XSFunctions Guide

Version 1.00

Keith A. Arnaud

HEASARC
Code 662
Goddard Space Flight Center
Greenbelt, MD 20771
USA

May 2014

Contents

Chapter 1

Introduction

This document describes the C++ classes available in the XSPEC model functions library (XSFunctions). While these are not controlled as rigorously as XSPEC external interfaces, we do try to keep them as stable as possible so we encourage their use in other XSPEC models.

The classes are

- Aped - generates CEI and NEI spectra using AtomDB input files.
- IonBalNei - calculates NEI ionization balances.
- NeutralOpacity - calculates opacities for neutral material.
- IonizedOpacity - calculates opacities for photo-ionized material.
- MZCompRefl - calculates Compton reflection using the Magzdiarz and Zdziarski code.

Chapter 2

Aped

The Aped classes store and use the data stored in the AtomDB files. The top-level class, Aped, comprises a vector of ApedTemperatureRecord objects, each of which comprises a vector of ApedElementRecord objects, each of which comprises a vector of ApedIonRecord objects. Thus, there is one ApedIonRecord for each temperature, element, and ion in the AtomDB files. The top-level Aped class provides the methods to load and use the atomic data.

2.1 ApedIonRecord class

```
class ApedIonRecord{
 public:

  int Ion;
  RealArray ContinuumEnergy;
  RealArray ContinuumFlux;
  RealArray ContinuumFluxError;
  RealArray PseudoContinuumEnergy;
  RealArray PseudoContinuumFlux;
  RealArray PseudoContinuumFluxError;
  RealArray LineEnergy;
  RealArray LineEnergyError;
  RealArray LineEmissivity;
  RealArray LineEmissivityError;
  IntegerArray ElementDriver;
  IntegerArray IonDriver;
  IntegerArray UpperLevel;
  IntegerArray LowerLevel;

  ApedIonRecord();    // default constructor
  ~ApedIonRecord();   // destructor
```

```
  ApedIonRecord& operator=(const ApedIonRecord&);    // deep copy

};
```

The class includes entries to store error estimates on the fluxes and emissivities although these are not used at the time of writing. Note that the same classes are used for NEI and CEI although for the latter the ElementDriver and IonDriver arrays are not required.

2.2 ApedElementRecord class

```
class ApedElementRecord{
 public:

  int AtomicNumber;
  vector<ApedIonRecord> IonRecord;

  ApedElementRecord();    // default constructor
  ~ApedElementRecord();   // destructor

  void LoadIonRecord(ApedIonRecord input);

  ApedElementRecord& operator=(const ApedElementRecord&);    // deep copy

};
```

Each ApedElementRecord object stores the atomic number of the element and the records for each of its ions.

2.3 ApedTemperatureRecord

```
class ApedTemperatureRecord{
 public:

  Real Temperature;
  vector<ApedElementRecord> ElementRecord;

  ApedTemperatureRecord();     //default constructor
  ~ApedTemperatureRecord();    //destructor

  void LoadElementRecord(ApedElementRecord input);
```

```
  ApedTemperatureRecord& operator=(const ApedTemperatureRecord&);    // deep copy

};
```

Each ApedTemperatureRecord object stores the temperature and the records for each element.

2.4 Aped class

```
class Aped{
 public:

  vector<ApedTemperatureRecord> TemperatureRecord;

  // These store the information in the initial PARAMETERS extension

  RealArray Temperatures;
  IntegerArray NelementsLine;
  IntegerArray NelementsCoco;
  IntegerArray Nline;
  IntegerArray Ncont;

  string coconame;
  string linename;

  Aped();     //default constructor
  ~Aped();    //destructor

  Aped& operator=(const Aped&);    // deep copy
```

The class provides methods to load the data from the AtomDB continuum and line files. The Read method just gets the information from the PARAMETERS extension, stores the filenames used and sizes the subsidiary objects correctly. To actually load the data use ReadTemperature.

```
  // Reads the continuum and line files and stores the names and
  // temperatures. Does not read the actual continuum and line data.

  int Read(string cocofilename, string linefilename);

  int ReadTemperature(const int TemperatureIndex);
  int ReadTemperature(const vector<int>& TemperatureIndex);
```

The following provide methods to extract some useful numbers.

```
int NumberTemperatures();     // return number of tabulated temperatures
int NumberElements();         // return number of tabulated elements
int NumberIons(int Z);        // return number of ions across all temperatures
                              // for element Z.
```

Whether the data for a given temperature index has been loaded can be checked using IsTemperatureLoaded.

```
bool IsTemperatureLoaded(const int TemperatureIndex);
```

The method to generate spectra for CEI plasmas is SumEqSpectra. This comes in several overloaded options depending on whether a single temperature of distribution of temperatures are required. It is also possible to use a different temperature for thermal broadening of lines than that used for the ionization balance. The energy bins on which the spectrum is to be calculated are specified by standard XSPEC energyArray. The elements required and their abundances are given by Zinput and abundance. Tinput and Dem give the temperature(s) and emission measure(s) required. If qtherm is true then lines will be thermally broadened while if velocity is non-zero then lines will be velocity broadened. These two options will be slower.

```
void SumEqSpectra(const RealArray& energyArray,
                  const IntegerArray& Zinput, const RealArray& abundance,
                  const Real Redshift, const Real& Tinput,
                  const Real& Dem, const bool qtherm, const Real velocity,
                  RealArray& fluxArray, RealArray& fluxErrArray);

void SumEqSpectra(const RealArray& energyArray,
                  const IntegerArray& Zinput, const RealArray& abundance,
                  const Real Redshift, const RealArray& Tinput,
                  const RealArray& Dem, const bool qtherm, const Real velocity,
                  RealArray& fluxArray, RealArray& fluxErrArray);

// case where the temperature used for the thermal broadening differs from that
// for the ionization

void SumEqSpectra(const RealArray& energyArray,
                  const IntegerArray& Zinput, const RealArray& abundance,
                  const Real Redshift, const Real& Tinput,
                  const Real& Tbinput, const Real& Dem, const bool qtherm,
                  const Real velocity, RealArray& fluxArray,
                  RealArray& fluxErrArray);

void SumEqSpectra(const RealArray& energyArray,
                  const IntegerArray& Zinput, const RealArray& abundance,
                  const Real Redshift, const RealArray& Tinput,
```

```
                    const RealArray& Tbinput, const RealArray& Dem,
                    const bool qtherm,
                    const Real velocity, RealArray& fluxArray,
                    RealArray& fluxErrArray);
```

For NEI plasmas, the method to calculate the spectrum is SumNeqSpectra. The inputs differ from SumEqSpectra only in including IonFrac, which specifies the ionization fractions required for each element, for each temperature.

```
  void SumNeqSpectra(const RealArray& energyArray,
                     const IntegerArray& Zinput, const RealArray& abundance,
                     const Real Redshift, const Real& Tinput,
                     const vector<RealArray>& IonFrac,
                     const bool qtherm, const Real velocity,
                     RealArray& fluxArray, RealArray& fluxErrArray);

  void SumNeqSpectra(const RealArray& energyArray,
                     const IntegerArray& Zinput, const RealArray& abundance,
                     const Real Redshift, const RealArray& Tinput,
                     const vector<vector<RealArray> >& IonFrac,
                     const bool qtherm, const Real velocity,
                     RealArray& fluxArray, RealArray& fluxErrArray);

  // case where the temperature used for the thermal broadening differs from that
  // for the ionization

  void SumNeqSpectra(const RealArray& energyArray,
                     const IntegerArray& Zinput, const RealArray& abundance,
                     const Real Redshift, const Real& Tinput,
                     const Real& Tbinput,
                     const vector<RealArray>& IonFrac,
                     const bool qtherm, const Real velocity,
                     RealArray& fluxArray, RealArray& fluxErrArray);

  void SumNeqSpectra(const RealArray& energyArray,
                     const IntegerArray& Zinput, const RealArray& abundance,
                     const Real Redshift, const RealArray& Tinput,
                     const RealArray& Tbinput,
                     const vector<vector<RealArray> >& IonFrac,
                     const bool qtherm, const Real velocity,
                     RealArray& fluxArray, RealArray& fluxErrArray);
```

All versions of SumEqSpectra and SumNeqSpectra operate through

```
  void SumSpectra(const RealArray& energyArray,
```

```
                const IntegerArray& Zinput, const RealArray& abundance,
                const Real Redshift, const RealArray& Tinput,
                const RealArray& Tbinput,
                const vector<vector<RealArray> >& IonFrac,
                const bool qtherm, const Real velocity, const bool isCEI,
                RealArray& fluxArray, RealArray& fluxErrArray);
```

Outside the class there are wrap-up routines which create an internal, static Aped object then call SumSpectra to calculate the output spectrum. There are overloaded versions corresponding to the options for SumEqSpectra and SumNeqSpectra. The int returned will be non-zero in the event of an error on reading the AtomDB files. For the CEI case these routines check the APECROOT variable to find the files to read. For the NEI case they check the NEIVERS and NEIAPECROOT variables.

```
int calcCEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const Real& Tinput,
                    const Real& Dem, const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcCEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const RealArray& Tinput,
                    const RealArray& Dem, const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcCEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const Real& Tinput,
                    const Real& Tbinput,
                    const Real& Dem, const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcCEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const RealArray& Tinput,
                    const RealArray& Tbinput,
                    const RealArray& Dem, const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcNEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const Real& Tinput,
                    const vector<RealArray>& IonFrac,
                    const bool qtherm, const Real velocity,
```

```
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcNEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const RealArray& Tinput,
                    const vector<vector<RealArray> >& IonFrac,
                    const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcNEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const Real& Tinput,
                    const Real& Tbinput,
                    const vector<RealArray>& IonFrac,
                    const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);

int calcNEISpectrum(const RealArray& energyArray,
                    const IntegerArray& Zinput, const RealArray& abundance,
                    const Real Redshift, const RealArray& Tinput,
                    const RealArray& Tbinput,
                    const vector<vector<RealArray> >& IonFrac,
                    const bool qtherm, const Real velocity,
                    RealArray& fluxArray, RealArray& fluxErrArray);
```

A couple of other useful routines also live in Aped.h. getAtomicMass returns the atomic mass for the given atomic number. apedInterpFlux is used to interpolate the continuum and pseudo-continuum arrays.

```
Real getAtomicMass(const int& AtomicNumber);

void apedInterpFlux(const RealArray& inputEnergy, const RealArray& inputFlux,
                    const Real& z15, const Real& coeff,
                    const RealArray& energyArray, RealArray&
fluxArray);
```

2.5 Examples

An example use of the Aped class to calculate a CEI spectrum can be found in the file smdem2.cxx in Xspec/src/XSFunctions. An example use to calculate an NEI spectrum can be found in vvgnei.cxx in the same directory.

Chapter 3

IonBalNei

The IonBalNei classes store the eigenvector data and calculate ionization fractions for an NEI collisional plasma. The top-level class IonBalNei comprises a vector of IonBalTemperatureRecord classes, each of which comprises a vector of IonBalElementRecord classes. Thus there is one IonBalElementRecord for each temperature and element in the eigevector data files. The top-level IonBalNei class provides methods to load and use the eigenvector data.

3.1 IonBalElementRecord

```
class IonBalElementRecord{
 public:

  int AtomicNumber;
  RealArray EquilibriumPopulation;
  RealArray Eigenvalues;
  vector<RealArray> LeftEigenvectors;
  vector<RealArray> RightEigenvectors;

  IonBalElementRecord();     // default constructor
  ~IonBalElementRecord();    // destructor

  void Clear();  // clear out the contents

  IonBalElementRecord& operator=(const IonBalElementRecord&);  // deep copy

};
```

This class stores the eigenvector data for the element with specified AtomicNumber. EquilibriumPopulation stores the CEI ion fractions.

3.2 IonBalTemperatureRecord

```
class IonBalTemperatureRecord{
 public:

  Real Temperature;
  vector<IonBalElementRecord> ElementRecord;

  IonBalTemperatureRecord();        // default constructor
  ~IonBalTemperatureRecord();       // destructor

  void LoadElementRecord(IonBalElementRecord input);

  void Clear();   // clear out the contents

  IonBalTemperatureRecord& operator=(const IonBalTemperatureRecord&);

};
```

Each IonBalTemperatureRecord object stores the temperature and the eigenvector data for each element.

3.3 IonBalNei

```
class IonBalNei{
 public:

  vector<IonBalTemperatureRecord> TemperatureRecord;
  RealArray Temperature;
  string Version;

  IonBalNei();       // default constructor
  ~IonBalNei();      // destructor

  IonBalNei& operator=(const IonBalNei&);    // deep copy

  void Clear();  // clear out the contents
```

The Version string is used to control which version of the eigenvector data files are to be used. If the version is changed then the object is reset using Clear and setVersion returns true. The setVersion method with no input checks NEIVERS and uses that if it has been set.

```
  string getVersion();
```

```
bool setVersion();
bool setVersion(string version);
```

This class provides a Read method to load data from the eigenvector files. This can either be done using a vector of atomic numbers and the directory name and version string, so that the relevant filenames can be constructed, or using a single atomic number and filename.

```
//  Read loads eigenvector data for a set of elements

int Read(vector<int> Z, string dirname, string versionname);

//  ReadElement loads eigenvector data for that element

int ReadElement(int Z, string filename);

void LoadTemperatureRecord(IonBalTemperatureRecord input);
```

The following provide methods to extract some useful numbers.

```
RealArray Temperatures();     // return tabulated temperatures
int NumberTemperatures();     // return number of tabulated temperatures
int NumberElements();         // return number of tabulated elements
bool ContainsElement(const int& Z); // returns true if data for Z
```

The CEI method returns the ionization fractions for the specified element for a collisional ionization equilibrium plasma with the given electron temperature.

```
RealArray CEI(const Real& Te, const int& Z);
```

The method to calculate ion fractions is Calc. It is overloaded to give a number of different options. If the initIonFrac array is used then this gives the initial ion fractions for the element Z, if not the element is assumed to start un-ionized.

```
// calculate the NEI ion fractions for electron temperature Te, ionization
// parameter tau and element Z.

RealArray Calc(const Real& Te, const Real& tau, const int& Z);
RealArray Calc(const Real& Te, const Real& tau, const int& Z,
               const RealArray& initIonFrac);

// Calculates ionization fractions at electron temperatures
// Te and a set of ionization parameters tau(i), i=1,..,n,
// where each tau is given weight(i). Electron temperature is
```

```
  // assumed to be linear function of tau.
  // Based on the old noneq.f.

  RealArray Calc(const RealArray& Te, const RealArray& tau,
 const RealArray& weight, const int& Z);
  RealArray Calc(const RealArray& Te, const RealArray& tau,
 const RealArray& weight, const int& Z,
                 const RealArray& initIonFrac);

  // Calculates ionization fractions at electron temperature
  // Te(n) and ionization parameter tau(n), for electron
  // temperatures Te given in a tabular form as a function of
  // ionization parameter tau.
  // Based on the old noneqr.f.
  // Does initIonFrac make sense in this case.

  RealArray Calc(const RealArray& Te, const RealArray& tau,
 const int& Z);
  RealArray Calc(const RealArray& Te, const RealArray& tau,
 const int& Z, const RealArray& initIonFrac);

};
```

Outside the class there are wrap-up functions to read (if necessary) the eigenvector files and calculate ion fractions. They use the NEISPEC xset variable to choose which version of the files to use. The various overloaded versions match to the Calc methods.

```
void calcNEIfractions(const Real& Te, const Real& tau, const int& Z,
      RealArray& IonFrac);
void calcNEIfractions(const Real& Te, const Real& tau, const IntegerArray& Z,
      vector<RealArray>& IonFrac);

void calcNEIfractions(const Real& Te, const Real& tau, const int& Z,
      const RealArray& initIonFrac, RealArray& IonFrac);
void calcNEIfractions(const Real& Te, const Real& tau, const IntegerArray& Z,
      const vector<RealArray>& initIonFrac,
      vector<RealArray>& IonFrac);

void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const RealArray& weight, const int& Z, RealArray& IonFrac);
void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const RealArray& weight, const IntegerArray& Z,
      vector<RealArray>& IonFrac);

void calcNEIfractions(const RealArray& Te, const RealArray& tau,
```

```
      const RealArray& weight, const int& Z,
      const RealArray& initIonFrac, RealArray& IonFrac);
void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const RealArray& weight, const IntegerArray& Z,
      const vector<RealArray>& initIonFrac,
      vector<RealArray>& IonFrac);

void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const int& Z, RealArray& IonFrac);
void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const IntegerArray& Z, vector<RealArray>& IonFrac);

void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const int& Z, const RealArray& initIonFrac,
      RealArray& IonFrac);
void calcNEIfractions(const RealArray& Te, const RealArray& tau,
      const IntegerArray& Z, const RealArray& initIonFrac,
      vector<RealArray>& IonFrac);
```

The wrapper routine to return the collisional ionization equilibrium fractions is

```
void calcCEIfractions(const Real Te, const IntegerArray& Z,
                      vector<RealArray>& IonFrac);
```

There are also handy routines to return and index into the temperatures and the number of temperatures

```
int getNEItempIndex(const Real& tkeV);
int getNEInumbTemp();
```

to providing debugging information

```
string writeIonFrac(const IntegerArray& Zarray, const vector<RealArray>& IonFrac);
string writeIonFrac(const int& Z, const IntegerArray& Zarray,
    const vector<RealArray>& IonFrac);
```

to do a binary search on a RealArray and return the index of the element immediately less than the input target

```
int locateIndex(const RealArray& xx, const Real x);
```

and to check whether arrays are identical (note that the C++11 standard includes this as part of the valarray class but this is not yet implemented in all compilers.

```
bool identicalArrays(const vector<RealArray>& a, const vector<RealArray>& b);
bool identicalArrays(const RealArray& a, const RealArray& b);
bool identicalArrays(const IntegerArray& a, const IntegerArray& b);
```

3.4 Examples

A calcNEIfractions use can be found in the file vvgnei.cxx in the directory Xspec/src/XSFunctions.

Chapter 4

NeutralOpacity

This class serves as a limited C++ interface to the Fortran gphoto and photo routines to parallel the IonizedOpacity class. It will use the cross-sections set using the xsect command. The method Setup should be used to initialize the object then GetValue or Get to return opacities for a single or multiple energies, respectively. IronAbundance is used to set the iron abundance (relative to the defined Solar) and Abundance the abundances of all other elements. IncludeHHe specifies whether to include the contributions of hydrogen and helium in the total opacity.

```
class NeutralOpacity{
 public:

  IntegerArray AtomicNumber;
  vector<string> ElementName;

  string CrossSectionSource;

  NeutralOpacity();     // default constructor
  ~NeutralOpacity();    // destructor

  void Setup();   // set up opacities
  void Get(RealArray inputEnergy, Real Abundance, Real IronAbundance,
           bool IncludeHHe, RealArray& Opacity);  // return opacities
  void GetValue(Real inputEnergy, Real Abundance, Real IronAbundance,
                bool IncludeHHe, Real& Opacity);  // return single opacity

};
```

4.1 Examples

An example use of NeutralOpacity can be found in the routine calcCompReflTotalFlux in the file MZCompRefl.cxx in the directory Xspec/src/XSFunctions.

Chapter 5

IonizedOpacity

This class calculates opacity of a photo-ionized material. At present it uses the Reilman & Manson (1979) opacities although this could be generalized in future. The Setup method reads the input files if necessary and calculate ion fractions for the requested ionization parameter, temperature and spectrum. GetValue or Get then return opacities for a single or multiple energies, respectively. IronAbundance is used to set the iron abundance (relative to the defined Solar) and Abundance the abundances of all other elements. IncludeHHe specifies whether to include the contributions of hydrogen and helium in the total opacity.

```
class IonizedOpacity{
 public:

  IntegerArray AtomicNumber;
  vector<string> ElementName;

  RealArray Energy;

  RealArray **ion;
  RealArray **sigma;
  RealArray *num;

  IonizedOpacity();     // default constructor
  ~IonizedOpacity();    // destructor

  void LoadFiles();     // internal routine to load model data files
  void Setup(Real Xi, Real Temp, RealArray inputEnergy,
             RealArray inputSpectrum);   // set up opacities
  void Get(RealArray inputEnergy, Real Abundance, Real IronAbundance,
           bool IncludeHHe, RealArray& Opacity);  // return opacities
  void GetValue(Real inputEnergy, Real Abundance, Real IronAbundance,
                bool IncludeHHe, Real& Opacity);  // return single opacity
```

```
};
```

5.1 Examples

An example use of IonizedOpacity can be found in the routine calcCompReflTotalFlux in the file MZCompRefl.cxx in the directory Xspec/src/XSFunctions.

Chapter 6

MZCompRefl

This class calculates Compton reflection using the Magzdiarz and Zdziarski code.

```
class MZCompRefl{
 public:

  MZCompRefl();          // default constructor
  ~MZCompRefl();         // default destructor

  void CalcReflection(string RootName, Real cosIncl, Real xnor,
                      Real Xmax, RealArray& InputX, RealArray& InputSpec,
                      RealArray& Spref);
```

The easiest way to use the MZCompRefl class is through the associated function calcCompReflTotalFlux.

```
void calcCompReflTotalFlux(string ModelName, Real Scale, Real cosIncl,
                           Real Abund, Real FeAbund, Real Xi, Real Temp,
                           Real inXmax, RealArray& X, RealArray& Spinc,
                           RealArray& Sptot);
```

ModelName is the name of model and is used to check the ModelName_PRECISION xset variable to determine the precision to which internal integrals should be calculated.

Scale is the fraction of reflected emission to include. If Scale is zero then no reflection component is included, a value of one corresponds to an isotropic source above an infinite disk. The special case of minus one will return only the reflected component.

cosIncl is the cosine of the inclination angle of the disk to the line of sight. The iron abundance is specified by FeAbund and the abundances of all other elements by Abund.

For an ionized disk Xi and Temp give the ionization parameter and temperature for the IonizedOpacity class. If Xi is zero then the NeutralOpacity class is used instead. In both these cases the boolean IncludeHHe is set to false.

The input energies and spectrum are given by the X and Spinc arrays. The energies are in units of $m_e c^2$ and the input spectrum is EF_E. The output spectrum Sptot is also EF_E. The input variable inXmax is the maximum value of X for which the reflected spectrum is calculated. This is useful because X and Spinc should be specified to a higher energy than required in the output because the energy downscattering in Compton reflection. Setting inXmax to the highest output energy required will save computation time.

6.1 Examples

An example use of calcCompReflTotalFlux can be found in the routine doreflect in the file ireflct.cxx in the directory Xspec/src/XSFunctions.