The xraylib API list of all functions

Tom Schoonjans edited this page 16 days ago · 22 revisions

Important: with the release of version 3.0.0, we have changed all float datatypes to double! This may require users to make modifications to their existing C/C++/Obj-C and Fortran codes, as well as recompile their software that depends on xraylib. Apart from this datatype transition, we also changed the CompoundParser prototype.

Unless explicitly stated otherwise, all energies are assumed to be expressed in keV, whether used as input argument or return value. Similarly, all angles are assumed to be expressed in radians.

Atomic weights

```
double AtomicWeight(int Z);
```

Given an element `Z`, returns its atomic weight in g/mol.

Element densities

```
double ElementDensity(int Z);
```

Given an element `Z`, returns its density at room temperature in g/cm^3.

Cross sections

```
double CS_Total(int Z, double E);
double CS_Photo(int Z, double E);
double CS_Rayl(int Z, double E);
double CS_Compt(int Z, double E);
```

Given an element `Z` and an energy `E`, these functions will return respectively the total absorption cross section, the photoionization cross section, the Rayleigh scattering cross section and the Compton scattering cross section, expressed in cm^2/g.

```
double CSb_Total(int Z, double E);
double CSb_Photo(int Z, double E);
double CSb_Rayl(int Z, double E);
double CSb_Compt(int Z, double E);
```

Identical to the last four functions, but the cross section is returned expressed in barn/atom.

```
double CS_KN(double E);
```

Given an energy `E`, returns the total Klein-Nishina cross section expressed in barn.

```
double CS_Photo_Partial(int Z, int shell, double E);
```

Given an element `Z`, shell-type macro `shell` and energy `E`, returns the partial photoionization cross section expressed in cm^2/g.

```
double CSb_Photo_Partial(int Z, int shell, double E);
```

Identical to the last function, but the cross section is returned in barn/electron.

```
double CS_Total_Kissel(int Z, double E);
double CSb_Total_Kissel(int Z, double E);
```

Identical to CS_Total and CSb_Total, but the photoionization contribution is calculated using the Kissel cross sections.

```
double CS_Energy(int Z, double E);
```

Given an element `Z` and an energy `E`, returns the mass-energy absorption cross section in cm^{2}g

```
double CS_Total_CP(const char compound[], double E);
double CS_Photo_CP(const char compound[], double E);
double CS_Rayl_CP(const char compound[], double E);
double CS_Compt_CP(const char compound[], double E);
double CSb_Total_CP(const char compound[], double E);
double CSb_Photo_CP(const char compound[], double E);
```

```
double CSb_Rayl_CP(const char compound[], double E);
double CSb_Compt_CP(const char compound[], double E);
double CS_Total_Kissel_CP(const char compound[], double E);
double CSb_Total_Kissel_CP(const char compound[], double E);
double CS_Energy(const char compound[], double E);
```

Identical to the earlier mentioned functions, but require a chemical formula or a NIST compound name `compound` as first argument.

Unpolarized differential scattering cross sections

```
double DCS_Thoms(double theta);
```

Given a scattering polar angle `theta`, returns the Thomson differential cross section expressed in barn.

```
double DCS_KN(double E, double theta);
```

Given an energy `E` and a scattering polar angle `theta`, returns the Klein-Nishina differential scattering cross section expressed in barn.

```
double DCS_Rayl(int Z, double E, double theta);
double DCS_Compt(int Z, double E, double theta);
```

Given an element `Z`, energy `E` and a scattering polar angle `theta`, returns respectively the differential Rayleigh and the differential Compton scattering cross section expressed in $cm^2/g/sterad$.

```
double DCSb_Rayl(int Z, double E, double theta);
double DCSb_Compt(int Z, double E, double theta);
```

Identical to the last two functions, but the cross section is returned expressed in barn/atom/sterad.

```
double DCS_Rayl_CP(const char compound[], double E, double theta);
double DCS_Compt_CP(const char compound[], double E, double theta);
double DCSb_Rayl_CP(const char compound[], double E, double theta);
double DCSb_Compt_CP(const char compound[], double E, double theta);
```

Identical to the earlier mentioned functions, but require a chemical formula or a NIST compound name `compound` as first argument.

Polarized differential scattering cross sections

```
double DCSP_Thoms(double theta, double phi);
```

Given a scattering polar angle `theta` and scattering azimuthal angle `phi`, returns the Thomson differential cross section for a polarized beam expressed in barn.

```
double DCSP_KN(double E, double theta, double phi);
```

Given an energy `E`, a scattering polar angle `theta` and scattering azimuthal angle `phi`, returns the Klein-Nishina differential cross section for a polarized beam expressed in barn.

```
double DCSP_Rayl(int Z, double E, double theta, double phi);
double DCSP_Compt(int Z, double E, double theta, double phi);
```

Given an element `Z`, an energy `E`, a scattering polar angle `theta` and scattering azimuthal angle `phi`, returns respectively the Rayleigh differential and Compton differential cross sections for a polarized beam expressed in $cm^2/g/sterad$.

```
double DCSPb_Rayl(int Z, double E, double theta, double phi);
double DCSPb_Compt(int Z, double E, double theta, double phi);
```

Identical to the last two functions, but the cross section is returned expressed in barn/atom/sterad.

```
double DCSP_Rayl_CP(const char compound[], double E, double theta, double phi);
double DCSP_Compt_CP(const char compound[], double E, double theta, double phi);
double DCSPb_Rayl_CP(const char compound[], double E, double theta, double phi);
double DCSPb_Compt_CP(const char compound[], double E, double theta, double phi);
```

Identical to the earlier mentioned functions, but require a chemical formula or a NIST compound name `compound` as first argument.

Scattering factors

In this section, we introduce the momentum transfer parameter `q`, which is used in several of the following functions. It should be noted that several definitions can be found of this parameter throughout the scientific literature, which vary mostly depending on the community where it is used. The crystallography and diffraction community for example, use the following definition:

> $q = 4\pi \times \sin(\theta)/\lambda$

with θ the angle between the incident X-ray and the crystal scattering planes according to Bragg's law, and λ the wavelength. xraylib uses however, a different definition, in which θ corresponds to the scattering angle, which in case of Bragg scattering is equal to twice the angle from the previous definition. This new definition has the advantage of being useful when working with amorphous materials, as well as with incoherent scattering. Furthermore, our definition drops the 4π scale factor, in line with the definition by Hubbell et al in *Atomic form factors, incoherent scattering functions, and photon scattering cross sections*, J. Phys. Chem. Ref. Data, Vol.4, No. 3, 1975:

> $q = E \times \sin(\theta/2) \times h \times c \times 10^8$

with E the energy of the photon, h Planck's constant and c the speed of light. The unit of the returned momentum transfer is then $Å^{-1}$.

```
double FF_Rayl(int Z, double q);
```

Given an element `Z` and a momentum transfer `q` (expressed in $Å^{-1}$), returns the atomic form factor for Rayleigh scattering.

```
double SF_Compt(int Z, double q);
```

Given an element `Z` and a momentum transfer `q` (expressed in $Å^{-1}$), returns the incoherent scattering function for Compton scattering.

```
double MomentTransf(double E, double theta);
```

Given an energy `E` and a scattering polar angle `theta`, returns the momentum transfer for X-ray photon scattering expressed in $Å^{-1}$.

```
double Fi(int Z, double E);
double Fii(int Z, double E);
```

Given an element `Z` and and energy `E`, returns respectively the anomalous scattering factors Δf' and Δf".

X-ray fluorescence line energies

```
double LineEnergy(int Z, int line);
```

Given an element `Z` and line-type macro `line`, returns the energy of the requested XRF line expressed in keV.

X-ray fluorescence yields

```
double FluorYield(int Z, int shell);
```

Given an element `Z` and shell-type macro `shell`, returns the corresponding fluorescence yield .

Auger yields

```
double AugerYield(int Z, int shell);
```

Given an element `Z` and shell-type macro `shell`, returns the corresponding Auger yield .

Coster-Kronig transition probabilities

```
double CosKronTransProb(int Z, int trans);
```

Given an element `Z` and transition-type macro `trans`, returns the corresponding Coster-Kronig transition probability.

Absorption edge energies

```
double EdgeEnergy(int Z, int shell);
```

Given an element `Z` and shell-type macro `shell`, returns the absorption edge energy expressed in keV.

Jump factors

```
double JumpFactor(int Z, int shell);
```

Given an element `Z` and shell-type macro `shell`, returns the jump factor.

X-ray fluorescence cross sections

```
double CS_FluorLine(int Z, int line, double E);
```

Given an element `Z`, a line-type macro `line` and an energy `E`, returns the XRF cross section expressed in cm^2/g

```
double CSb_FluorLine(int Z, int line, double E);
```

Identical to the previous function, but returns the cross section expressed in barn/atom. These last two functions calculate the XRF cross sections assuming the jump factor approximation. We also offer XRF cross sections calculated using the partial photoelectric effect cross sections calculated by Kissel et al. The corresponding functions are:

```
double CS_FluorLine_Kissel(int Z, int line, double E);
double CSb_FluorLine_Kissel(int Z, int line, double E);
```

Recently we introduced XRF cross sections that take into account cascade effects, both those coming from radiative transitions and those from non-radiative transitions :

```
double CS_FluorLine_Kissel_Cascade(int Z, int line, double E);
double CSb_FluorLine_Kissel_Cascade(int Z, int line, double E);
double CS_FluorLine_Kissel_Nonradiative_Cascade(int Z, int line, double E);
double CSb_FluorLine_Kissel_Nonradiative_Cascade(int Z, int line, double E);
double CS_FluorLine_Kissel_Radiative_Cascade(int Z, int line, double E);
double CSb_FluorLine_Kissel_Radiative_Cascade(int Z, int line, double E);
double CS_FluorLine_Kissel_no_Cascade(int Z, int line, double E);
double CSb_FluorLine_Kissel_no_Cascade(int Z, int line, double E);
```

`CS_FluorLine_Kissel` and `CbS_FluorLine_Kissel` are mapped to resp. `CS_FluorLine_Kissel_Cascade` and `CSb_FluorLine_Kissel_Cascade`. Using these functions, it is possible to examine the influence of the two different cascade types separately, but keep in mind that in reality they will always be occuring simultaneously.

All `CS_FluorLine*` functions offer XRF cross sections for both K- and L-lines (provided the corresponding shells can be excited), but only the `CS_FluorLine_Kissel*` functions offer also the M-line XRF cross sections!

Radiative rates

```
double RadRate(int Z, int line);
```

Given an element `Z` and a line-type macro `line`, returns the radiative rate.

Non-radiative rates

```
double AugerRate(int Z, int auger);
```

Given an element `Z` and an Auger-type macro `auger` corresponding with the electrons involved, returns the non-radiative rate.

Atomic level widths

```
double AtomicLevelWidth(int Z, int shell);
```

Given an element `Z` and a shell-type macro `shell`, returns the atomic level width in keV.

Compton energy

```
double ComptonEnergy(double E0, double theta);
```

Given an initial photon energy `E0` and a scattering polar angle `theta`, returns the photon energy after Compton scattering.

Refractive indices

```
double Refractive_Index_Re(const char compound[], double E, double rho);
double Refractive_Index_Im(const char compound[], double E, double rho);
xrlComplex Refractive_Index(const char compound[], double E, double rho);
```

Given a chemical formula `compound`, energy `E` and a density `rho`, return respectively the real, the imaginary or both parts of the refractive index. For a definition of `xrlComplex`, see the crystal diffraction section.

Compton profiles

```
double ComptonProfile(int Z, double pz);
```

Given an element `Z` and a momentum `pz` (expressed in atomic units), returns the Compton scattering profile summed over all shells.

```
double ComptonProfile_Partial(int Z, int shell, double pz);
```

Given an element `Z`, a shell-type macro `shell` and a momentum `pz`, returns the Compton scattering profile for a particular subshell.

Electronic configurations

```
double ElectronConfig(int Z, int shell);
```

Given an element `Z` and a shell-type macro `shell`, returns the number of electrons the shell possesses.

Crystal diffraction

```
Crystal_Struct* Crystal_GetCrystal(const char* material, Crystal_Array* c_array);
```

Get a pointer to a Crystal_Struct of a given crystal `material` from `c_array`. If `c_array` is `NULL` then the internal array of crystals is searched. If not found, `NULL` is returned. The `c_array` argument is only used in C, C++ and Fortran. The other bindings support only the internal array.

```
double Bragg_angle (Crystal_Struct* cryst, double E, int i_miller, int j_miller, int k_mille
```

Computes the Bragg angle in radians for a given crystal `cryst`, energy `E` and Miller indices `i_miller`, `j_miller` and `k_miller`.

```
double Q_scattering_amplitude(Crystal_Struct* cryst, double E, int i_miller, int j_miller, i
```

Computes the Q scattering amplitude for a given crystal `cryst`, incident energy `E`, Miller indices (`i_miller`, `j_miller` and `k_miller`) and relative angle `rel_angle`.

```
void Atomic_Factors (int Z, double E, double q, double debye_factor, double* f0, double* f_p
```

Computes the atomic factors f_0 `f0`, $\Delta f'$ `f_prime` and $\Delta f''$ `f_prime2` for a given element `Z`, incident energy `E`, momentum transfer `q` and Debye factor `debye_factor`. `f0`, `f_prime` and `f_prime2` are pointers to doubles in C, C++, Fortran and IDL BUT return values in Perl, Python and Lua!!

```
xrlComplex Crystal_F_H_StructureFactor (Crystal_Struct* cryst, double E, int i_miller, int j
```

Computes the F_H Structure factor for a given crystal `cryst`, incident energy `E`, Miller indices (`i_miller`, `j_miller` and `k_miller`), Debye factor `debye_factor` and relative angle `rel_angle`. The return value is a complex number.

```
xrlComplex Crystal_F_H_StructureFactor_Partial (Crystal_Struct* crystal, double energy, int
```

with:

```
typedef struct {
        double re;
        double im;
} xrlComplex;
```

with:

- `re`: real part
- `im`: imaginary part

Wherever possible for the bindings (Python, IDL, Perl, Ruby, Fortran, C#), we have tried using the native complex number datatype in favor of a direct analogue of the xrlComplex struct.

See also `Crystal_F_H_StructureFactor`.

The Atomic structure factor has three terms:

> $F_H = f_0 + \Delta f' + \Delta f''$.

For each of these three terms, there is a corresponding `*_flag` argument which controls the numerical value used in computing F_H:

`*_flag` = 0 → Set this term to 0.

`*_flag` = 1 → Set this term to 1. Only used for f_0.

`*_flag` = 2 → Set this term to the value given.

```
double Crystal_UnitCellVolume (Crystal_Struct* cryst);
```

Computes the unit cell volume for a crystal `cryst`. Structures obtained from the official array will have their volume in .volume

```
double Crystal_dSpacing (Crystal_Struct* cryst, int i_miller, int j_miller, int k_miller);
```

Computes the d-spacing for a given crystal `cryst` and Miller indices (`i_miller`, `j_miller` and `k_miller`). The routine assumes that if `cryst->volume` is nonzero then it holds a valid value. If (i,j,k) = (0,0,0) then zero is returned.

Compound parser

```
struct compoundData {
        int nElements;
        double nAtomsAll;
        int *Elements;
        double *massFractions;
};
```

with:

- `nElements` : number of different elements in the compound
- `nAtomsAll` : number of atoms in the formula. Since indices may be real numbers, this member variable is of type double
- `Elements` : a dynamically allocated array (length = `nElements`) containing the elements, in ascending order
- `massFractions` : a dynamically allocated array (length = `nElements`) containing the mass fractions of the elements in Elements

```
struct compoundData *CompoundParser(const char compoundString[])
```

The CompoundParser function will parse a chemical formula `compoundString` and will allocate a compoundData structure with the results if successful, otherwise `NULL` is returned. Chemical formulas may contain (nested) brackets, followed by an integer or real number (with a dot) subscript. Examples of accepted formulas are: `H2O` , `Ca5(PO4)3F` , `Ca5(PO4)F0.33Cl0.33(OH)0.33` .

The allocated memory should be freed with

```
FreeCompoundData(struct compoundData *cd); (C/C++/Obj-C and Fortran only)
```

```
char * AtomicNumberToSymbol(int Z);
```

The AtomicNumberToSymbol function returns a pointer to a string containing the element for atomic number `Z` . If an error occurred, the `NULL` string is returned. The string should be freed after usage with the xrlFree function (C/C++/Obj-C and Fortran only).

```
int SymbolToAtomicNumber(char *symbol);
```

The SymbolToAtomicNumber function returns the atomic number that corresponds with element `symbol` . If the element does not exist, 0 is returned.

NIST compound catalog

```
struct compoundDataNIST {
        char *name;
        int nElements;
        int *Elements;
        double *massFractions;
        double density;
};
```

with:

- `name` : a string containing the full name of the compound, as retrieved from the NIST database
- `nElements` : number of different elements in the compound
- `Elements` : a dynamically allocated array (length = `nElements`) containing the elements, in ascending order
- `massFractions` : a dynamically allocated array (length = `nElements`) containing the mass fractions of the elements in Elements
- `density` : the density of the compound, expressed in g/cm^3

```
struct compoundDataNIST *GetCompoundDataNISTByName(const char compoundString[]);

struct compoundDataNIST *GetCompoundDataNISTByIndex(int compoundIndex);
```

Using these two functions it is possible to query the contents of NISTs catalog of compound compositions and densities. The former takes a compound name `compoundString` and if a match is found, the corresponding newly allocated compoundDataNIST structure is returned, while the latter takes an index `compoundIndex` in the form of a NIST compound-type macro. The list of compound names can be queried using:

```
char **GetCompoundDataNISTList(int *nCompounds);
```

which returns a `NULL` terminated array of strings. Optionally, pass a pointer to an integer `nCompounds` to obtain the number of strings in the array (pass `NULL` if value is not required). This option is only present in the C/C++/Obj-C implementation. The list can also be obtained at our online xraylib calculator.

After usage, the returned compoundDataNIST structures should be freed with (C/C++/Obj-C and Fortran only):

```
void FreeCompoundDataNIST(struct compoundDataNIST *compoundData);
```

Radionuclides

```
struct radioNuclideData {
        char *name;
        int Z;
        int A;
        int N;
        int Z_xray;
        int nXrays;
        int *XrayLines;
        double *XrayIntensities;
        int nGammas;
        double *GammaEnergies;
        double *GammaIntensities;
};
```

with:

- `name` : a string containing the mass number (A), followed by the chemical element (e.g. 55Fe)
- `Z` : atomic number of the radionuclide
- `A` : mass number of the radionuclide
- `N` : number of neutrons of the radionuclide
- `Z_xray` : atomic number of the nuclide after decay, which should be used in calculating the energy of the emitted X-ray lines using `LineEnergy`
- `nXrays` : number of emitted characteristic X-rays
- `XrayLines` : a dynamically allocated array (length = `nXrays`) of line-type macros, identifying the emitted X-rays
- `XrayIntensities` : a dynamically allocated array (length = `nXrays`) of photons per disintegration, one value per emitted X-ray
- `nGammas` : number of emitted gamma-rays
- `GammaEnergies` : a dynamically allocated array (length = `nGammas`) of emitted gamma-ray energies
- `GammaIntensities` : a dynamically allocated array (length = `nGammas`) of emitted gamma-ray photons per disintegration

```
struct radioNuclideData *GetRadioNuclideDataByName(const char radioNuclideString[]);

struct radioNuclideData *GetRadioNuclideDataByIndex(int radioNuclideIndex);
```

Use these two functions to query xraylib's database of X-ray emission profiles for several important radionuclides. The former expects the name `radioNuclideString` of a radionuclide, while the latter takes a radionuclide-type macro `radioNuclideIndex`. When successful, a freshly allocated radioNuclideData structure is returned. Query the list of names using:

```
char **GetRadioNuclideDataList(int *nRadioNuclides);
```

which returns a `NULL` terminated array of strings. Optionally, pass a pointer to an integer `nRadioNuclides` to obtain the number of strings in the array (pass `NULL` if value is not required). This option is only present in the C/C++/Obj-C implementation. The list can also be obtained at our online xraylib calculator.

After usage, the returned radioNuclideData structures should be freed with (C/C++/Obj-C and Fortran only):

```
void FreeRadioNuclideData(struct radioNuclideData *radioNuclideData);
```

Error handling

xraylib's error handling consists of error messages that are presented to the user whenever necessary, most commonly when invalid arguments were detected. Internally a global variable `ExitStatus` will be set to 1 at the first occurrence of an error. It is possible to force the calling program to exit in this case, by using the `SetHardExit` function, before calling the function that could trigger the error. If you are merely interested in knowing whether an error occurred or not, check the return value of `GetExitStatus`, after calling the function that could trigger the error. Consider setting `SetExitStatus(0)` before to ensure that no previous errors would influence the outcome of `GetExitStatus`. The following functions determine the error handling behavior:

```
void SetHardExit(int hard_exit);
```

Pass 1 to exit the program at the first error.

```
void SetExitStatus(int exit_status);

int GetExitStatus(void);
```

These two functions allow for the setting and getting of the `ExitStatus`. If it is equal to 1, then an error has occurred, while 0 indicates normal operation.

The authors of xraylib strongly discourage the use of these three functions. Due to the internal use of a global variable, they are NOT thread-safe. Furthermore, some functions will set the `ExitStatus` variable to 1, although no error has occurred! This is considered a bug and will be fixed in a future update. For now, the authors recommend to check the return value of the functions: 0 indicates either an error or an unavailable quantity.

Since some of the error messages are in fact meaningless (in particular those produced by the XRF cross section functions), it may be useful to turn them off completely. Use the following functions for this purpose:

```
void SetErrorMessages(int status);

int GetErrorMessages(void);
```

Passing 0 to `SetErrorMessages` suppresses the output of the error messages, while 1 restores it.