half the scattering chamber to reach the molecular beam).

The path the electron beam travels through the scattering chamber is roughly a cylinder 34.6 cm long and 350 μm wide (the typical FWHM of the electron beam) with a calculated volume ($V = \pi r^2 h$) of 3.33×10^{-5} ℓ. Using the ideal gas law ($n = pV/RT$) and assuming that only single scattering events occur, the number of background gas molecules in the beam path is calculated as 8.36×10^{10}. Then the following proportion relation can be used:

$$\frac{N_{bkg}}{N_{MB}} = \frac{N_{bkg}^{S}}{N_{MB}^{S}}, \tag{A1.1}$$

where N_{bkg} is the number of background gas molecules in the electron beam path, N_{MB} is the number of molecules in the interaction region, N_{bkg}^{S} is the number of electrons scattered from the background gas, and N_{MB}^{S} is the number of electrons scattered from the interaction region. By plugging in the values for N_{bkg}, N_{bkg}^{S}, and N_{MB}^{S}, the number of molecules in the interaction region, N_{MB}, is found to be 4.39×10^{11}.

Assuming that the interaction region is a cube 350 μm on a side, the ideal gas law can be used again to obtain the pressure in the region. p = 0.53 torr.

Appendix II. The UED_2004Doc class in the UED_2004 program: "UED_2004Doc.h"

This class contains all the information of a structural refinement: all parameters, data, scattering factors, scaling atoms, active pixel range, etc. CUED_2004Doc is the data storage "document" class in the UED_2004 program. See comments for brief descriptions of each datum.

```
double raw_data[NUM_PIX];       // raw data from .xl0 file
double raw_sigma[NUM_PIX];      // sigma values for raw_data from .snr file
CString data_file;              // the name of the data file
CString sigma_file;             // the name of the sigma file
double s[NUM_PIX];              // s as a function of pix
double delta_s[NUM_PIX];        // the width of each pixel in s for integration approximation
CRange ActiveRange;                     // range of data used for fitting, etc. (in pixels)
BOOL datafile_loaded;                   // true if raw data loaded
BOOL sigmafile_loaded;                  // true if raw sigmas loaded
BOOL data_loaded;                       // true if data loaded
BOOL sigma_loaded;                      // true if sigmas are loaded
BOOL molecule_loaded;                   // true if at least on molecule is loaded, IM(s) calc'd
BOOL theory_loaded;                     // true if theory is loaded, sM(s) calculated
BOOL s_flag;                            // true if is s calculated
BOOL elements_loaded;                   // are the elements loaded?

BOOL scatt_factor_flag;                         // are scattering factors loaded
double scat_factor_step_size;                   // step in s of terms in scattering factor files
int num_raw_scat_factors;                       // S_MAX / SCAT_FACTOR_STEP_SIZE
```

```
double *raw_f_factors[NUM_ELEMENTS];          // the raw f values from the file
double *raw_n_factors[NUM_ELEMENTS];          // the raw eta values from the file
double *raw_S_factors[NUM_ELEMENTS];          // the raw S values from the file
double f_factor[NUM_ELEMENTS][NUM_PIX];       // f - elastic scattering factors
double n_factor[NUM_ELEMENTS][NUM_PIX];       // n - (eta) phase factors
double S_factor[NUM_ELEMENTS][NUM_PIX];       // S - inelastic scattering factors

CString ffactorfile;                          // the name of the file from which f loaded
CString nfactorfile;                          // the name of file from which etas loaded
CString Sfactorfile;                          // the name of the file from which S loaded
double camera_length;                         // camera length
double electron_KE;                           // electron kinetic energy
double pixel_size;                            // size of pixels
double wave_constant;                         //  4pi/lamda
double sMs[NUM_PIX];                          // the theoretical sM(s) curve
double scale_factor;                          // scale factor for data
double atomic_scale;                          // scale factor for atomic component (1)
int ebeam_width;                              // FWHM of the electron beam

CString elements_file;                        // the file containing the supported elements
int num_elements;                             // number of supported elements
int *elements;                                // atomic numbers of supported elements
int scaling_atom1;                            // index of scaling atom1 in elements[]
int scaling_atom2;                            // index of scaling atom2 in elements[]
BOOL reference;                               // is there an atomic reference gas?
int atomic_ref;                       // index of atomic reference gas in elements[]

BOOL inelastic_flag;          // include inelastic scattering factors in atomic scattering?

int poly_bkg_order;                           // order of polynomial background
double poly_bkg[MAX_POLY_ORDER];              // coefficients of the polynomial background

BOOL InitDoc();                               // initializes the document data
BOOL ClearDoc();                              // clears the document data

CRange frRange;                               // range in angstroms for f(r) (e.g. 0-7)
BOOL TheoryfrLoaded;          // TRUE if the theoretical f(r) curves have been calculated
BOOL ExptfrLoaded;            // TRUE if the experimental f(r) curves have been calculated
double damping_constant;              // the k in the exp(k s^2) term in calculation of f(r)

int num_models;               //the current number of CModel objects (# of molecules)

void OnLoadData(double *xl0, CString filename);   // load the date from xl0[]
void OnLoadSigma(double *snr, CString filename);  // load the sigmas from snr[]

double out_data[NUM_PIX];                         // final output data
double out_sigma[NUM_PIX];                        // final output sigma
double out_theory[NUM_PIX];                       // final output theoretical data

BOOL IsDataLoaded(void) { return data_loaded; }       // returns true if data is loaded
BOOL IsSigmaLoaded(void) { return sigma_loaded; }     // returns true if sigmas are loaded
BOOL IsTheoryLoaded(void) { return theory_loaded; }   // returns true if sM(s) is calc'd
```

```
BOOL IsMoleculeLoaded(void) { return molecule_loaded; }          // returns true if at least one
                                                                 molecule is loaded, IM(s) calc'd
BOOL Is_s_Calcd() {return s_flag;}                               //true if s calculated
BOOL IsElementsLoaded() {return elements_loaded;}                // true if the elements file is loaded
BOOL IsScattLoaded() {return scatt_factor_flag;}     //true if scattering factors loaded
BOOL IncludeInelastic() {return inelastic_flag;}     // true if including inelastic scattering contributions
void SetInelasticFlag(BOOL flag);                    // sets whether inelastic factors used
BOOL UseAtomicRef() {return reference;}              // true if reference atom being used
void SetAtomicRefFlag(BOOL flag);                    // sets whether reference atom used

int GetRangeMin(void) {return ActiveRange.range_min;}     // returns sMs range minimum (pixels)
int GetRangeMax(void) {return ActiveRange.range_max;}     // returns sMs range maximum (pixels)
void SetRangeMin(int xmin);                               // sets sMs range minimum (pixels)
void SetRangeMax(int xmax);                               // sets sMs range maximum (pixels)

double *fr_theory;                                        // theoretical f(r)
double *fr_expt;                                          // experimental f(r)

BOOL IsTheoryfrLoaded() {return TheoryfrLoaded;}          // returns TheoryfrLoaded;
BOOL IsExptfrLoaded() {return ExptfrLoaded;}              // returns ExptfrLoaded;
int GetfrRangeMin(void) {return frRange.range_min;}       // returns fr range minimum (angstroms)
int GetfrRangeMax(void) {return frRange.range_max;}       // returns fr range maximum (angstroms)
void SetfrRange(int xmin, int xmax);                      // sets fr range      (angstroms)
void CalcRadialDistribution();                            // calculates the f(r) curves
double GetDampConst() {return damping_constant;}          // returns the damping constant, k
void SetDampConst(double dc);                    // change the value of the sMs damping constant

double GetfFactor(int i, int j) {if(j>=0 && j<NUM_PIX && i>=0 && i<NUM_ELEMENTS)
        return f_factor[i][j]; else return 0.0;}          // returns elastic factor at j pixel for i element
double GetnFactor(int i, int j) {if(j>=0 && j<NUM_PIX && i>=0 && i<NUM_ELEMENTS)
        return n_factor[i][j]; else return 0.0;}          // returns eta factor at j pixel for i element
double GetSFactor(int i, int j) {if(j>=0 && j<NUM_PIX && i>=0 && i<NUM_ELEMENTS)
        return S_factor[i][j]; else return 0.0;}          // returns S factor at j pixel for i element

void SetfFactor(int i, int j, double ff)
        {if(j>=0 && j<(S_MAX/SCAT_FACTOR_STEP_SIZE) && i>=0 && i<NUM_ELEMENTS)
        raw_f_factors[i][j] = ff;}   // sets f factor for j = s, and i = element
void SetnFactor(int i, int j, double nf)
        {if(j>=0 && j<(S_MAX/SCAT_FACTOR_STEP_SIZE) && i>=0 && i<NUM_ELEMENTS)
        raw_n_factors[i][j] = nf;}  // sets eta factor for j = s, and i = element
void SetSFactor(int i, int j, double Sf)
        {if(j>=0 && j<(S_MAX/SCAT_FACTOR_STEP_SIZE) && i>=0 && i<NUM_ELEMENTS)
        raw_S_factors[i][j] = Sf;}  // sets S factor for j = s, and i = element
double GetRawfFactor(int i, int j)
        {if(j>=0 && j<(S_MAX/SCAT_FACTOR_STEP_SIZE) && i>=0 && i<NUM_ELEMENTS)
        return raw_f_factors[i][j]; else return 0.0;}   // returns f factor at j = s, for i element
double GetRawnFactor(int i, int j)
        {if(j>=0 && j<(S_MAX/SCAT_FACTOR_STEP_SIZE) && i>=0 && i<NUM_ELEMENTS)
        return raw_n_factors[i][j]; else return 0.0;}   // returns eta factor at j = s, for i element
double GetRawSFactor(int i, int j)
        {if(j>=0 && j<(S_MAX/SCAT_FACTOR_STEP_SIZE) && i>=0 && i<NUM_ELEMENTS)
        return raw_S_factors[i][j]; else return 0.0;}   // returns S factor at j = s, for i element
CString GetfFactorFile(void) {return ffactorfile;}       // returns f factor file name
```

```
CString GetnFactorFile(void) {return nfactorfile;}     // returns eta factor file name
CString GetSFactorFile(void) {return Sfactorfile;}     // returns S factor file name
void SetfFactorFile(CString ffile);                    // sets f factor file name
void SetnFactorFile(CString nfile);                    // sets eta factor file name
void SetSFactorFile(CString Sfile);                    // sets S factor file name
double GetScatFactorStepSize() {return scat_factor_step_size;}// returns step size (in s) in scatt factor files
void SetScatFactorStepSize(double step);               // sets step size (in s) in scatt factor files

CString GetDataFile() {return data_file;}              // returns name of .xl0 data file
CString GetSigmaFile() {return sigma_file;}            // returns name of .snr file
double GetRawData(int pixel) {return raw_data[pixel];}         // returns value of .xl0 at pixel
double GetRawSigma(int pixel) {return raw_sigma[pixel];}     // returns value of .snr at pixel

void LoadElements(int num, int *z);            // loads array of Z (*z) for # (num) of supported elements
int GetElement(int i)                          // returns the atomic number of element at index=i
        {if(elements_loaded) return elements[i]; else return 0;}
int GetNumElements() {return num_elements;}        // returns the number of supported elements
CString GetElementsFile() {return elements_file;}     // returns the title of the supported elements file
void SetElementsFile(CString fn);                      // sets the name of the supported elements file

void SetScaleAtoms(int atom1, int atom2);              // sets the I, J atoms for s/fIfJ
int GetScaleAtom1() {return scaling_atom1;}        // returns I (scale atom #1)
int GetScaleAtom2() {return scaling_atom2;}        // returns J (scale atom #2)
void SetAtomicRef(int ref_gas);                        // sets the reference atom (index in elements file)
int GetAtomicRef() { return atomic_ref;}               // returns index in elements file for ref atom

double GetCamDist(void) {return camera_length;}    // returns the camera length
void SetCamDist(double camdist);                       // sets the camera length
double GetScaleFactor(void) {return scale_factor;}     // returns the data scale factor
void SetScaleFactor(double scale);                     // sets the data scale factor
void SetAtomicScale(double scale);                     // sets the atomic scattering scale factor
double GetAtomicScale() {return atomic_scale;}         // returns the atomic scale factor
double GetElectronKE(void) {return electron_KE;}   // returns the electron kinetic energy
double GetPixSize(void) { return pixel_size;}          // returns the detector pixels size
void SetElectronKE(double eKE);                        // sets the electron kinetic energy
void SetPixSize(double pix);                           // sets the detector pixel size
int GeteBeamWidth() {return ebeam_width;}          // returns the FWHM of the electron beam
void SeteBeamWidth(int width);                         // sets the FWHM of the electron beam
double Get_s(int i) {if(i>=0 && i<NUM_PIX) return s[i]; else return 0.0;} // returns the value of s at pixel i
double GetDelta_s(int i) {if(i>=0 && i<NUM_PIX) return delta_s[i]; else return 0.0;}
                                                       // returns the value of delta s at pixel i

CString* StatusLog;                                    // status of data – the log file
int LogLoc;                                            // number of lines in StatusLog
void AddStatusString(CString NewLog);                  // adds a string to the StatusLog
void SaveStatus();                                     // Saves the StatusLog to a file
void ClearLog();                                       // Clears the LogView, empties StatusLog[]
BOOL IsLogFile;                                        // Has a log file been chosen?
CString LogFile;                                       // the log file name

void Calc_s(void);                                     // Calculates s at each pixel
void LoadScatteringFactors(void);                      // Loads scattering factors from files
```

```
void CalcIMs();                         // calculates the theoretical molecular scattering IM(s)T
void CalcIMs(int index);   // calculates the theoretical molecular scattering IM(s)T for nulecule[index]
int sms_mode;                                   // how to prepare theoretical sMs
int bkg_mode;                                   // how to prepare the poly background
void PrepareData();                             // prepares experimental sMs from raw data
void PrepareTheory();                           // prepares theoretical sMs

CModel* nulecule;                       // the array of molecules and all their parameters
BOOL ModelLoaded;                       // has at least one CModel been loaded?
int GetNumModels() {return num_models;} //returns the number of molecules
void SetNumModels(int num);                     // sets new CModel array

CModel* InitModels;                     // initial values of the models as loaded from input files
void LoadInitModels();                  // loads the initial values into the InitModels array
BOOL InitialsLoaded;                    // Has the ModelInit array been allotted?
double DeltaQ();                        // the value of difference between fit and initial values
double GetInitParam(int which_param);   // returns the initial value of FitParamLabel[which_param]
                                        from InitModels[FitParamMolIndex[which_params]]

double tolerance;                               // non-linear chi-squared fitting tolerance

BOOL ChisqrR();                                 // calculates Chi Squared and R
double chisqr;                                  // Chi Squared
double R;                                       // R value

int GetPolyBkgOrder()
        {return poly_bkg_order;}        // returns the order of the polynomial used for the background
void SetPolyBkgOrder(int order) ;       // set the order of the polynomial background
double GetPolyBkg(int order)                    // returns coeficient of s^order
        {if(order>=0 && order<MAX_POLY_ORDER)
        return poly_bkg[order]; else return 0.0;}
void SetPolyBkg(int order, double coef) ;       // sets the coefficient (coef) of the polynomial at order term

int NumParams;                          // total number of fittable non-linear parameters
BOOL ParamLoadedFlag;                   // Have the parameters been loaded?
BOOL ParamReadyFlag;                    // Are the parameters ready for fitting?
CString* FitParamLabels;                // parameter labels: r1,r2,a2,T, etc.
CString* FitParamValues;        // use string format so that first character can be a '=' if it is a dependent
double* FitParamResults;        // parameters after fitting
double* FitParamErrors;                 // error bars for the fitted parameters
int* FitParamMolIndex;                  // index of molecule in nulecule CModel array
BOOL* FitThisParam;                     // 1 if the parameter is to be fitted
void PrepareFitParams();        // fills the previous several arrays with the nonlinear parameters
void UpdateModel(int index);    // Updates nulecule[index]

BOOL ProductOnly_flag;                  // make sMs and fr product only format
```

Appendix III. The global constants in UED_2004 program: “define.h”

The define.h header file declares all the constants used in the program. These are used as initial values for some conditional parameters and most can be changed via GUI windows. Most are self-explanatory, but comments are included for further clarity.

```
const int NUM_PIX = 256;                                    // number of pixels in .xl0 data

const int G_WIDTH = 650;                                    // parameters for sMs plot
const int G_HEIGHT = 340;
const int G_LEFT = 70;
const int G_RIGHT = 580;
const int G_TOP = 25;
const int G_BOTTOM = 295;

const int SMALL_PIX_STEP = 1;                   // pixel stepping parameters for sMs plot,  arrow keys
const int BIG_PIX_STEP = 5;

const int STATUS_MAX = 200;                     // line of text in CLogView::StatusLog

const double PIX_SIZE = 115.6;                  // pixel size (microns)
const double CAM_DIST = 13.485147;              // Camera length (cm);
const int EBEAM_WIDTH = 3;                      // width of electron beam (in pixels)
const double ELECTRON_KE = 30;                  // electron kinetic energy (kV)

const double PI = 3.14159265359;                // pi.  mmmmm pi
const double BOHR = 0.52918;                    // Bohr radius in Angstroms

const int NUM_ELEMENTS = 14;                    // number of elements supported by this program

const CString ELEMENTS_FILE = "Elements_2004.dat";
                                                // lists elements for which we have scattering factors

const double SCAT_FACTOR_STEP_SIZE = 0.01;          //  step in s for scattering factors in data files

const CString F_FACTOR_FILE = "sf_FMAG30K.dat";     //  elastic scattering factor file name
const CString N_FACTOR_FILE = "sf_FTHE30K.dat";     //  phase factor file name
const CString S_FACTOR_FILE = "s30K.dat";           //  inelastic scattering factor file name

const double S_MAX = 20.0;                          // maximum value s can hold

const int MAX_MODELS = 10;                          // number of model files supported
const int MAX_POLY_ORDER = 5;          // maximum order of polynomial background order 10 -> s^9

const int MLS = 0;                                  // Model from .mls file
const int XYZ = 1;                              // Model from cartesian coordinates in .xyz file
const int ZMX = 2;                              // Model a z-matrix from .zmx file
const int NEWFILE = 0;
const int OLDFILE = 1;
```

```
const int SMS_SFIFJ = 0;                    // make sMs by multiplying Ms by s and dividing by fifj
const int SMS_SATOMIC = 1;                  // make sMs by multiplying Ms by s and dividing by the atomic scatting
const int SMS_S5 = 2;                       // make sMs by multiplying Ms by s to the fifth power

const double RESOLUTION = 0.0001;           // the number at which two distances are considered equal in mls format (angstroms)

const int BKG_IREFT = 0;                    // multiply poly bkg by IRefT (like in UEDana)
const int BKG_ONE = 1;                      // leave poly bkg untouched
const int BKG_1OVERS = 2;                   // multiply poly bkg by 1/s
const int BKG_1OVERS2 = 3;                  // multiply poly bkg by 1/s^2

const int MAX_ELEMENTS = 104;
const CString ELEMENTS[MAX_ELEMENTS] =
{"H","He","Li","Be","B","C","N","O","F","Ne","Na","Mg","Al","Si",
          "P" ,"S" ,"Cl","Ar","K" ,"Ca","Sc","Ti","V" ,"Cr","Mn","Fe","Co","Ni","Cu","Zn","Ga","Ge",
          "As","Se","Br","Kr","Rb","Sr","Y" ,"Zr","Nb","Mo","Tc","Ru","Rh","Pd","Ag","Cd","In","Sn",
          "Sb","Te","I" ,"Xe","Cs","Ba","La","Ce","Pr","Nd","Pm","Sm","Eu","Gd","Tb","Dy","Ho","Er",
          "Tm","Yb","Lu","Hf","Ta","W" ,"Re","Os","Ir","Pt","Au","Hg","Tl","Pb","Bi","Po","At","Rn",
   "Fr","Ra","Ac","Th","Pa","U" ,"Np","Pu","Am","Cm","Bk","Cf","Es","Fm","Md","No","Lr","X"};

const CString FILE_ID = "UED_2004";       // first piece of data in files associated with this program

const int FR_MIN = 0;                           // range min for f(r) in angstroms
const int FR_MAX = 7;                           // range max for f(r) in angstroms
const double DAMPING_CONSTANT = 0.005;      // damping constant required for calculation of f(r)

// constants for least squares fitting
const int MAX_ITERATIONS = 100;             // maximum number of iterations
const int LAMDA_OVERFLOW = 100000;          // maximum value of lamda
const double DELTA_FOR_DERIV = 0.01;        // step around center point to find slope
const double TOLERANCE = 0.000001;          // difference in chiqsr between 2 steps marking stationary point
```

Appendix IV. The molecular model – the CModel object.

The class CModel contains all the information for a single molecule in the UED_2004 program. Model.cpp contains the code for loading a molecule as either a z-matrix (zmx), a list of Cartesian coordinates (xyz), or a list of internuclear distances with degeneracies, mean amplitudes of vibration, and anharmonicities (mls). This class is the

heart of UED theory and is included here in near completeness. First is the header file for a declaration of the class elements.

Header file, “Model.h”

```
class CAtom {                    // one atom
        public:
                int atomic_number;
                double atomic_mass;
                CString name;
                int order;
                CString label;
                double xyz[3];

                CAtom();
                void init(CString, int);

                inline bool operator==(const CAtom& first) const;
};

class CBond {                    // one bond as defined in the z-matrix
        public:
                double value;
                CAtom center1;
                CAtom center2;
                CString label;

                CBond();
                void init(CAtom,CAtom);
};

class CAngle {                   // one angle as defined in the z-matrix
        public:
                CAtom center1;
                CAtom center2;
                CAtom center3;
                double value;
                CString label;

                CAngle();
                void init(CAtom,CAtom,CAtom);
};

class CDihedral {                // one dihedral as defined in the z-matrix
        public:
                CAtom center1;
                CAtom center2;
                CAtom center3;
                CAtom center4;
                double value;
                CString label;
```

```
                CDihedral();
                void init(CAtom,CAtom,CAtom,CAtom);
};

class CDistance {                   // internuclear distance object (for each internuclear distance)
        public:
                CAtom pair[2];  // the two atoms comprising the pair
                double r;           // the distance between the two atoms
                int multi;          // degeneracy of the distance  (e.g  # of equivalent C-C distances)
                double l;                   // mean amplitude of vibration
                double lm;
                double temperature;         // temperature of distance
                double a;                   // anharmonicity
                double dr;                  // centrifugal distortion
                double k;                   // k vector

                CDistance();
         inline bool operator==(const CDistance& first) const;
};

CString input_file;                         // file from which model is loaded
CString label;                              // title of model from file header

CDistance* distance;                        // array of interuclear distances
int num_r;                                  // number of unique internuclear distances

int type;                                   //  mls = MLS = 0, xyz = XYZ = 1, zmx = ZMX = 2

int num_atoms;                              // number of atoms present in the molecule
CAtom* atoms;                               // all atoms in the molecule
int num_dummy;                              // number of dummy atoms

int num_par_zmx;                            // number of fittable parameters in ZMX
CBond* bonds;                               // bonds in the zmx
CAngle* angles;                             // angles in the zmx
CDihedral* dihedrals;                       // dihedral angles in the zmx
CString* param_names;   // the labels of the unique fitting parameters for ZMX format
CString* param_vals;    // the values of the unique parameters in string format for ZMX

BOOL empty;                                 //  is some model loaded
BOOL mls_flag;                  // has mls file been loaded (CDistance* distance created)

double fraction;                // percent of this molecule in total mixture
double temperature;             // vibrational temperature

int num_params;                 // number of parameters that may be fitted for all formats

double* sMsSingle;                          // theoretical IM(s)for this molecule
BOOL sms_flag;                              // is the IM(s) loaded

BOOL po_flag;                               // is this molecule considered parent for product only
```

```
void ReadMLS(CString filename, int read);  // read structure from mls file
void ReadXYZ(CString filename, int read);
        // read structural input from an .xyz file of Cartesian coordinates
void ReadZMX(CString filename, int read); // read the structure from a .zmx z-matrix file
CModel& operator=(const CModel& other);          // assignment operator
void ClearAll();                                 // clear this object
void ZMXtoXYZ();                                 // convert the z-matrix to Cartesian coordinates
void XYZtoMLS();                                 // convert the Cartesians to internuclear distances
void SetTemperature(double new_temp);     // set the temperature of the molecule
void Calculate_l();                              // calculate the mean amplitudes of vibration

int GetNumParams();                 // returns the number of fittableparameters in this molecule
void GetParams(CString* par_names, CString* par_vals);       // get those parameters
void UpdateParams(CString* par_names, CString* par_vals);  //update the fitted parameters

double CalcIt(CString equation);                 // returns solution to equation, for dependencies
```

The "Model.cpp" file – the code.

```
CModel::CModel()                    // constructor
{
        empty = TRUE;
        fraction = 0.0;
        temperature = 298;
        input_file = "";
        label = "";
        type = MLS;
        mls_flag = FALSE;
        num_r = 0;
        sms_flag = FALSE;
        num_params = 0;
        num_dummy = 0;
        po_flag = FALSE;
}

CModel::~CModel()                   // destructor
{
        if(!empty && mls_flag){
                delete [] distance;
                delete [] sMsSingle;
                delete [] atoms;
        }
        if(!empty && type==XYZ && !mls_flag) delete [] atoms;
        if(!empty && type==ZMX) {
                if(!mls_flag) delete [] atoms;
                if(num_atoms > 1) delete [] bonds;
                if(num_atoms > 2) delete [] angles;
                if(num_atoms > 3) delete [] dihedrals;
                delete [] param_names;
                delete [] param_vals;
        }
}
```

```
CModel::CAtom::CAtom()            // constructor of the CAtom object
{
        atomic_number = 104;
        name = "X"; label = "X";
        atomic_mass = 0.0;
        order = 0;
        xyz[0]=0.0;xyz[1]=0.0;xyz[2]=0.0;
}

void CModel::CAtom::init(CString abbrev, int index)            //assigning values to CAtom members
{
        CString bad = " 0123456789\t\n,";
        int place = abbrev.FindOneOf(bad);
        CString  temp;
        const double ATOMIC_MASSES[MAX_ELEMENTS] =
        {1.008,4.003,6.941,9.012,10.81,12.01,14.01,16.00,19.00,20.18,22.99,24.30,26.98,28.09,30.97,32.
        07,35.45,39.95,39.10,40.08,44.96,47.88,50.94,52.00,54.94,55.85,58.93,58.69,63.55,65.39,69.72,7
        2.61,74.92,78.96,79.90,83.80,85.47,87.62,88.91,91.22,92.91,95.94,98.91,101.1,102.9,106.4,107.9,
        112.4,114.8,118.7,121.8,127.6,126.9,131.3,132.9,137.3,138.9,140.1,140.9,144.2,144.9,150.4,152.
        0,157.2,158.9,162.5,164.9,167.3,168.9,173.0,175.0,178.5,180.9,183.8,186.2,190.2,192.2,195.1,19
        7.0,200.6,204.4,207.2,209.0,210.0,210.0,222.0,223.0,226.0,227.0,232.0,231.0,238.0,237.0,239.1,2
        43.1,247.1,247.1,252.1,252.1,257.1,256.1,259.1,260.1,0.0};

        if(place > 0) temp = abbrev.Left(place);
        else temp = abbrev;
        for(int i=0;i<104;i++){
                if(temp.CompareNoCase(ELEMENTS[i])==0){ atomic_number = i+1; name =
                ELEMENTS[i]; label = abbrev;}
        }
        atomic_mass = ATOMIC_MASSES[atomic_number - 1];
        order = index;
        xyz[0]=0.0;xyz[1]=0.0;xyz[2]=0.0;
}

inline bool CModel::CAtom::operator==(const CModel::CAtom& first) const
{
        return (first.atomic_number == atomic_number && first.order == order);
}

CModel::CBond::CBond()                    // CBond object constructor
{
        center1.init("X",1);
        center2.init("X",2);
        value = 0.74;
}

void CModel::CBond::init(CAtom one, CAtom two)
{
        center1 = one;
        center2 = two;
        value = 0.0;            //dummy value
}
```

```
CModel::CAngle::CAngle()                    // CAngle constructor
{
        center1.init("X",1);
        center2.init("X",2);
        center3.init("X",3);
        value = 180.0;
}

void CModel::CAngle::init(CAtom one, CAtom two, CAtom three)
{
   center1 = one;
   center2 = two;
   center3 = three;
   value = 0.0;             //dummy value
//      dependent = FALSE;
}

CModel::CDihedral::CDihedral(){                     // CDihedral constructor
    center1.init("X",1);
    center2.init("X",2);
    center3.init("X",3);
    center4.init("X",4);
    value = 180.0;
 //     dependent = FALSE;
}

void CModel::CDihedral::init(CAtom one, CAtom two, CAtom three, CAtom four)
 {
    center1 = one;
    center2 = two;
    center3 = three;
    center4 = four;
    value = 360.0;      //dummy value
 //     dependent = FALSE;
}

CModel::CDistance::CDistance()    // CDistance constructor – one for each unique internuclear distance
{
        pair[0].atomic_number = 0; pair[1].atomic_number = 0; r = 0.0; multi = 1; //done = false;
        temperature = 298; l = 1.0; a = 0.0; dr=0.0; k=0.0;
}

inline bool CModel::CDistance::operator==(const CDistance& first) const
{
        return (fabs(first.r - r) < RESOLUTION)
                &&((first.pair[0].atomic_number == pair[0].atomic_number &&
                first.pair[1].atomic_number==pair[1].atomic_number)
                || (first.pair[0].atomic_number == pair[1].atomic_number &&
                first.pair[1].atomic_number==pair[0].atomic_number));
}

void CModel::ReadMLS(CString filename, int read)        // reads distances from the .mls file
```

```
{
	INT_PTR result;
	BOOL status;

	CString datext = "MLS";
	CString datfilter ="Molecule input file (*.mls) |*.MLS||";

	CFileDialog datafiledialog(TRUE, datext, NULL,
	OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
	datfilter, NULL);
	if (read==NEWFILE) {result = datafiledialog.DoModal();datext=datafiledialog.GetPathName();}
	else {datext=filename;result =IDOK;}

	input_file = datext;
	if(result != IDOK) return;
	CStdioFile mls;
	CFileException exception;
	status = mls.Open(input_file,CFile::modeRead,&exception);

	if(!status){
		char s[100];
		sprintf(s,"Error Opening File for Reading. Code:%d",exception.m_cause);
		return;
	}

	if(!empty) ClearAll();
	empty = false;
	type = MLS;

	CString* temp_pairs;
	char temp1[50];
	char input[100];
	CString one,two, three;
	CString atomic;
	CString formula;
	int howmany;
	int i,j;
	int atom_index = 0;
	CString letters;
	CString numbers;

	mls.ReadString(input, sizeof input);
	label.Format("%s",input); label.Trim();
	mls.ReadString(input,sizeof input);
	sscanf(input,"%d %s",&num_r,&temp1);
	formula.Format("%s",temp1); formula.Trim();

	temp_pairs = new CString[num_r];
	distance = new CDistance[num_r];
	for(j=0;j<num_r;j++) {
		mls.ReadString(input,sizeof input);

		sscanf(input,"%s %d %le %le %le",&temp1,&distance[j].multi,
```

```
                &distance[j].r,&distance[j].l,&distance[j].a);
        distance[j].lm = distance[j].l*distance[j].l+1.5*distance[j].a*distance[j].a*
                distance[j].l*distance[j].l*distance[j].l*distance[j].l;
        distance[j].dr=1.5*distance[j].a*distance[j].lm*distance[j].lm;
        distance[j].k=distance[j].l*distance[j].l*distance[j].l*distance[j].l/6;
        temp_pairs[j] = temp1;
}
for(j=0;j<num_r;j++){
        if(temp_pairs[j].GetLength()==2) {
                one = _T(temp_pairs[j][0]); two = _T(temp_pairs[j][1]);
                distance[j].pair[0].init(one,j); distance[j].pair[1].init(two,j);
        }
        if(temp_pairs[j].GetLength()==3) {
                if(islower(temp_pairs[j][1])) {one = temp_pairs[j].Left(2);
                        _T(two=temp_pairs[j][2]);}
                if(islower(temp_pairs[j][2])) {one = _T(temp_pairs[j][0]); two =
                        temp_pairs[j].Right(2);}
                distance[j].pair[0].init(one,j); distance[j].pair[1].init(two,j);
        }
        if(temp_pairs[j].GetLength()==4) {
                one = temp_pairs[j].Left(2); two = temp_pairs[j].Right(2);
                distance[j].pair[0].init(one,j); distance[j].pair[1].init(two,j);
        }
        distance[j].temperature = temperature;
}
delete [] temp_pairs;

mls.Close();

// now put distances in numerical order
CDistance temp_r1;
for(j=0;j<num_r;j++){
        temp_r1 = distance[j];
        for(int i=j+1;i<num_r;i++){
                if(distance[i].r < distance[j].r) {
                        temp_r1 = distance[j];
                        distance[j] = distance[i];
                        distance[i] = temp_r1;

                }
        }
}
int num_dists = 0;
for(j=0;j<num_r;j++) num_dists += distance[j].multi;
num_atoms = int(sqrt(double(2.0*num_dists))) + 1;

// get the atom array //
atoms = new CAtom[num_atoms];
numbers = "123456789";
letters = "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz";
while(atom_index<num_atoms){
        atomic = formula.Left(formula.FindOneOf(numbers));
        one = formula.Right(formula.GetLength() - formula.FindOneOf(numbers));
```

```
                if(one.FindOneOf(letters) < 0) two = one;
                else two = one.Left(one.FindOneOf(letters));
                sscanf(two,"%d",&howmany);
                for(i=0;i<howmany;i++){
                        three.Format("%s%d",atomic,atom_index);
                        atoms[atom_index].init(three, atom_index);
                        atom_index++;
                }
                if(one.FindOneOf(letters) == -1) break;
                formula = one.Right(one.GetLength() - one.FindOneOf(letters));
        }
        num_dummy=0;
        mls_flag = TRUE;
        sMsSingle = new double[NUM_PIX];
}
```

Read structure in Cartesian coordinates from a .xyz file.

```
void CModel::ReadXYZ(CString filename, int read)
{
        INT_PTR result;
        BOOL status;
        CString datext = "XYZ";
        CString datfilter ="Molecule input file (*.xyz) |*.XYZ||";

        CFileDialog datafiledialog(TRUE, datext, NULL,
        OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        datfilter, NULL);
        if (read==NEWFILE) {result = datafiledialog.DoModal();datext=datafiledialog.GetPathName();}
        else {datext=filename;result =IDOK;}
        if(result != IDOK) return;

        if(!empty) ClearAll();
        empty = false;
        input_file = datext;
        type = XYZ;
        fraction = 0.0;
        temperature = 298;

        CStdioFile xyz;
        CFileException exception;
        status = xyz.Open(input_file,CFile::modeRead,&exception);

        if(!status){
                char s[100];
                sprintf(s,"Error Opening File for Reading. Code:%d",exception.m_cause);
                return;
        }

        char temp[50];
        char input[100];
        CString temp1;
        double x,y,z;
        CString bad = " 0123456789\t\n,";
```

```
	xyz.ReadString(input, sizeof input);
	label.Format("%s",input); label.Trim();

	xyz.ReadString(input, sizeof input);
	sscanf(input,"%d",&num_atoms);
	num_dummy = 0;

	atoms = new CAtom[num_atoms];

	for(int i=0;i<num_atoms;i++) {
		xyz.ReadString(input, sizeof input);
		sscanf(input, "%s\t%le\t%le\t%le",&temp,&x,&y,&z);
		temp1 = temp;
		atoms[i].xyz[0] = x; atoms[i].xyz[1] = y; atoms[i].xyz[2] = z;
	}

	xyz.Close();
}
```

Since structural refinement is performed using a z-matrix for the model structure, this function is of critical importance. The structure is read from a .zmx file. Dependencies are writted into the .zmx file and read within this function.

```
void CModel::ReadZMX(CString filename, int read)
{
	INT_PTR result;
	CString datext = "ZMX";
	CString datfilter ="Molecule input file (*.zmx) |*.ZMX||";

	CFileDialog datafiledialog(TRUE, datext, NULL, OFN_HIDEREADONLY
		OFN_OVERWRITEPROMPT, datfilter, NULL);
	if (read==NEWFILE) {result = datafiledialog.DoModal(); datext =
		datafiledialog.GetPathName();}
	else {datext = filename; result =IDOK;}
	if(result != IDOK) return;

	CStdioFile f;
	CFileException exception;
	BOOL status;

	status = f.Open(datext, CFile::modeRead, &exception);

	if(!status){
		char s[100];
		sprintf(s, "Error opening file for reading. Code: %d",
			exception.m_cause);
		AfxMessageBox(s, MB_OK,0);
		return;
	}
```

```
if(!empty) ClearAll();
empty = FALSE;
input_file = datext;
type = ZMX;
char input[200];

fraction = 0.0;
char temp[7][20];
f.ReadString(input,sizeof input);
label = input; label.Trim();
f.ReadString(input, sizeof input);
sscanf(input,"%d",&num_atoms);

 CString* atom_names;
atom_names = new CString[num_atoms];
atoms = new CAtom[num_atoms];

if(num_atoms==1) {
        f.ReadString(input, sizeof input);
        sscanf(input,"%s",&atom_names[0]);
        atoms[0].init(atom_names[0],0);
        num_par_zmx = 0;
        num_r = 0;
        return;
}

CString* bonded_to;
CString* bond_names;
CString* angled_to;
CString* angle_names;
CString* da_to;
CString* da_names;

if(num_atoms>1){
        bonded_to = new CString[num_atoms-1];
        bond_names = new CString[num_atoms-1];
        bonds = new CBond[num_atoms-1];
        if(num_atoms > 2) {
                angled_to = new CString[num_atoms-2];
                angle_names = new CString[num_atoms-2];
           angles = new CAngle[num_atoms-2];
                if(num_atoms >3) {
                   da_to = new CString[num_atoms-3];
                   da_names = new CString[num_atoms-3];
                   dihedrals = new CDihedral[num_atoms-3];
                }
        }
}

for(int i=0;i<num_atoms;i++){
        f.ReadString(input,sizeof input);
        if(i==0){ sscanf(input,"%s",&temp[0]); atom_names[i] = temp[0]; }
```

```
            if(i==1){
                sscanf(input,"%s %s %s", &temp[0], &temp[1], &temp[2]);
                atom_names[i] = temp[0]; bonded_to[i-1] = temp[1]; bond_names[i-1]=temp[2];
            }
            if(i==2){
                sscanf(input,"%s %s %s %s %s",&temp[0], &temp[1], &temp[2], &temp[3],
                    &temp[4]);
                atom_names[i] = temp[0]; bonded_to[i-1] = temp[1]; bond_names[i-1]=temp[2];
                angled_to[i-2]=temp[3];angle_names[i-2]=temp[4];
            }
            if(i>2){
                sscanf(input,"%s %s %s %s %s %s %s",
                    &temp[0],&temp[1],&temp[2],&temp[3],&temp[4],
                    &temp[5],&temp[6]);
                atom_names[i] = temp[0];
                bonded_to[i-1] = temp[1];
                bond_names[i-1] = temp[2];
                angled_to[i-2] = temp[3];
                angle_names[i-2] = temp[4];
                da_to[i-3] = temp[5];
                da_names[i-3] = temp[6];
            }
            atoms[i].init(atom_names[i],i);
            if(i==0) continue;
            for(int k=0;k<i;k++){
                if(bonded_to[i-1]==atoms[k].label){
                    bonds[i-1].init(atoms[i],atoms[k]);
                    bonds[i-1].label = bond_names[i-1];
                }
            }
            if(i==1) continue;
            for(k=0;k<i;k++){
                if(angled_to[i-2]==atoms[k].label){
                    angles[i-2].init(atoms[i],bonds[i-1].center2,atoms[k]);
                     angles[i-2].label = angle_names[i-2];
                }
            }
            if(i==2) continue;
            for(k=0;k<i;k++){
                if(da_to[i-3]==atoms[k].label){
                    dihedrals[i-3].init(atoms[i],bonds[i-1].center2,angles[i-2].center3,
                        atoms[k]);
                     dihedrals[i-3].label = da_names[i-3];
                }
            }
        }

        delete [] atom_names;
        if(num_atoms >1) {delete [] bonded_to; delete [] bond_names;}
        if(num_atoms >2) {delete [] angled_to; delete [] angle_names; }
        if(num_atoms >3) {delete [] da_to; delete [] da_names; }

        double parameter = 0;
```

```
CString name = ""; CString bad = " \t\n";
CString test = "ooo";

int max_par;
if(num_atoms == 2) max_par = 1;
else max_par = 3 * num_atoms - 6;

char temp_par[10];
char temp_val[50];
CString* temp1;
temp1 = new CString[max_par];
CString* temp2;
temp2 = new CString[max_par];
num_par_zmx = 0;
i = 0; int j = 0;
temp1[i] = name = "stinky";
f.ReadString(input,sizeof input);     // reads the blank line after z-matrix and before parameters

//This Loop reads the list of parameter names and their values from the bottom of the .zmx file
//Each line should contain a name and a number (e.g.  "r1    1.5") or an equation (e.g. "a2 =(360-
//          a3)/2")
while(!name.IsEmpty() && i<max_par){
          f.ReadString(input,sizeof input);
          sscanf(input,"%s %s",&temp_par,&temp_val);
          name = temp1[i] = temp_par;
          temp2[i] = temp_val;
          temp1[i].Trim(); temp2[i].Trim();
          if(test == temp1[i]) break;
          test = temp1[i];
          num_par_zmx++; i++;
}
f.Close();

param_names = new CString[num_par_zmx];
param_vals = new CString[num_par_zmx];

for(i=0;i<num_par_zmx;i++){
          param_names[i] = temp1[i];
          param_vals[i] = temp2[i];
          if(temp2[i][0] == '='){
                    for(j=0;j<num_atoms-1;j++){
                              if(temp1[i] == bonds[j].label) {
                                        bonds[j].value = CalcIt(temp2[i]);
                              }
                              if(j<num_atoms-2)
                                        if(temp1[i] == angles[j].label){
                                                  angles[j].value = CalcIt(temp2[i]);
                                        }
                              if(j<num_atoms-3)
                                        if(temp1[i] == dihedrals[j].label){
                                                  dihedrals[j].value = CalcIt(temp2[i]);
                                        }
                    }
```

```
                }
                else {
                        sscanf(temp2[i],"%lf",&parameter);
                        for(j=0;j<num_atoms-1;j++){
                                if(temp1[i] == bonds[j].label) bonds[j].value = parameter;
                                if(j<num_atoms-2)
                                        if(temp1[i] == angles[j].label){
                                                if(parameter < 0.0) parameter *= -1;
                                                if(parameter >= 360.0) parameter = parameter/360 -
                                                        int(parameter/360);
                                                if(parameter > 180.0) parameter = 360-parameter;
                                                angles[j].value = parameter;
                                        }
                                if(j<num_atoms-3)
                                        if(temp1[i] == dihedrals[j].label)
                                                dihedrals[j].value = parameter;
                        }
                }

        }
        delete [] temp1;
        delete [] temp2;
}
```

This function takes formula describing constraints that are written into the .zmx file and calculates the solutions. It's essentially a basic math reader.

```
double CModel::CalcIt(CString equation)  // max: three pieces
{
        /* supports some math operations in the zmatrix like :
                        =r1+3.0
                        =r1+r2
                        =r1+r2-1.2
                        =r1/2.0
                        =(r1+r2)/3.0
                        =2.0*(360.0-a1)
 */
        if(equation.Find("=")!=0) return 0.0;
        CString entry = equation.Right(equation.GetLength()-1);
        CString ops = "+-/*";
        CString temp = entry;
        CString first, second, third;
        double value1=0.0, value2=0.0, value3=0.0;
        char op1,op2;
        int num_pieces = 0;
        int i = 1;

        if(entry.FindOneOf("()")>=0) {
                if(entry.Find('(')==0){
                        temp = entry.Right(entry.GetLength()-1);
                        temp = temp.Left(temp.Find(')'));
```

```
                op1 = temp[temp.FindOneOf(ops)];
                first = temp.Left(temp.FindOneOf(ops));
                second = temp.Right(temp.GetLength()-first.GetLength()-1);
                temp = entry.Right(entry.GetLength()-entry.Find(')')-1);
                op2 = temp[temp.FindOneOf(ops)];
                third = temp.Right(temp.GetLength()-1);

                for(int j=0;j<num_atoms-1;j++){
                        if(first == bonds[j].label) value1 = bonds[j].value;
                        if(second == bonds[j].label) value2 = bonds[j].value;
                        if(third == bonds[j].label) value3 = bonds[j].value;
                        if(j<num_atoms-2){
                                if(first == angles[j].label) value1 = angles[j].value;
                                if(second == angles[j].label) value2 = angles[j].value;
                                if(third == angles[j].label) value3 = angles[j].value;
                        }
                        if(j<num_atoms-3){
                                if(first == dihedrals[j].label) value1 = dihedrals[j].value;
                                if(second == dihedrals[j].label) value2 = dihedrals[j].value;
                                if(third == dihedrals[j].label) value3 = dihedrals[j].value;
                        }
                }

                if(value1==0.0)   sscanf(first,"%le",&value1);
                if(value2==0.0) sscanf(second,"%le",&value2);
                if(value3==0.0) sscanf(third,"%le",&value3);

                if(op1=='+'){
                        if(op2=='+') return (value1+value2)+value3;
                        if(op2=='-') return (value1+value2)-value3;
                        if(op2=='/') return (value1+value2)/value3;
                        if(op2=='*') return (value1+value2)*value3;
                }
                if(op1=='-'){
                        if(op2=='+') return (value1-value2)+value3;
                        if(op2=='-') return (value1-value2)-value3;
                        if(op2=='/') return (value1-value2)/value3;
                        if(op2=='*') return (value1-value2)*value3;
                }
                if(op1=='/'){
                        if(op2=='+') return (value1/value2)+value3;
                        if(op2=='-') return (value1/value2)-value3;
                        if(op2=='/') return (value1/value2)/value3;
                        if(op2=='*') return (value1/value2)*value3;
                }
                if(op1=='*'){
                        if(op2=='+') return (value1*value2)+value3;
                        if(op2=='-') return (value1*value2)-value3;
                        if(op2=='/') return (value1*value2)/value3;
                        if(op2=='*') return (value1*value2)*value3;
                }
        }
        else {
```

```
temp = entry.Left(entry.Find('('));
op1 = temp[temp.FindOneOf(ops)];
first = temp.Left(temp.FindOneOf(ops));
temp = entry.Right(entry.GetLength()-entry.Find('(')-1);
temp = temp.Left(temp.Find(')'));
op2 = temp[temp.FindOneOf(ops)];
second = temp.Left(temp.FindOneOf(ops));
third = temp.Right(temp.GetLength()-second.GetLength()-1);

for(int j=0;j<num_atoms-1;j++){
        if(first == bonds[j].label) value1 = bonds[j].value;
        if(second == bonds[j].label) value2 = bonds[j].value;
        if(third == bonds[j].label) value3 = bonds[j].value;
        if(j<num_atoms-2){
                if(first == angles[j].label) value1 = angles[j].value;
                if(second == angles[j].label) value2 = angles[j].value;
                if(third == angles[j].label) value3 = angles[j].value;
        }
        if(j<num_atoms-3){
                if(first == dihedrals[j].label) value1 = dihedrals[j].value;
                if(second == dihedrals[j].label) value2 = dihedrals[j].value;
                if(third == dihedrals[j].label) value3 = dihedrals[j].value;
        }
}

if(value1==0.0)   sscanf(first,"%le",&value1);
if(value2==0.0) sscanf(second,"%le",&value2);
if(value3==0.0) sscanf(third,"%le",&value3);

if(op1=='+'){
        if(op2=='+') return value1+(value2+value3);
        if(op2=='-') return value1+(value2-value3);
        if(op2=='/') return value1+(value2/value3);
        if(op2=='*') return value1+(value2*value3);
}
if(op1=='-'){
        if(op2=='+') return value1-(value2+value3);
        if(op2=='-') return value1-(value2-value3);
        if(op2=='/') return value1-(value2/value3);
        if(op2=='*') return value1-(value2*value3);
}
if(op1=='/'){
        if(op2=='+') return value1/(value2+value3);
        if(op2=='-') return value1/(value2-value3);
        if(op2=='/') return value1/(value2/value3);
        if(op2=='*') return value1/(value2*value3);
}
if(op1=='*'){
        if(op2=='+') return value1*(value2+value3);
        if(op2=='-') return value1*(value2-value3);
        if(op2=='/') return value1*(value2/value3);
        if(op2=='*') return value1*(value2*value3);
}
```

```
		}
	}
	temp = entry; i=1;
	while(i>0){
		i = temp.FindOneOf(ops);
		temp = temp.Right(temp.GetLength()-i-1);
		num_pieces++;
	}

	first = entry.Left(entry.FindOneOf(ops));
	if(num_pieces==1) second = entry.Right(entry.GetLength()-1); //jftest 1/28/05
	if(num_pieces==2) second = entry.Right(entry.GetLength()-first.GetLength()-1);
	op1 = entry[entry.FindOneOf(ops)];

	if(num_pieces==3){
		temp = entry.Right(entry.GetLength()-entry.FindOneOf(ops)-1);
		second = temp.Left(temp.FindOneOf(ops));
		op2 = temp[temp.FindOneOf(ops)];
		third = temp.Right(temp.GetLength()-second.GetLength()-1);
	}

	for(int j=0;j<num_atoms-1;j++){
		if(first == bonds[j].label) value1 = bonds[j].value;
		if(second == bonds[j].label) value2 = bonds[j].value;
		if(num_pieces==3) if(third == bonds[j].label) value3 = bonds[j].value;
		if(j<num_atoms-2){
			if(first == angles[j].label) value1 = angles[j].value;
			if(second == angles[j].label) value2 = angles[j].value;
			if(num_pieces==3) if(third == angles[j].label) value3 = angles[j].value;
		}
		if(j<num_atoms-3){
			if(first == dihedrals[j].label) value1 = dihedrals[j].value;
			if(second == dihedrals[j].label) value2 = dihedrals[j].value;
			if(num_pieces==3) if(third == dihedrals[j].label) value3 = dihedrals[j].value;
		}
	}

	if(value1==0.0){
		sscanf(first,"%le",&value1);
	}
	if(value2==0.0){
		sscanf(second,"%le",&value2);
	}
	if(num_pieces==1 && op1=='-') return -1*value2;
	if(num_pieces==2){
		if(op1=='+') return value1+value2;
		if(op1=='-') return value1-value2;
		if(op1=='/') return value1/value2;
		if(op1=='*') return value1*value2;
	}
	if(num_pieces==3){
		if(value3==0.0){
			sscanf(third,"%le",&value3);
```

```
                }
                if(op1=='+'){
                        if(op2=='+') return value1+value2+value3;
                        if(op2=='-') return value1+value2-value3;
                        if(op2=='/') return value1+value2/value3;
                        if(op2=='*') return value1+value2*value3;
                }
                if(op1=='-'){
                        if(op2=='+') return value1-value2+value3;
                        if(op2=='-') return value1-value2-value3;
                        if(op2=='/') return value1-value2/value3;
                        if(op2=='*') return value1-value2*value3;
                }
                if(op1=='/'){
                        if(op2=='+') return value1/value2+value3;
                        if(op2=='-') return value1/value2-value3;
                        if(op2=='/') return value1/value2/value3;
                        if(op2=='*') return value1/value2*value3;
                }
                if(op1=='*'){
                        if(op2=='+') return value1*value2+value3;
                        if(op2=='-') return value1*value2-value3;
                        if(op2=='/') return value1*value2/value3;
                        if(op2=='*') return value1*value2*value3;
                }
        }
        return 0.0;
}
```

Converts the z-matrix to Cartesian coordinates.

```
void CModel::ZMXtoXYZ()         // zmx -> xyz
 {
        if(type!=ZMX || empty) return;
        const double dtor = PI/180;
        double tm = 1.0E-05;
        int i = 2, k, j;
        int* bond_atoms = new int[num_atoms];
        int* angle_atoms = new int[num_atoms];
        int* dihedral_atoms = new int[num_atoms];
        CCalc calc;

        for(int g=0;g<num_atoms-1;g++) bond_atoms[g+1] = bonds[g].center2.order;
        for(g=0;g<num_atoms-2;g++) angle_atoms[g+2] = angles[g].center3.order;
        for(g=0;g<num_atoms-3;g++) dihedral_atoms[g+3] = dihedrals[g].center4.order;

        atoms[0].xyz[0] = 0.0;    //x up/down in plane
        atoms[0].xyz[1] = 0.0;    //y out of plane
        atoms[0].xyz[2] = 0.0;    //z left/right in plane

        if(num_atoms > 1) {
                atoms[1].xyz[0] = 0.0;
                atoms[1].xyz[1] = 0.0;
                atoms[1].xyz[2] = bonds[0].value;
```

```
}

if(num_atoms > 2){
        atoms[2].xyz[0] = bonds[1].value * sin(dtor*angles[0].value);
        atoms[2].xyz[1] = 0.0;
        if(bonds[1].center2 == atoms[0]) atoms[2].xyz[2] = bonds[1].value *
                cos(dtor*angles[0].value);
        if(bonds[1].center2 == atoms[1]) atoms[2].xyz[2] = atoms[1].xyz[2] - bonds[1].value *
                cos(dtor*angles[0].value);

        if(fabs(atoms[2].xyz[0]) < tm){
                for(i=3;i<num_atoms;i++){
                        if(i+1>num_atoms||fabs(atoms[i-1].xyz[0])>=tm) break;
                        atoms[i].xyz[0] = bonds[i-1].value*sin(dtor*angles[i-2].value);
                        atoms[i].xyz[1] = 0.0;
                        atoms[i].xyz[2] = atoms[bond_atoms[i]].xyz[2] - bonds[i-1].value *
                                cos(dtor*angles[i-2].value) *
                                calc.dsign(1.0,atoms[bond_atoms[i]].xyz[2]-
                                atoms[angle_atoms[i]].xyz[2]);
                }
        }

        k = i;
        if(i==2) k=3;
        for(j=k;j<num_atoms;j++){
                // v1 is the vector from zmx.dihedrals[x].center4 to zmx.dihedrals[x].center3
                double v1[3],v2[3],vp[3],vj[3],al1[3],al2[3],al3[3],al4[3];
                // al1 is the unit vector of v1
                for (int f=0; f<3;f++) v1[f]=atoms[angle_atoms[j]].xyz[f]-
                        atoms[dihedral_atoms[j]].xyz[f]; calc.univec(&al1,v1);
                // v1 is the vector from zmx.dihedrals[x].center2 to zmx.dihedrals[x].center1
                for (f=0; f<3;f++) v2[f]=atoms[bond_atoms[j]].xyz[f]-
                        atoms[angle_atoms[j]].xyz[f];
                calc.univec(&al2,v2);    // al2 is the unit vector of v2
                calc.vecprd(&vp,al1,al2);
                for(i=0;i<3;i++) al3[i] = vp[i]/sqrt(1.0-
                        (al1[0]*al2[0]+al1[1]*al2[1]+al1[2]*al2[2])*
                        (al1[0]*al2[0]+al1[1]*al2[1]+al1[2]*al2[2]));
                calc.vecprd(&al4,al3,al2);

                for(i=0;i<3;i++){
                        vj[i]=bonds[j-k+2].value*(-al2[i]*cos(dtor*angles[j-k+1].value)+
                        al4[i]*sin(dtor*angles[j-k+1].value)*cos(dtor*dihedrals[j-k].value)+
                        al3[i]*sin(dtor*angles[j-k+1].value)*sin(dtor*dihedrals[j-k].value));
                        atoms[j].xyz[i] = vj[i] + atoms[bond_atoms[j]].xyz[i];
                        if(fabs(atoms[j].xyz[i])<tm) atoms[j].xyz[i] = 0.0;
                }
        }
}
delete [] bond_atoms;
delete [] angle_atoms;
delete [] dihedral_atoms;
```

```
        num_dummy = 0;                  // counts dummy atoms
        CAtom temp;
        int temp1 = num_atoms;
        for(i=0;i<temp1;i++){
                if(atoms[i].name == ELEMENTS[MAX_ELEMENTS-1]){
                        num_dummy++;
                }
        }
}
```

This function converts the Cartesian coordinate set to a list of internuclear distances accompanied by their mean amplitude of vibration and anharmonicity – this is the input for the calculation of the theoretical *sM*(*s*).

```
void CModel::XYZtoMLS()                                         //xyz -> mls
{
        if(empty || (type != ZMX && type != XYZ)) return;
        int j = num_atoms - num_dummy;;

        if(num_atoms == 1) {num_r=0; return;}

        int num_dists = (j * (j - 1)) / 2;              //get internuclear distances
        CDistance* ij = new CDistance[num_dists];

        int k = 0;
        for(int i=0;i<num_atoms;i++){
                if(atoms[i].name == ELEMENTS[MAX_ELEMENTS-1]) continue; //skip dummy atoms
                for(j=i+1;j<num_atoms;j++){
                        if(atoms[j].name == ELEMENTS[MAX_ELEMENTS-1]) continue;
                        ij[k].r = sqrt((atoms[i].xyz[0]-atoms[j].xyz[0])*(atoms[i].xyz[0]-
                                atoms[j].xyz[0])+(atoms[i].xyz[1]-atoms[j].xyz[1])*(atoms[i].xyz[1]-
                                atoms[j].xyz[1])+(atoms[i].xyz[2]-atoms[j].xyz[2])*(atoms[i].xyz[2]-
                                atoms[j].xyz[2]));
                        ij[k].pair[0] = atoms[i]; ij[k].pair[1] = atoms[j];
                        k++;
                }
        }

        CDistance* temp = new CDistance[num_dists];
        double tl = 1000.0; int q = 0;                  //put distances in numerical order
        for(i=0;i<num_dists;i++){
                for(j=0;j<num_dists;j++){
                        if(ij[j].r < tl){tl = ij[j].r; q = j;}
                }
                if(ij[q].r < 1000.0) temp[i] = ij[q];
                ij[q].r = 1000.0; tl = 1000.0;
        }
        delete [] ij;
```

```
        j=0; k=0;
        for(i=1;i<num_dists;i++){           //count number of unique internuclear distances
                if(temp[j] == temp[i]) k++;
                else j = i;
        }
        num_r = num_dists - k;

        if(mls_flag) delete [] distance;
        distance = new CDistance[num_r];

        i=0;j=0;
        distance[j] = temp[j];
        for(i=1;i<num_dists;i++){           //sort degenerate distances into final array
                if(distance[j] == temp[i]) distance[j].multi++;
                else{
                        j++;
                        distance[j] = temp[i];
                }
        }

        delete [] temp;
        SetTemperature(temperature);
        Calculate_l();
        mls_flag = TRUE;
        sMsSingle = new double[NUM_PIX];
}
```

Sets the molecular temperature.

```
void CModel::SetTemperature(double new_temp)
{
        if(new_temp < 5) new_temp = 5;
        temperature = new_temp;
   for(int i=0;i<num_r;i++) distance[i].temperature = new_temp;
}
```

Calculates the mean amplitudes of vibration at the assigned molecular temperature and sets the anharmonicity constant. Solution for the vibrational frequency requires functions of the CCalc object which are displayed in Appendix V.

```
void CModel::Calculate_l()     //mean amplitude of vibration and anharmonicity
{
        double mu = 0.0, ampsqr = 0.0, f, temp_ratio;
        CCalc calc;

        for(int i=0;i<num_r;i++){
                distance[i].k = 0.0; distance[i].dr = 0.0;
                distance[i].l = 0.0; distance[i].a = 0.0;
```

```
                mu = (distance[i].pair[0].atomic_mass *
                        distance[i].pair[1].atomic_mass)/(distance[i].pair[0].atomic_mass +
                        distance[i].pair[1].atomic_mass)*1.2;
                if(mu==0.0) continue;
                if(distance[i].pair[0].atomic_number == 6||distance[i].pair[1].atomic_number==6){
                        if(distance[i].pair[0].atomic_number==6 &&
                                distance[i].pair[1].atomic_number==6){
                                if(distance[i].r < 1.8) distance[i].l = 0.01384+0.0234*distance[i].r-
                                        0.00015*distance[i].r*distance[i].r;
                                else distance[i].l = 0.013837 + 0.023398 * distance[i].r - 0.000147 *
                                        distance[i].r * distance[i].r;
                        }
                        else if(distance[i].pair[0].atomic_number == 1 ||
                                distance[i].pair[1].atomic_number==1)
                                distance[i].l = 0.050134+0.02738*distance[i].r-
                                0.001805*distance[i].r*distance[i].r;
                        else distance[i].l = 0.013837+0.023398*distance[i].r-
                                0.000147*distance[i].r*distance[i].r;
                }
                else if(distance[i].pair[0].atomic_number == 1||distance[i].pair[1].atomic_number==1)
                        distance[i].l = 0.050134+0.02738*distance[i].r-
                                0.001805*distance[i].r*distance[i].r;
                else distance[i].l = 0.013837+0.023398*distance[i].r-0.000147*distance[i].r*distance[i].r;
                ampsqr = distance[i].l * distance[i].l;
                f = calc.FindRoot(mu,ampsqr);
                temp_ratio = (cosh(23.98*f/distance[i].temperature))*(sinh(23.98*f/300))/
                        (sinh(23.98*f/distance[i].temperature))/(cosh(23.98*f/300));
                distance[i].l *= sqrt(temp_ratio);

                if(distance[i].r > 1.85*(1.0+sqrt(distance[i].pair[0].atomic_number*
                        distance[i].pair[1].atomic_number/36.0)*0.084)) distance[i].a = 0.0;
                else if(distance[i].temperature > 600) distance[i].a = 2.0;
                else distance[i].a = 2.0;
                distance[i].lm = distance[i].l*distance[i].l+1.5*distance[i].a*distance[i].a*distance[i].l*
                        distance[i].l*distance[i].l*distance[i].l;
                distance[i].dr=1.5*distance[i].a*distance[i].lm*distance[i].lm;
                distance[i].k=distance[i].l*distance[i].l*distance[i].l*distance[i].l/6;
        }
}
```

Returns the number of fittable parameters in the molecule.

```
int CModel::GetNumParams()
{
        int temp_num_params = 0;
        if(type == MLS){
                temp_num_params += num_r;               // for the r values
                temp_num_params += num_r;               // for the l values
        }
        else if(type == XYZ){
                temp_num_params += num_r;               // for the l values
                temp_num_params++;                      // for total T fitting
        }
        else if(type == ZMX){
```

```
                temp_num_params += num_par_zmx;                 // parameters in the z-matrix
                temp_num_params++;                                      // for total T fitting
        }
        num_params = temp_num_params;
        return temp_num_params;
}
```

Stores in names of the fittable parameters and their values in the input arrays.

```
void CModel::GetParams(CString* par_names, CString* par_vals)
{
        int j;
        int k=0;

        if(type == MLS){
                for(j=0;j<num_r;j++){
                        par_names[k] = "r" + distance[j].pair[0].label + distance[j].pair[1].label;
                        par_vals[k].Format("%.10lf",distance[j].r);
                        k++;
                        par_names[k] = "l" + distance[j].pair[0].label + distance[j].pair[1].label;
                        par_vals[k].Format("%.10lf",distance[j].l);
                        k++;
                }
        }
        else if(type == XYZ){
                for(j=0;j<num_r;j++){
                        par_names[k] = "l" + distance[j].pair[0].label + distance[j].pair[1].label;
                        par_vals[k].Format("%.10lf",distance[j].l);
                        k++;
                }
                par_names[k] = "Temperature";
                par_vals[k].Format("%.10lf",temperature);
                k++;
        }
        else if(type == ZMX){
                for(j=0;j<num_par_zmx;j++){
                        par_names[k] = param_names[j];
                        par_vals[k] = param_vals[j];
                        k++;
                }
                par_names[k] = "Temperature";
                par_vals[k].Format("%.10lf",temperature);
                k++;
        }
}
```

Updates the parameters after fitting.

```
void CModel::UpdateParams(CString* par_names, CString* par_vals)
{
        int i,j;
        int k=0;
        double parameter;
```

```
if(type == MLS){
        for(j=0;j<num_r;j++){
                sscanf(par_vals[k],"%lf",&distance[j].r);
                k++;
                sscanf(par_vals[k],"%lf",&distance[j].l);
                k++;
        }
}
else if(type == XYZ){
        for(j=0;j<num_r;j++){
                sscanf(par_vals[k],"%lf",&distance[j].l);
                k++;
        }
        sscanf(par_vals[k],"%lf",&temperature);
        k++;
        XYZtoMLS();
}
else if(type == ZMX){
        for(j=0;j<num_par_zmx;j++){
                param_names[j] = par_names[k];
                param_vals[j] = par_vals[k];
                k++;
        }
        for(i=0;i<num_par_zmx;i++){
                if(param_vals[i][0] == '='){
                        for(j=0;j<num_atoms-1;j++){
                                if(param_names[i] == bonds[j].label) {
                                        bonds[j].value = CalcIt(param_vals[i]);
                                }
                                if(j<num_atoms-2)
                                        if(param_names[i] == angles[j].label){
                                                angles[j].value = CalcIt(param_vals[i]);
                                        }
                                if(j<num_atoms-3)
                                        if(param_names[i] == dihedrals[j].label){
                                                dihedrals[j].value = CalcIt(param_vals[i]);
                                        }
                        }
                }
                else{
                        sscanf(param_vals[i],"%lf",&parameter);
                        for(j=0;j<num_atoms-1;j++){
                                if(param_names[i] == bonds[j].label)
                                        bonds[j].value = parameter;
                                if(j<num_atoms-2)
                                        if(param_names[i] == angles[j].label){
                                                if(parameter < 0.0) parameter *= -1;
                                                if(parameter >= 360.0) parameter =
                                                        parameter/360 - int(parameter/360);
                                                if(parameter > 180.0)
                                                        parameter = 360-parameter;
                                                angles[j].value = parameter;
                                        }
```

```
                                        if(j<num_atoms-3)
                                                if(param_names[i] == dihedrals[j].label)
                                                        dihedrals[j].value = parameter;
                                }
                        }
                }
                sscanf(par_vals[k],"%lf",&temperature);
                k++;
                ZMXtoXYZ();
                XYZtoMLS();
        }
}
```

Appendix V. The calculation of the curves in the UED_2004 program: "Calc.cpp"

The Calc.cpp code file contains all the mathematical machinery (other than the refinement code). The following functions are the important aspects of the calculation of the theoretical and experimental $sM(s)$ and a few other odds and ends. The intention was to increase readability of the code by separating related functions into separate classes and source files. See the comments below for descriptions of individual functions.

Calc_s calculates the s value at each pixel given the necessary pieces of information (See Chapter 4). The array s[] contains each s value and the array delta_s[] contains the s step at each pixel needed for calculation of $f(r)$.

```
void CCalc::Calc_s(double pix_size, double cam_dist, double wave_constant, double *s, double* delta_s)
{
        double width;
        double half_width1, half_width2;
        for(int i=0; i<NUM_PIX; i++){
                width = i * 0.0001 * pix_size / cam_dist;                                       // tan(theta)
                s[i] = wave_constant * sin(atan(width)/2);

                half_width1 = (i + 0.5) * 0.0001 * pix_size / cam_dist;
                half_width2 = (i - 0.5) * 0.0001 * pix_size / cam_dist;
                delta_s[i] = wave_constant * sin(atan(half_width1)/2) - wave_constant *
                        sin(atan(half_width2)/2);
        }
```

```
}
```

Returns 4pi/lamda for the calculation of *s*.

```
double CCalc::WaveConstant(double electron_ke)
{
        return sqrt((1022.0 + electron_ke) * electron_ke) / 12.398521 * 4 * PI;   // 4pi/lamda
}
```

Returns scattering factor interpolated to value sprime.

```
double CCalc::InterpolateScattFactor(double s1, double sprime, double scatt1, double scatt2)
{
        double scale = fabs(s1 - sprime)/SCAT_FACTOR_STEP_SIZE;
        return scale * (scatt2 - scatt1) + scatt1;
}

 double CCalc::dsign(double X1, double X2)
{
        if (X2 >= 0.0) return fabs(X1); else return -1.0 * fabs(X1);
}
```

Given vector R, this function returns its unit vector AL.

```
void CCalc::univec(double (*AL)[3], double R[3])
{
    double FAC = 1.0/sqrt(R[0]*R[0] + R[1]*R[1] + R[2]*R[2]);
    for (int I=0;I<3;I++) (*AL)[I] = R[I] * FAC;
}
```

Returns the vector product of X[] and Y[].

```
void CCalc::vecprd(double (*VP)[3],double X[3], double Y[3])
{
    (*VP)[0]=X[1]*Y[2]-X[2]*Y[1];
    (*VP)[1]=X[2]*Y[0]-X[0]*Y[2];
    (*VP)[2]=X[0]*Y[1]-X[1]*Y[0];
}
```

The following functions: FindRoot, zbrak, zbrent, and fx find the value of *v*, the harmonic vibrational frequency for an internuclear pair. See Section 4.5.2 on mean amplitudes of vibration. The frequency is assumed to be independent of temperature and thusly allows the mean amplitude to be scaled to the experimental temperature. These functions can be found on pp. 350 – 362 in Numerical Recipes in C – see reference in Chapter 4 (Press et al).

```
double CCalc::FindRoot(double mu, double ampsqr)
{
```

```
        int N = 100;
        const int NBMAX = 1;
        double X1 = 0.1;
        double X2 = 2000.0;
        int i,nb = NBMAX;
        double tol,root;
        double *xb1 = new double [NBMAX+1];
        double *xb2 = new double [NBMAX+1];
        zbrak(X1,X2,N,xb1,xb2,&nb,mu,ampsqr);
        for (i=1;i<=nb;i++) {
                tol=(1.0e-6)*(xb1[i]+xb2[i])/2.0;
                root=zbrent(xb1[i],xb2[i],tol,mu,ampsqr);
        }
        delete [] xb1;
        delete [] xb2;
        return root;
}

void CCalc::zbrak(double x1, double x2, int n, double xb1[], double xb2[], int *nb,double mu, double
ampsqr)
{
   int nbb,i;
   double x,fp,fc,dx;
        nbb=0;
   dx=(x2-x1)/n;
   fp=fx(x=x1,mu,ampsqr);
   for (i=1;i<=n;i++) {
      fc=fx(x += dx,mu,ampsqr);
      if (fc*fp <= 0.0) {
                xb1[++nbb]=x-dx;
                xb2[nbb]=x;
                if(*nb == nbb) return;
            }
      fp=fc;
   }
   *nb = nbb;
}

double CCalc::zbrent(double x1, double x2, double tol,double mu,double ampsqr)
{
   int ITMAX=100;
   double EPS=3.0e-8;
   int iter;
   double a=x1,b=x2,c=x2,d,e,min1,min2;
   double fa=fx(a,mu,ampsqr),fb=fx(b,mu,ampsqr),fc,p,q,r,s,tol1,xm;

   if ((fa > 0.0 && fb > 0.0) || (fa < 0.0 && fb < 0.0))
                AfxMessageBox("Root must be bracketed in zbrent.",MB_OK,0);
   fc=fb;
   for (iter=1;iter<=ITMAX;iter++) {
      if ((fb > 0.0 && fc > 0.0) || (fb < 0.0 && fc < 0.0)) {
                c=a; fc=fa; e=d=b-a;
      }
```

```
	if (fabs(fc) < fabs(fb)) {
		a=b; b=c; c=a; fa=fb; fb=fc; fc=fa;
	}
	tol1=2.0*EPS*fabs(b)+0.5*tol;
	xm=0.5*(c-b);
	if (fabs(xm) <= tol1 || fb == 0.0) return b;
	if (fabs(e) >= tol1 && fabs(fa) > fabs(fb)) {
		s=fb/fa;
		if (a == c) {
			p=2.0*xm*s;
			q=1.0-s;
		}
		else {
			q=fa/fc;
			r=fb/fc;
			p=s*(2.0*xm*q*(q-r)-(b-a)*(r-1.0));
			q=(q-1.0)*(r-1.0)*(s-1.0);
		}
		if (p > 0.0) q = -q;
		p=fabs(p);
		min1=3.0*xm*q-fabs(tol1*q);
		min2=fabs(e*q);
		if (2.0*p < (min1 < min2 ? min1 : min2)) {
				e=d; d=p/q;
		}
		else d=xm; e=d;
	}
	else d=xm; e=d;
	a=b; fa=fb;
	if (fabs(d) > tol1) b += d;
	else b += (xm >= 0.0 ? fabs(tol1) : -fabs(tol1));
	fb=fx(b,mu,ampsqr);
  }
  AfxMessageBox("Maximum number of iterations exceeded in zbrent.",MB_OK,0);
  return 0.0;
}

double CCalc::fx(double x,double mu,double ampsqr)
{
	return 0.5047/mu/x*cosh(23.98*x/300)/sinh(23.98*x/300)-ampsqr;
}
```

This function calculates the theoretical molecular scattering intensity.

```
void CCalc::CalcTheoryIMs(CUED_2004Doc* pDoc, double* IMstemp, int which)
{
	int fx1, fx2, i, j;
	double ra;
	double single;

	for(fx1=0;fx1<NUM_PIX;fx1++){
		IMstemp[fx1] = 0.0;
		for (fx2 = 0; fx2 < pDoc->nulecule[which].num_r; fx2++) {
			for(int k=0;k<pDoc->GetNumElements();k++){
```

```
                                    if(pDoc->nulecule[which].distance[fx2].pair[0].atomic_number ==
                                            pDoc->GetElement(k)) i = k;
                                    if(pDoc->nulecule[which].distance[fx2].pair[1].atomic_number ==
                                            pDoc->GetElement(k)) j = k;
                            }
                            ra = pDoc->nulecule[which].distance[fx2].r;
                            single = pDoc->GetfFactor(i,fx1) * pDoc->GetfFactor(j,fx1);
                            single *= 2.0 * pDoc->nulecule[which].distance[fx2].multi;
                            ra -= pDoc->nulecule[which].distance[fx2].l *
                                    pDoc->nulecule[which].distance[fx2].l  /
                                    pDoc->nulecule[which].distance[fx2].r;
                            ra += 1.5 * pDoc->nulecule[which].distance[fx2].a *
                                    pDoc->nulecule[which].distance[fx2].l *
                                    pDoc->nulecule[which].distance[fx2].l;
                            single *= sin(pDoc->Get_s(fx1) * ra) / (pDoc->Get_s(fx1) * ra);
                            single *= cos(pDoc->GetnFactor(i,fx1) - pDoc->GetnFactor(j,fx1));
                            single *= exp(-pDoc->Get_s(fx1) * pDoc->Get_s(fx1) *
                                    pDoc->nulecule[which].distance[fx2].l *
                                    pDoc->nulecule[which].distance[fx2].l / 2);
                            if(pDoc->Get_s(fx1) == 0.0 || ra == 0.0 || pDoc->nulecule[which].distance[fx2].r
                                    == 0.0) single = 0.0;
                            IMstemp[fx1] += single;
                    }
            }
}
```

The following function smooths the scattering intensity with respect to the width of the electron beam. Wider electron beams cause washing out of the signal which UED theory attempts to model.

```
void CCalc::BoxcarSmooth(int num_to_smo, double* smoo_temp)
{
        int s, s1, s2;
        double smoothed_point;

        int bs_start = 0;
        int bs_range = NUM_PIX;

        if(num_to_smo == 2 * (num_to_smo / 2))  num_to_smo++;

        if(num_to_smo > 1) {
                num_to_smo = (num_to_smo - 1) / 2;

                for (s = bs_start; s < bs_range; s++) {
                        if(s < (num_to_smo + bs_start))
                                s1 = s - bs_start;
                        else if((bs_range - 1 - s) < num_to_smo)
                                s1 = bs_range - 1 - s;
                        else
                                s1 = num_to_smo;
```

```
                        smoothed_point = 0.0;
                        for (s2 = -s1; s2 <= s1; s2++)
                                smoothed_point += smoo_temp[s + s2];
                        smoothed_point /= (2 * s1 + 1);

                        smoo_temp[s] = smoothed_point;

                }
        }
}
```

The calculation of χ^2.

```
double CCalc::ChiSqr(double* theory, double* experiment, double* weight, int range_min, int range_max)
{
        double chisqr = 0.0;

        for(int i=range_min;i<range_max; i++)
                chisqr += (theory[i] - experiment[i]) * (theory[i] - experiment[i]) * weight[i];

        return chisqr;
}
```

The calculation of *R*.

```
double CCalc::R(double* theory, double* experiment, double* weight, int range_min, int range_max)
{
        double R;
        double chisqr = 0.0;
        double Tsqr = 0.0;

        for(int i=range_min;i<range_max; i++){
                chisqr += (theory[i] - experiment[i]) * (theory[i] - experiment[i]) * weight[i];
                Tsqr += theory[i] * theory[i] * weight[i];
        }

        R = sqrt(chisqr/Tsqr);
        return R;
}
```

The theoretical total atomic scattering is calculated in this function.

```
void CCalc::CalcAtomic(CUED_2004Doc* pDoc, double* atomic)
{
        int a=0;
        int i;
        double temp;
        for(i=0;i<NUM_PIX;i++) atomic[i] = 0.0;

        for(i=0;i<pDoc->GetNumModels();i++){
                for(int j=0;j<pDoc->nulecule[i].num_atoms;j++){
                        if(pDoc->nulecule[i].atoms[j].name == ELEMENTS[MAX_ELEMENTS-1])
                                continue;
                        for(int k=0;k<pDoc->GetNumElements();k++){
```

```
                                        if(pDoc->nulecule[i].atoms[j].atomic_number ==
                                                pDoc->GetElement(k)){a = k; break;}
                                }
                                for(int l=0;l<NUM_PIX;l++){
                                        temp = pDoc->GetfFactor(a,l) * pDoc->GetfFactor(a,l);
                                        if(pDoc->IncludeInelastic() && pDoc->Get_s(l) != 0.0)
                                                temp += 4*pDoc->GetSFactor(a,l)/(BOHR*BOHR*
                                                pDoc->Get_s(l)*pDoc->Get_s(l)*pDoc->Get_s(l)*
                                                pDoc->Get_s(l));
                                        atomic[l] += pDoc->nulecule[i].fraction * temp;
                                }
                        }
                }
}
```

This function calculates the atomic scattering of a single atom. This is usually used when calculating the atomic scattering for the reference gas (e.g. Xenon).

```
void CCalc::CalcSingleAtomic(CUED_2004Doc* pDoc, int atom, double* atomic)
{
        for(int i=0;i<NUM_PIX;i++){
                atomic[i] = pDoc->GetfFactor(atom,i) * pDoc->GetfFactor(atom,i);
                if(pDoc->IncludeInelastic() && pDoc->Get_s(i) != 0.0)
                        atomic[i] += 4*pDoc->GetSFactor(atom,i)/(BOHR*BOHR*
                                pDoc->Get_s(i)*pDoc->Get_s(i)*pDoc->Get_s(i)*pDoc->Get_s(i));
        }
}
```

This function prepares the final theoretical $sM(s)$ by first adding together the $I_M^t(s)$ curves for each component molecule weighted by their fractional contribution. Then the total $I_M^t(s)$ is leveled (a choice of three methods, although only the first is used) to form the $sM(s)$.

```
void CCalc::PrepareTheorysMs(CUED_2004Doc* pDoc, double* sMs)
{
        double* atomic;
        int i,j;

        for(i=0;i<NUM_PIX;i++){
                sMs[i] = 0.0;
                for(j=0;j<pDoc->GetNumModels();j++){
                        if(pDoc->nulecule[j].sms_flag){
                                sMs[i] += pDoc->nulecule[j].fraction *
                                        pDoc->nulecule[j].sMsSingle[i];
                        }
```

```
                        if(pDoc->ProductOnly_flag && pDoc->nulecule[j].po_flag){
                                sMs[i] += -1 * pDoc->nulecule[j].fraction *
                                        pDoc->nulecule[j].sMsSingle[i];
                        }
                }
        }
        if(pDoc->sms_mode == SMS_SFIFJ){
                double fifj;
                for(i=0;i<NUM_PIX;i++){
                        sMs[i] *= pDoc->Get_s(i);
                        fifj = pDoc->GetfFactor(pDoc->GetScaleAtom1(),i) *
                                pDoc->GetfFactor(pDoc->GetScaleAtom2(),i);
                        if(fifj != 0.0) sMs[i] /= fifj;
                        else sMs[i] = 0.0;
                }
        }
        else if(pDoc->sms_mode == SMS_SATOMIC){
                atomic = new double[NUM_PIX];
                for(i=0;i<NUM_PIX;i++) atomic[i] = 0.0;
                CalcAtomic(pDoc, atomic);
                for(i=0;i<NUM_PIX;i++) if(atomic[i] != 0.0) sMs[i] *= (pDoc->Get_s(i) / atomic[i]);
                delete [] atomic;
        }
        else if(pDoc->sms_mode == SMS_S5){
                for(i=0;i<NUM_PIX;i++) if(pDoc->Get_s(i)!=0.0) sMs[i] /= (pDoc->Get_s(i) *
                        pDoc->Get_s(i) * pDoc->Get_s(i) * pDoc->Get_s(i) * pDoc->Get_s(i));
        }
}
```

Raw data enters the analysis program as $R^N(pix)$ [see Eq. (4.9)]. Section 4.3.2 explains how the following function extracts from the input data the experimental $sM^e(s)$.

```
void CCalc::PrepareData(CUED_2004Doc* pDoc, double* out_data, double* out_sigma)
{
// muliplty raw_data by IXeT and scale factor then subtract IAT and background, then multiply by (s/fifj)
        double* IXeT;
        IXeT = new double[NUM_PIX];
        double* ItotAT;
        ItotAT = new double[NUM_PIX];
        double* total_bkg;
        total_bkg = new double[NUM_PIX];
        int i,j;

        CalcPolyBkg(pDoc, total_bkg);
        CalcAtomic(pDoc, ItotAT);
        CalcSingleAtomic(pDoc, pDoc->GetAtomicRef(), IXeT);
        for(i=0;i<NUM_PIX;i++){
                if(pDoc->UseAtomicRef()){
                        out_data[i] *= IXeT[i];
                        out_sigma[i] *= IXeT[i];
```

```
                }
                out_data[i] *= pDoc->GetScaleFactor();
                out_sigma[i] *= pDoc->GetScaleFactor();
                out_data[i] -= (pDoc->GetAtomicScale() * ItotAT[i]);
                out_data[i] -= total_bkg[i];

                if(pDoc->ProductOnly_flag){
                        for(j=0;j<pDoc->GetNumModels();j++){
                                if(pDoc->nulecule[j].po_flag){
                                        out_data[i] += -1*pDoc->nulecule[j].fraction*
                                                pDoc->nulecule[j].sMsSingle[i];
                                }
                        }
                }

                if(pDoc->sms_mode == SMS_SFIFJ){
                        out_data[i] *= pDoc->Get_s(i);
                        out_sigma[i] *= pDoc->Get_s(i);
                        out_data[i] /= (pDoc->GetfFactor(pDoc->GetScaleAtom1(),i) *
                                pDoc->GetfFactor(pDoc->GetScaleAtom2(),i));
                        out_sigma[i] /= (pDoc->GetfFactor(pDoc->GetScaleAtom1(),i) *
                                pDoc->GetfFactor(pDoc->GetScaleAtom2(),i));
                }
                else if(pDoc->sms_mode == SMS_SATOMIC){
                        out_data[i] *= (pDoc->Get_s(i) / ItotAT[i]);
                        out_sigma[i] *= (pDoc->Get_s(i) / ItotAT[i]);
                }
                else if(pDoc->sms_mode == SMS_S5){
                        out_data[i] *= (pDoc->Get_s(i) * pDoc->Get_s(i) * pDoc->Get_s(i) *
                                pDoc->Get_s(i) * pDoc->Get_s(i));
                        out_sigma[i] *= (pDoc->Get_s(i) * pDoc->Get_s(i) * pDoc->Get_s(i) *
                                pDoc->Get_s(i) * pDoc->Get_s(i));
                }
        }
        delete [] IXeT;
        delete [] ItotAT;
        delete [] total_bkg;
}
```

Calculated the inverse of matrix[][] and stores the result in inverted[][].

```
void CCalc::InvertMatrix(double** matrix, double** inverted, int dimension)
{
        double** M;
        int i, j, k;
        double s;

        M = new double *[dimension];
        for(i=0;i<dimension;i++) M[i] = new double[dimension];

   for(i = 0; i < dimension; i++){
                for(j = 0; j < dimension; j++){
                        inverted[i][j] = (i == j) ? 1.0 : 0.0;
                }
```

```
        }
        for(i = 0; i < dimension; i++){
                for(j = 0; j < dimension; j++){
                        M[i][j] = matrix[i][j];
                }
        }

        for(i = 0; i < dimension; i++){
                if(M[i][i] == 0.0){
                        for(j = i+1; j < dimension; j++){
                                if(M[j][i] != 0.0f){
                                        SwapMatrixRow(M, i, j, dimension);
                                        SwapMatrixRow(inverted, i, j, dimension);
                                        break;
                                }
                        }
                        if(j == dimension){
                                AfxMessageBox("Singular trans mat encountered.");
                                return;
                        }
                }
                if(M[i][i] != 1.0){
                        s = 1.0 / M[i][i];
                        ScaleMatrixRow(M, i, s, dimension);
                        ScaleMatrixRow(inverted, i, s, dimension);
                }
                for(j = i+1; j < dimension; j++){
                        for(k = i+1; k < dimension; k++){
                                M[j][k] -= M[j][i] * M[i][k];
                        }
                        for(k = 0; k < dimension; k++){
                                inverted[j][k] -= M[j][i] * inverted[i][k];
                        }
                }
        }
        for(i = dimension-1; i >= 0; i--){
                for(j = 0; j < i; j++){
                        for(k = 0; k < dimension; k++){
                                inverted[j][k] -= M[j][i] * inverted[i][k];
                        }
                }
        }
        for(i=0;i<dimension;i++) delete [] M[i];
        delete [] M;
}
```

Swaps rows i and j in matrix m.

```
void CCalc::SwapMatrixRow(double** m, int i, int j, int dimension)
{
  double tmp;
  for(int k = 0; k < dimension; k++){
    tmp = m[i][k];
    m[i][k] = m[j][k];
```

```
   m[j][k] = tmp;
  }
}
```

Multiplies all values in row i of matrix m by s.

```
void CCalc::ScaleMatrixRow(double** m, int i, double s, int dimension)
{
  int k;
  for(k = 0; k < dimension; k++){
    m[i][k] *= s;
  }
}
```

Returns the determinant of matrix.

```
double CCalc::Determinant(double** matrix, int dimension)
{
        double top_prod, bot_prod;
        double top_sum, bot_sum;
        double determ = 0;
        int i,j,k;
        double *column;

        column = new double[dimension];

        top_sum = 0.0; bot_sum = 0.0;
        for(i=0;i<dimension;i++){
                top_prod = 1.0; bot_prod = 1.0;
                for(j=0;j<dimension;j++){
                        top_prod *= matrix[j][j];
                        bot_prod *= matrix[j][dimension - j];
                        column[j] = matrix[0][j];
                }
                top_sum += top_prod;
                bot_sum += bot_prod;
                for(j=0;j<dimension-1;j++){
                        for(k=0;k<dimension;k++){
                                matrix[j][k] = matrix[j+1][k];
                        }
                }
                for(k=0;k<dimension;k++) matrix[dimension-1][k] = column[k];
        }

        determ = top_sum + bot_sum;

        delete [] column;

        return determ;
}
```

Calculates the polynomial background.

```
void CCalc::CalcPolyBkg(CUED_2004Doc* pDoc, double* total_bkg)
{
```

```
        double bkg[NUM_PIX];
        int i;

        for(i=0;i<NUM_PIX;i++) total_bkg[i] = 0.0;

        for(i=0;i<pDoc->GetPolyBkgOrder();i++){
                InitPolyBkg(pDoc, bkg);
                for(int j=0;j<NUM_PIX;j++){
                        bkg[j] *= pDoc->GetPolyBkg(i);
                        for(int k=0;k<i;k++){
                                bkg[j] *= pDoc->Get_s(j);
                        }
                        total_bkg[j] += bkg[j];
                }
        }
}
```

Initializes the polynomial background. The first bkg_mode is used since the raw data is multiplied by the theoretical atomic scattering of xenon.

```
void CCalc::InitPolyBkg(CUED_2004Doc* pDoc, double* polybkg_term)
{
        double bkg = 1.0;
        int i;
        double IXeT[NUM_PIX];

        CalcSingleAtomic(pDoc, pDoc->GetAtomicRef(), IXeT);

        for(i=0;i<NUM_PIX;i++) polybkg_term[i] = 0.0;
        if(pDoc->bkg_mode == BKG_IREFT) for(i=0;i<NUM_PIX;i++) polybkg_term[i] = IXeT[i];
        if(pDoc->bkg_mode == BKG_ONE) for(i=0;i<NUM_PIX;i++) polybkg_term[i] = 1.0;
        if(pDoc->bkg_mode == BKG_1OVERS) for(i=0;i<NUM_PIX;i++)
                polybkg_term[i] = 1/pDoc->Get_s(i);
        if(pDoc->bkg_mode == BKG_1OVERS2) for(i=0;i<NUM_PIX;i++)
                 polybkg_term[i] = 1/(pDoc->Get_s(i)*pDoc->Get_s(i));
}
```

This is the approximation of sine transform of $sM(s)$.

```
void CCalc::Calculate_fr(CUED_2004Doc* pDoc, double* sMs, double* fr)
{
        int i,j;
        double r_factor;
        double range = pDoc->GetfrRangeMax() - pDoc->GetfrRangeMin();

        for(i=0;i<NUM_PIX;i++){
                fr[i] = 0.0;
                r_factor = i * range/(NUM_PIX-1) + pDoc->GetfrRangeMin();
                for(j=0;j<pDoc->GetRangeMax();j++){
                        fr[i] += sMs[j] * sin(pDoc->Get_s(j)*r_factor) * exp(-1 *
                                pDoc->GetDampConst() * pDoc->Get_s(j) * pDoc->Get_s(j)) *
                                pDoc->GetDelta_s(j);
```

```
                }
        }
}
```

Appendix VI. The fitting procedure in the UED_2004 program: "LeastSquare.cpp"

This contains the functions that calculate the exact solution for the linear parameters as well as the functions that conduct the Levenberg-Marquardt non-linear least-square optimization. Both linear fitting and non-linear refinement can be found in Numerical Recipes in C (Press et al, see references of Chapter 4).

```
void CLeastSquare::FitLinears(CUED_2004Doc *pDoc, double** basis, double *params, double* error,
        BOOL* fit_which, double* weight, int num_par, BOOL* fract_mark)
{
        // see Jon's Lab Notebook #2 p.78-79
        int start_pix = pDoc->GetRangeMin();
        int end_pix = pDoc->GetRangeMax();
        int num_fit = 0;
        int num_const;
        int i, j, k;
        double temp_sum;
        double* const_sum;
        BOOL* fit_fracts;
        double fract_total = 0.0;
        double const_fract_total = 0.0;
        double lamda;

        double** fit_basis;
        double** const_basis;
        double* fit_pars;
        double* const_pars;

        double** fit_matrix;
        double** covariance;

        CCalc calc;

        for(i=0;i<num_par;i++){
                if(fit_which[i]) num_fit++;
                if(fract_mark[i]) fract_total += params[i];
        }
        num_const = num_par - num_fit;

        fit_fracts = new BOOL[num_fit];
        fit_basis = new double *[num_fit];
```

```
for(i=0;i<num_fit;i++) fit_basis[i] = new double[NUM_PIX];
const_basis = new double *[num_const];
for(i=0;i<num_const;i++) const_basis[i] = new double[NUM_PIX];
fit_pars = new double[num_fit];
const_pars = new double[num_const];

k = 0; j = 0;
for(i=0;i<num_par;i++){
        if(fit_which[i]){
                fit_fracts[k] = FALSE;
                for(int l=0;l<NUM_PIX;l++) fit_basis[k][l] = basis[i][l];
                fit_pars[k] = params[i];
                if(fract_mark[i]) fit_fracts[k] = TRUE;
                k++;
        }
        else{
                for(int l=0;l<NUM_PIX;l++) const_basis[j][l] = basis[i][l];
                const_pars[j] = params[i];
                if(fract_mark[i]) const_fract_total += params[i];
                j++;
        }
}

fit_matrix = new double *[num_fit];
covariance = new double *[num_fit];
for(i=0;i<num_fit;i++){
        fit_matrix[i] = new double[num_fit];
        covariance[i] = new double[num_fit];
}

for(i=0;i<num_fit;i++){
        for(k=0;k<num_fit;k++){
                fit_matrix[i][k] = 0.0;
                for(j=start_pix;j<end_pix;j++){
                        fit_matrix[i][k] += weight[j]*fit_basis[i][j]*fit_basis[k][j];
                }
        }
}

calc.InvertMatrix(fit_matrix, covariance, num_fit);

const_sum = new double[num_fit];
for(i=0;i<num_fit;i++){
        const_sum[i] = 0.0;
        for(k=0;k<num_const;k++){
                temp_sum = 0.0;
                for(j=start_pix;j<end_pix;j++){
                        temp_sum += const_basis[k][j] * fit_basis[i][j] * weight[j];
                }
                temp_sum *= (-1 * const_pars[k]);
                const_sum[i] += temp_sum;
        }
}
```

```
	delete [] const_pars;
	for(i=0;i<num_const;i++) delete [] const_basis[i];
	delete [] const_basis;

	for(i=0;i<num_fit;i++){
		fit_pars[i] = 0.0;
		for(int k=0;k<num_fit;k++){
			fit_pars[i] += const_sum[k] * covariance[i][k];
		}
	}

	fract_total -= const_fract_total;
	for(i=0;i<num_fit;i++) if(fit_fracts[i]) fract_total -= fit_pars[i];

	for(i=0;i<num_fit;i++){
		const_sum[i] = 0.0;
		for(k=0;k<num_fit;k++){
			const_sum[i] += fit_fracts[k] * covariance[i][k];
		}
	}

	for(i=0;i<num_fit;i++){
		delete [] fit_basis[i];
		delete [] fit_matrix[i];
	}
	delete [] fit_matrix;
	delete [] fit_basis;

	temp_sum = 0.0;
	for(i=0;i<num_fit;i++) if(fit_fracts[i]) temp_sum += const_sum[i];

	delete [] fit_fracts;

	if(temp_sum != 0.0) lamda = fract_total / temp_sum;
	else lamda = 0.0;

	for(i=0;i<num_fit;i++) fit_pars[i] += lamda * const_sum[i];

	delete [] const_sum;

	k = 0;
	for(i=0;i<num_par;i++){
		if(fit_which[i]){
			params[i] = fit_pars[k];
			error[i] = sqrt(covariance[k][k])*pDoc->GetScaleFactor();
			k++;
		}
	}

	for(i=0;i<num_fit;i++) delete [] covariance[i];
	delete [] covariance;
	delete [] fit_pars;
}
```

This function prepares the set of theory and experimental data for the solution of the linear parameters (coefficients).

```
void CLeastSquare::PrepareLinearBasis(CUED_2004Doc *pDoc, double** basis, double *params,
        double* weight, int num_par)
{
        CCalc calc;
        double* IRefT;

        double fract_sum = 0.0;

        int i,j,k;

        for(i=0;i<num_par;i++){
                params[i] = 0.0;
                for(j=0;j<NUM_PIX;j++) basis[i][j] = 0.0;
        }
        for(i=0;i<NUM_PIX;i++) weight[i] = 0.0;

        // load the theory IMt(s) into basis (fractions)
        k=0;
        for(i=0;i<pDoc->GetNumModels();i++){
                if(pDoc->nulecule[i].fraction != 0.0){
                        params[k] = pDoc->nulecule[i].fraction;
                        for(j=0;j<NUM_PIX;j++) basis[k][j] = pDoc->nulecule[i].sMsSingle[j];
                        k++;
                }
        }

        // load the s^i into basis (background polynomial parameters)
        for(i=0;i<pDoc->GetPolyBkgOrder();i++){
                params[k] = pDoc->GetPolyBkg(i);
                calc.InitPolyBkg(pDoc, basis[k]);
                for(j=0;j<NUM_PIX;j++){
                        for(int l=0;l<i;l++) basis[k][j] *= pDoc->Get_s(j);
                }
                k++;
        }

        // load the data into basis (scale factor)
        params[k] = pDoc->GetScaleFactor()*-1;
        IRefT = new double[NUM_PIX];
        calc.CalcSingleAtomic(pDoc, pDoc->GetAtomicRef(), IRefT);
        for(j=0;j<NUM_PIX;j++){
                basis[k][j] = pDoc->GetRawData(j);
                weight[j] = 1/(pDoc->GetRawSigma(j) * pDoc->GetRawSigma(j));
                if(pDoc->UseAtomicRef()){
                        basis[k][j] *= IRefT[j];
                        weight[j] /= (IRefT[j] * IRefT[j]);
                }
```

```
        }
        delete [] IRefT;
        k++;

        // load the atomic scattering into basis  (atomic scattering coeficient)
        params[k] = pDoc->GetAtomicScale();
        calc.CalcAtomic(pDoc, basis[k]);

        if(k+1 != num_par) AfxMessageBox("Error in linear parameters!");
}
```

This function checks the user input and determines the number of linear parameters that will be solved for.

```
int CLeastSquare::GetNumLinearPars(CUED_2004Doc *pDoc)
{
        int numpars = 0;

        numpars += pDoc->GetNumModels();

        numpars += pDoc->GetPolyBkgOrder();          // for the background

        numpars++;                                   // for the scale factor

        numpars++;                                   // for the atomic scattering coefficient

        return numpars;
}
void CLeastSquare::UpdateLinears(CUED_2004Doc *pDoc, double *params)
{
        int i,k;

        k=0;
        for(i=0;i<pDoc->GetNumModels();i++){
                if(pDoc->nulecule[i].fraction != 0.0){
                        pDoc->nulecule[i].fraction = params[k];
                        k++;
                }
        }

   for(i=0;i<pDoc->GetPolyBkgOrder();i++){
                pDoc->SetPolyBkg(i,params[k]);
                k++;
        }

        pDoc->SetScaleFactor(params[k]*-1);
        k++;

        pDoc->SetAtomicScale(params[k]);
}
```

This function performs the Levenberg-Marquardt non-linear least-squares refinement of the structural parameters, molecular temperature, and camera distance.

```
void CLeastSquare::FitNonlinearParams(CUED_2004Doc *pDoc, BOOL FitCamDist, double *cam_error,
        double** basis, double *linears, double* lin_errors, BOOL* fit_which_lin, double* weight,
        int num_lin_par, BOOL* fracts)
{
        // see numerical recipes p.681+

        double chisqr1, chisqr2, chisqr3;
        CString status;

        double old_cam_dist;

        double *y1;
        double *y2;
        double *y3;
        double *w1;
        double *w2;
        double *w3;

        int num_fit=0;
        int range = pDoc->GetRangeMax() - pDoc->GetRangeMin();
        int i,j,k,l,n;
        double** dyda;                                  // function, first derivative of chisqr
        double* beta;                                   // first derivative of chisqr
        double** alpha;                                 // Hessian
        double** alphaprime;            // Hessian with the lamda for Levenberg-Marquardt routine
        double** inverted_matrix;               // inverted Hessian
        CCalc calc;

        double* step;                                   // the step added to the parameters
        double lamda = 0.001;                           // adjustable parameter for Levenberg-Marquardt
        double* old_params;
        double* temp_errors;

        for(i=0;i<pDoc->NumParams;i++) if(pDoc->FitThisParam[i]) num_fit++;
        if(FitCamDist) num_fit++;

        status.Format("Total parameters = %d.  Total parameters in fit = %d", pDoc->NumParams +
                FitCamDist, num_fit);
        pDoc->AddStatusString(status);

        dyda = new double *[num_fit];
        beta = new double[num_fit];
        alpha = new double *[num_fit];
        alphaprime = new double *[num_fit];
        inverted_matrix = new double *[num_fit];
        step = new double[num_fit];
        old_params = new double[num_fit];
        for(i=0;i<num_fit;i++){
```

```
            dyda[i] = new double[range];
            alpha[i] = new double[num_fit];
            alphaprime[i] = new double[num_fit];
            inverted_matrix[i] = new double[num_fit];
}
y1 = new double[range];
y2 = new double[range];
y3 = new double[range];
w1 = new double[range];
w2 = new double[range];
w3 = new double[range];

for(n=0;n<MAX_ITERATIONS;n++){
            ///// center point on ChiSqrd curve  for numerical derivative approximation  /////
            pDoc->ChisqrR();

            chisqr2 = pDoc->chisqr;
            for(i=pDoc->GetRangeMin(), j=0;i<pDoc->GetRangeMax(); i++){
                        y2[j] = pDoc->out_data[i] - pDoc->out_theory[i];
                        w2[j] = 1/(pDoc->out_sigma[i] * pDoc->out_sigma[i]);
                        j++;
            }

            k=0;
            for(i=0;i<pDoc->NumParams + FitCamDist;i++){
                        if(i<pDoc->NumParams){
                                    if(!pDoc->FitThisParam[i]) continue;
                                    sscanf(pDoc->FitParamValues[i],"%lf",&old_params[k]);
                                                // get the original parameter
                        }
                        else old_cam_dist = pDoc->GetCamDist();

                        if(i<pDoc->NumParams){
                                    pDoc->FitParamValues[i].Format("%.8lf",(old_params[k] +
                                                DELTA_FOR_DERIV));
                                    pDoc->UpdateModel(pDoc->FitParamMolIndex[i]);
                        }
                        else{
                                    pDoc->SetCamDist(old_cam_dist + DELTA_FOR_DERIV);
                                    pDoc->Calc_s();
                                    pDoc->LoadScatteringFactors();
                        }
                        pDoc->CalcIMs();
                        PrepareLinearBasis(pDoc, basis, linears, weight, num_lin_par);
                        FitLinears(pDoc, basis, linears, lin_errors, fit_which_lin, weight, num_lin_par,
                                    fracts);
                        UpdateLinears(pDoc, linears);
                        pDoc->PrepareTheory();
                        pDoc->PrepareData();
                        pDoc->ChisqrR();
                        chisqr1 = pDoc->chisqr;
                        for(l=pDoc->GetRangeMin(), j=0;l<pDoc->GetRangeMax(); l++){
                                    y1[j] = pDoc->out_data[l] - pDoc->out_theory[l];
```

```
                    w1[j] = 1/(pDoc->out_sigma[l] * pDoc->out_sigma[l]);
                    j++;
            }

            /////  step the other direction and get th next point on ChiSqrd curve for
                    numerical derivative approximation  ////
            if(i<pDoc->NumParams){
                    pDoc->FitParamValues[i].Format("%.8lf",(old_params[k] -
                            DELTA_FOR_DERIV));
                    pDoc->UpdateModel(pDoc->FitParamMolIndex[i]);
            }
            else{
                    pDoc->SetCamDist(old_cam_dist - DELTA_FOR_DERIV);
                    pDoc->Calc_s();
                    pDoc->LoadScatteringFactors();
            }
            pDoc->CalcIMs();
            PrepareLinearBasis(pDoc, basis, linears, weight, num_lin_par);
            FitLinears(pDoc, basis, linears, lin_errors, fit_which_lin, weight, num_lin_par,
                    fracts);
            UpdateLinears(pDoc, linears);
            pDoc->PrepareTheory();
            pDoc->PrepareData();
            pDoc->ChisqrR();
            chisqr3 = pDoc->chisqr;
            for(l=pDoc->GetRangeMin(), j=0;l<pDoc->GetRangeMax(); l++){
                    y3[j] = pDoc->out_data[l] - pDoc->out_theory[l];
                    w3[j] = 1/(pDoc->out_sigma[l] * pDoc->out_sigma[l]);
                    j++;
            }

            for(j=0;j<range;j++) dyda[k][j] = (y1[j] - y3[j])/(2 * DELTA_FOR_DERIV);
            beta[k] = 0.0;
            for(j=0;j<range;j++) beta[k] += -1 * w2[j] * y2[j] * dyda[k][j] - 0.5 * y2[j] *
                    y2[j] * (w1[j] - w3[j])/(2 * DELTA_FOR_DERIV);
                            //first deriv of chisqr


            if(i<pDoc->NumParams){
                    // bring it back to original position
                    pDoc->FitParamValues[i].Format("%.8lf",old_params[k]);
                    pDoc->UpdateModel(pDoc->FitParamMolIndex[i]);
            }
            else{
                    pDoc->SetCamDist(old_cam_dist);
                    pDoc->Calc_s();
                    pDoc->LoadScatteringFactors();
            }
            k++;
    }

    for(k=0;k<num_fit;k++){
            for(l=0;l<num_fit;l++){
```

```
                    alpha[k][l] = 0.0;
                    alphaprime[k][l] = 0.0;
                    inverted_matrix[k][l] = 0.0;
                    for(j=0;j<range;j++){
                        alpha[k][l] += w2[j] * dyda[k][j] * dyda[l][j];
                        alphaprime[k][l] += w2[j] * dyda[k][j] * dyda[l][j];
                    }
                    if(k==l)  alphaprime[k][l] += lamda;
                }
            }
            calc.InvertMatrix(alphaprime, inverted_matrix, num_fit);
            for(k=0;k<num_fit;k++){
                step[k] = 0.0;
                for(l=0;l<num_fit;l++){
                    step[k] += inverted_matrix[k][l] * beta[l];
                }
            }

            k=0;
            j = -1;
            for(i=0;i<pDoc->NumParams + FitCamDist;i++){
                if(i<pDoc->NumParams){
                    if(!pDoc->FitThisParam[i]) continue;
                    if(j != -1 && j != pDoc->FitParamMolIndex[i])
                        pDoc->UpdateModel(pDoc->FitParamMolIndex[i]);
                    j = pDoc->FitParamMolIndex[i];
                    pDoc->FitParamResults[i] = old_params[k] + step[k];
                    pDoc->FitParamValues[i].Format("%.8lf",pDoc->FitParamResults[i]);
                    status.Format("%s = %.8lf; beta = %.8lf; alpha = %.8lf",
                        pDoc->FitParamLabels[i], pDoc->FitParamResults[i], beta[k],
                        alpha[k][k]);
                    pDoc->AddStatusString(status);
                }
                else{
                    pDoc->SetCamDist(old_cam_dist + step[k]);
                    status.Format("Camera Distance = %.8lf; beta = %.8lf; alpha = %.8lf",
                        pDoc->GetCamDist(), beta[k], alpha[k][k]);
                    pDoc->AddStatusString(status);
                }
                k++;
            }
            if(j!=-1) pDoc->UpdateModel(j);
            if(FitCamDist){
                pDoc->Calc_s();
                pDoc->LoadScatteringFactors();
            }
            pDoc->CalcIMs();
            PrepareLinearBasis(pDoc, basis, linears, weight, num_lin_par);
            FitLinears(pDoc, basis, linears, lin_errors, fit_which_lin, weight, num_lin_par, fracts);
            UpdateLinears(pDoc, linears);
            pDoc->PrepareTheory();
            pDoc->PrepareData();
            pDoc->ChisqrR();
```

```
            chisqr1 = pDoc->chisqr;
            status.Format("Chisqr before step = %lf.  Chisqr after step = %lf.", chisqr2, chisqr1);
            pDoc->AddStatusString(status);


            if(fabs(chisqr1 - chisqr2) < pDoc->tolerance) break;   // instead of TOLERANCE
            if(chisqr2 > chisqr1) lamda /= 10;
            else {
                  k = 0;
                  j = -1;
                  for(i=0;i<pDoc->NumParams + FitCamDist;i++){
                        if(i<pDoc->NumParams){
                              if(!pDoc->FitThisParam[i]) continue;
                              if(j != -1 && j != pDoc->FitParamMolIndex[i])
                                    pDoc->UpdateModel(pDoc->FitParamMolIndex[i]);
                              j = pDoc->FitParamMolIndex[i];
                              pDoc->FitParamResults[i] = old_params[k];
                              pDoc->FitParamValues[i].Format("%.8lf",old_params[k]);
                        }
                        else pDoc->SetCamDist(old_cam_dist);
                        k++;
                  }
                  if(j!=-1) pDoc->UpdateModel(j);
                  if(FitCamDist){
                        pDoc->Calc_s();
                        pDoc->LoadScatteringFactors();
                  }
                  pDoc->CalcIMs();
                  PrepareLinearBasis(pDoc, basis, linears, weight, num_lin_par);
                  FitLinears(pDoc, basis, linears, lin_errors, fit_which_lin, weight, num_lin_par,
                        fracts);
                  UpdateLinears(pDoc, linears);
                  pDoc->PrepareTheory();
                  pDoc->PrepareData();
                  if(chisqr2 < chisqr1) lamda *= 10;
                  else {pDoc->AddStatusString("Fitting Error!"); break;}
            }
            if(lamda > LAMDA_OVERFLOW) {
                  pDoc->AddStatusString("Lamda Overflow!");
                  n++;
                  break;
            }
      }

      pDoc->ChisqrR();
      status.Format("Step %d: Final Chi Squared = %.4lf, DeltaQ = %.8lf", n, pDoc->chisqr,
            pDoc->DeltaQ());
      pDoc->AddStatusString(status);
      temp_errors = new double[num_fit];
      calc.InvertMatrix(alpha,inverted_matrix,num_fit);
      for(k=0;k<num_fit;k++){
            for(l=0;l<num_fit;l++){
                  if(k==l){
```

```
                                        temp_errors[k] = sqrt(inverted_matrix[k][l]);
//                                      temp_errors[k] = sqrt(inverted_matrix[k][l]) *
//                                              sqrt(pDoc->chisqr/range);
                                                        // this is the formula in the old program
                                }
                        }
                }

                status.Format("Non-linear fitting took %d steps.", n);
                pDoc->AddStatusString(status);
                k=0;
                for(i=0;i<pDoc->NumParams + FitCamDist;i++){
                        if(i<pDoc->NumParams){
                                if(pDoc->FitThisParam[i]){
                                        pDoc->FitParamErrors[i] = temp_errors[k];
                                        k++;
                                }
                                else pDoc->FitParamErrors[i] = 0.0;
                        }
                        else {
                                *cam_error = temp_errors[k];
                                k++;
                        }
                }
                pDoc->UpdateAllViews(NULL);

                for(i=0;i<num_fit;i++){
                        delete [] dyda[i];
                        delete [] alpha[i];
                        delete [] alphaprime[i];
                        delete [] inverted_matrix[i];
                }
                delete [] dyda;
                delete [] step;
                delete [] temp_errors;
                delete [] alpha;
                delete [] alphaprime;
                delete [] inverted_matrix;
                delete [] old_params;
                delete [] beta;
                delete [] y1;
                delete [] y2;
                delete [] y3;
                delete [] w1;
                delete [] w2;
                delete [] w3;
}
```