PyAtomDB Documentation

Release 0.10.13

Adam Foster

Feb 06, 2023

CONTENTS

The AtomDB Project consists of a large atomic database designed for creating spectra of collisionally excited plasmas in the UV and X-ray wavebands for use in astronomy and astrophysics research. It has been successfully used for some time within several different spectral analysis suites, such as XSPEC and Sherpa.

There are two main parts to AtomDB:

The Astrophysical Plasma Emission Database (APED)
: A series of FITS files which store the atomic data necessary for modelling emission from collisional plasmas from elements from H through to Zn.

The Astrophysical Plasma Emission Code (APEC)
: Code which takes the APED and uses it to model a collisional plasma, creating emissivity files for line and continuum emission. These output files are then used in a range of models such as the `apec`, `nei` and `pshock` models amongst others.

Starting in 2015, PyAtomDB was developed to achieve a number of goals:

1. To replace the APEC code (formerly in C) with a more flexible tool for larger data sets
2. To allow interactive user access to the underlying database, which was always freely available but sometimes difficult to use
3. To enable creating more flexible spectra based on the outputs of AtomDB
4. To enhance the types of spectra which can be modeled in AtomDB, and astronomer interaction with these models
5. To facililtate inclusion of AtomDB models in other software (often python based).

CHAPTER

ONE

CONTENTS

Contents

1.1 Installation

Warning: PyAtomDB runs only under Python 3. It will not work on Python 2. If you need to install Python 3 in your system, you can use your package manager or there are many other sources which can help you including Anaconda.

Once you have Python 3 installed, you may (depending on your system) have to add a `3` to many of the command line commands, e.g. `python` becomes `python3`, or `pip` becomes `pip3`. Commands in this guide omit the `3`. Note that actual Python code is unaffected.

PyAtomDB can be installed in two ways:

1. From PyPI , using the simple `pip install pyatomdb` command.
2. From GitHub , using the command `git clone https://github.com/AtomDB/pyatomdb.git` to get the source, then `pip install -e python </path/to/folder/with/setup.py/in/it>`.
3. If using `Conda`: Pyatomdb is not packaged with Conda as it requires compiling of some C code and (as far as I can tell) Conda cannot handle this. I recommend installing the dependencies independently, e.g.: `conda install requests wget "numpy>=1.9.0" "scipy>=1.9.0" joblib mock astropy pycurl`, then install pip within conda (`conda install pip`), and then `pip install -e python </path/to/folder/with/setup.py/in/it>`.
4. Using `setuptools` (deprecated): python setup.py install

Note that for the PyPI and setuptools options the `--user` option can be useful, as it will install software in your local path if you do not have administrator priviledges on your machine.

You can check that the installation was successful by running:

```
>>> import pyatomdb
```

If it does not immediately throw out an error, it has been successful. It will then start asking about installing the AtomDB files, see the next section. Note that there is no longer a need to run the initialize script.

Warning: pycURL issues can arise when installing. If your install above worked without errors, then you are fine. If you encountered errors, and they are related to pycurl, you will need to consult your system's package manager (or conda if that is what you are using) and install pycurl separately - e.g. `conda install pycurl`.

I'm not sure why this refuses to install directly with pip sometimes, but it seems to be a recurring feature.

1.1.1 ATOMDB Directory

Whenever you import the PyAtomDB module, it performs a check for the $ATOMDB directory. This directory is where the AtomDB data files will be stored. These are not distributed with the python package as they are large and most people will only need a few. PyAtomDB will download the APED data files on demand as you require them, and they are then stored in this directory until you manually delete them. If you need to recover disk space, you can delete anything in the $ATOMDB/APED directory without repercussions - PyAtomDB will re-download the files if it needs them in the future.

You will be asked to select a directory for installation and then whether to download the emissivity files. It is important that this directory is one where you have write access, as in the future further files will be added there by the code automatically.

Once installation is complete, ensure that you add the ATOMDB variable to your shell startup file. Assuming you have installed into /home/username/atomdb, in bash, add this line to your ~/.bashrc or ~/.bash_profile files (depends on which one is sourced by your system):

```
export ATOMDB=/home/username/atomdb
```

or for csh, add this to your ~/.cshrc or ~/.cshrc.login:

```
setenv ATOMDB /home/username/atomdb
```

Recent version of Mac OS have moved to zsh, in which case modify your ~/.zshrc file as for bash above.

1.1.2 Usage Data Collection

You will also be asked about anonymous usage data. In order to track roughly how many people are using PyAtomDB, a randomly generated number is created when you install PyAtomDB and stored in your `$ATOMDB/userdata` file. Whenever PyAtomDB has to fetch a new file this number, the filename and the current timestamp is stored on our system so we can estimate how many users there are. We have no way to connect this to actual individuals, it simply tells us roughly how many unique active users there are.

If you decline, this number is set to 00000000, and otherwise PyAtomDB functions as normal.

1.2 PyAtomDB Example Scripts

1.2.1 Module Structure

These are examples of using the PyAtomDB module in your projects. They can all be found in the examples subdirectory of your PyAtomDB installation.

The examples here are separated by the module in which they are implemented. Generally speaking, the roles of these modules can be split into the following areas:

spectrum
: Obtaining emissivities from the AtomDB emissivity files (e.g. the `apec` model). This module is used to create spectra as required.

apec
: Related to the APEC code - creating the AtomDB emissivity files from the underlying atomic database

atomdb
: Accessing the atomic database - returning rates and coefficients, fetching files

atomic
: Basic atomic functions - getting atomic masses, converting to element symbols etc.

util
: File utilities (not generally interesting to end users)

const
: List of physical constants and code-related constants used throughout PyAtomDB (again, not generally interesting to end users)

1.2.2 Spectra

The spectrum.py module contains routines for taking the spectrum generated from the apec model and extracting line emissivities or continuum emission, applying responses, changing abundances, etc. In these examples, we will use the Chandra ACIS-S MEG +1 order grating responses as examples where one is required, but you can use any.

Similarly, we use `pylab` for plotting. You can of course use whatever system you like for plotting, this is just what we did. It is included in matplotlib and therefore hopefully present on most systems.

1.2.2.1 CIESession Class

The heart of the spectral analysis is the spectrum.py class. This reads in the results of an apec run (by default, $ATOMDB/apec_line.fits and $ATOMDB/apec_coco.fits) and allows the user to obtain spectra at a range of temperatures accounting for instrument responses, thermal and velocity broadening, abundance changes and other issues.

Making a Spectrum

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()

# create a set of energy bins (in keV) for the response. Note these are
# the n edges of the n-1 bins.
ebins = numpy.linspace(0.3,1.0,10000)

# set the response (raw keyword tells pyatomdb it is not a real response file)
sess.set_response(ebins, raw=True)

kT = 0.4 # temperature in keV
spec = sess.return_spectrum(kT)

# we now have a spectrum and the energy bins,

# prepend a zero to the spectrum so that energy bins and spectrum have
# the same length. If you use a different plotting system you may
# need to add this to the end.
spec = numpy.append(0, spec)

# Returned spectrum has units of photons cm^5 s^-1 bin^-1
fig = pylab.figure()
fig.show()
ax = fig.add_subplot(211)
ax.plot(sess.ebins_out, spec, drawstyle='steps', label='dummy response')

# label the figure
ax.set_xlabel('Energy (keV)')
ax.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')

# zoom in a bit
#ax.set_xlim([0.8,0.85])

# make plot appear

# now repeat the process with a real response
# set response
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# get spectrum, prepend zero
spec = sess.return_spectrum(kT)
spec = numpy.append(0, spec)

ax2 = fig.add_subplot(212)
ax2.plot(sess.ebins_out, spec, drawstyle='steps', label='Chandra MEG')
```

(continues on next page)

(continued from previous page)

```
# add legends
ax.legend(loc=0)
ax2.legend(loc=0)

# zoom in on small sections of the spectrum
ax.set_xlim([0.7, 0.9])
ax2.set_xlim([0.7, 0.9])

# set axes
ax2.set_xlabel('Energy (keV)')
ax2.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')

# adjust plot spacing so labels are visible
pylab.matplotlib.pyplot.subplots_adjust(hspace = 0.34)

# draw graphs
pylab.draw()

zzz=input("Press enter to continue")

# save image files
fig.savefig('spectrum_session_examples_1_1.pdf')
fig.savefig('spectrum_session_examples_1_1.svg')
```

Fig. 1: A kT=0.4keV simple spectrum created with and without an instrument response

Handling Large Response Matrices

Some missions, for example XRISM, have a large number of bins in their response files (XRISM has 50,000). Creating a 50,000x50,000 response matrix would require of order 16GB of memory, most of which would be zeros. Setting `sparse=True` when setting the response will use sparse matrices to save memory. The code below shows how to do this and also how to compare any innacuracies caused (testing so far has shown nothing larger than numerical rounding errors, 1 part in 10^{15})

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()


kT = 0.4 # temperature in keV

# Returned spectrum has units of photons cm^5 s^-1 bin^-1
```

(continues on next page)

(continued from previous page)

```
# create figure
gs = pylab.matplotlib.gridspec.GridSpec(2, 1, height_ratios=[2, 1])
fig = pylab.figure()
fig.show()
ax1 = fig.add_subplot(gs[0]) # for data
ax2 = fig.add_subplot(gs[1], sharex=ax1) # for ratio

# label the figure
ax2.set_xlabel('Energy (keV)')
ax1.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')
ax2.set_ylabel('Ratio')

# set response - using a regular matrix
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# get spectrum, prepend zero
spec = sess.return_spectrum(kT)
spec = numpy.append(spec[0], spec)

ax1.plot(sess.ebins_out, spec, drawstyle='steps', label='Regular')

# change the response to a sparse one
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf',
                  sparse=True)
spec_sparse = sess.return_spectrum(kT)
spec_sparse = numpy.append(spec_sparse[0], spec_sparse)

ax1.plot(sess.ebins_out, spec_sparse, '--', drawstyle='steps', label='Sparse')

# plot the ratio
ax2.plot(sess.ebins_out, spec/spec_sparse, drawstyle='steps',\
         label='Regular/Sparse')

# add legend
ax1.legend(loc=0)
ax2.legend(loc=0)

# zoom in on small sections of the spectrum
ax1.set_xlim([0.5, 2.0])

# adjust plot spacing so labels are visible
pylab.matplotlib.pyplot.subplots_adjust(hspace = 0)

# draw graphs
pylab.draw()

zzz=input("Press enter to continue")

# save image files
fig.savefig('spectrum_session_examples_1b_1.pdf')
fig.savefig('spectrum_session_examples_1b_1.svg')
```

Fig. 2: A kT=0.4keV simple spectrum created with regular and sparse instrument responses. The same code can be used to compare the accuracy of the two methods for different response matrices, but will require large amounts of memory if using large responses.

Line Broadening

By default, line broadening is off. The command `session.set_broadening` allows you to turn on thermal broadening and, if desired, add additional turbulent velocity broadening too.

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()

# create a set of energy bins (in keV) for the response. Note these are
# the n edges of the n-1 bins.
ebins = numpy.linspace(1.0,1.2,501)

# set the response (raw keyword tells pyatomdb it is not a real response file)
sess.set_response(ebins, raw=True)

kT = 4.0 # temperature in keV
vel = 400.0 # velocity to broaden with when using velocity broadening (km/s)

spec = sess.return_spectrum(kT)

# we now have a spectrum and the energy bins,

# prepend a zero to the spectrum so that energy bins and spectrum have
# the same length. If you use a different plotting system you may
# need to add this to the end.
spec = numpy.append(0, spec)

# Returned spectrum has units of photons cm^5 s^-1 bin^-1
fig = pylab.figure()
fig.show()
ax = fig.add_subplot(211)
ax.plot(sess.ebins_out, spec, drawstyle='steps', label='dummy response')

# label the figure
ax.set_xlabel('Energy (keV)')
ax.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')



sess.set_broadening(True)
spec = sess.return_spectrum(kT)
spec = numpy.append(0, spec)
```

(continues on next page)

(continued from previous page)

```
ax.plot(sess.ebins_out, spec, drawstyle='steps', label='thermal')

sess.set_broadening(True, velocity_broadening=vel)
spec = sess.return_spectrum(kT)
spec = numpy.append(0, spec)
ax.plot(sess.ebins_out, spec, drawstyle='steps', label='%.0f km/s'%(vel))

# make plot appear

# now repeat the process with a real response

# reset broadening to off.
sess.set_broadening(False)

# set response
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# get spectrum, prepend zero
spec = sess.return_spectrum(kT)
spec = numpy.append(0, spec)

ax2 = fig.add_subplot(212)
ax2.plot(sess.ebins_out, spec, drawstyle='steps', label='Chandra MEG')

# add legends
ax.legend(loc=0)
ax2.legend(loc=0)

# zoom in on small sections of the spectrum
ax.set_xlim([1.150, 1.180])
ax2.set_xlim([1.150, 1.180])

# set axes
ax2.set_xlabel('Energy (keV)')
ax2.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')

sess.set_broadening(True)
spec = sess.return_spectrum(kT)
spec = numpy.append(0, spec)
ax2.plot(sess.ebins_out, spec, drawstyle='steps', label='thermal')

sess.set_broadening(True, velocity_broadening=vel)
spec = sess.return_spectrum(kT)
spec = numpy.append(0, spec)
ax2.plot(sess.ebins_out, spec, drawstyle='steps', label='%.0f km/s'%(vel))


# adjust plot spacing so labels are visible
pylab.matplotlib.pyplot.subplots_adjust(hspace = 0.34)

# draw graphs
pylab.draw()
```

(continues on next page)

(continued from previous page)

```
zzz=input("Press enter to continue")

# save image files
fig.savefig('spectrum_session_examples_2_1.pdf')
fig.savefig('spectrum_session_examples_2_1.svg')
```

Fig. 3: A kT=3.0keV spectrum unbroadened, thermally broadened and then additionally velocity broadend.

Changing Abundances

There are several ways to change the abundances. By default, all are set to 1.0 times the solar value of Anders and Grevesse. The `session.set_abund` command implements this.

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()

# set response
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# for plotting
fig = pylab.figure()
fig.show()
ax = fig.add_subplot(111)

# make a spectrum at 1keV
s1 = sess.return_spectrum(1.0)

# change the abundance of Iron
sess.set_abund(26,2.0)
s2 = sess.return_spectrum(1.0)
# change the abundance of Magnesium and Silicon
sess.set_abund([12,14],0.5)
s3 = sess.return_spectrum(1.0)
# change the abundance of Neon and Iron separately
sess.set_abund([10,26],[2.0,0.5])
s4 = sess.return_spectrum(1.0)
# plot all this
offset = max(s1)
ax.plot(sess.ebins_out, numpy.append(0,s1), drawstyle='steps', label='Original')
ax.plot(sess.ebins_out, numpy.append(0,s2)+offset, drawstyle='steps', label='Fex2')
ax.plot(sess.ebins_out, numpy.append(0,s3)+offset*2, drawstyle='steps', label='Mg, Six0.5
↪')
```

(continues on next page)

(continued from previous page)

```
ax.plot(sess.ebins_out, numpy.append(0,s4)+offset*3, drawstyle='steps', label='Nex2,␣
↪Fex0.5')
ax.legend(ncol=4, loc=0)
ax.set_xlim([0.8,1.5])
ax.set_xlabel('Energy (keV)')
ax.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')
pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('spectrum_session_examples_3_1.pdf')
fig.savefig('spectrum_session_examples_3_1.svg')
```

Fig. 4: A kT=1.0keV spectrum with assorted different abundances applied.

You can also change the entire abundance set in use using `session.set_abundset`.

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()

# set response
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# for plotting
fig = pylab.figure()
fig.show()
ax = fig.add_subplot(111)

# get a list of the available abundance sets
sess.set_abundset()
# this prints them to screen.



for iabset, abset in enumerate(['Allen', 'AG89', 'GA88', 'Feldman', 'GA10', 'Lodd09',
↪'AE82']):

  # set abundset
  sess.set_abundset(abset)

  # get spectrum at 1keV
  s1 = sess.return_spectrum(1.0)
  if iabset == 0:
    baseoffset = max(s1)

  offset = iabset*baseoffset
```

(continues on next page)

(continued from previous page)

```
  #plot
  ax.plot(sess.ebins_out, numpy.append(0,s1)+offset, drawstyle='steps', label=abset)

ax.set_xlim([0.8,2.1])

ax.set_xlabel('Energy (keV)')
ax.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')
ax.legend(loc=0, ncol=4)
pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('spectrum_session_examples_4_1.pdf')
fig.savefig('spectrum_session_examples_4_1.svg')
```

Fig. 5: A kT=1.0keV spectrum with assorted different abundance sets applied.

Return line list

To obtain a list of lines and their emissivities in a spectral range, use `session.return_linelist`. This returns the data as a numpy array of the lines in the region, applying all current abundance information to the linelist. It also interpolates in temperature between the two nearest points. Note that since AtomDB cuts off lines with emissivities below 10^{-20} $ph\ cm^3 s^{-1}$, lines will diappear below this emissivity. Lines beyond the last temperature point at which they are tabulated will not be included in the returned linelist.

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()

# set response
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# get a linelist between 1.85 and 1.88 Angstroms for a kT=4keV plasma

llist = sess.return_linelist(4.0,[1.85,1.88])

# select only the strongest 10 lines
llist.sort(order='Epsilon')
llist = llist[-10:]

# show the column names
print(llist.dtype.names)
```

(continues on next page)

(continued from previous page)

```
# expected output:
# ('Lambda', 'Lambda_Err', 'Epsilon', 'Epsilon_Err', 'Element', 'Ion', 'UpperLev', 'LowerLev')

# print the data
print(llist)

# expected output:
#[(1.8635   , nan, 8.8202775e-19, nan, 26, 24,    47, 1)
# (1.8532   , nan, 1.0113579e-18, nan, 26, 24, 10089, 6)
# (1.8622   , nan, 1.0681659e-18, nan, 26, 24, 10242, 3)
# (1.863    , nan, 2.5785468e-18, nan, 26, 24, 10247, 2)
# (1.8611   , nan, 2.8751725e-18, nan, 26, 24,    48, 1)
# (1.8659   , nan, 3.8414809e-18, nan, 26, 24, 10248, 3)
# (1.8554125, nan, 7.2097873e-18, nan, 26, 25,     6, 1)
# (1.8595169, nan, 7.7184372e-18, nan, 26, 25,     5, 1)
# (1.8681941, nan, 1.2001933e-17, nan, 26, 25,     2, 1)
# (1.8503995, nan, 3.5732330e-17, nan, 26, 25,     7, 1)]

# we can then do the same thing, but including the effective area of the instrument.
# So the Epsilon columnw will be multiplied by the effective area at the line's energy
llist = sess.return_linelist(4.0,[1.85,1.88], apply_aeff=True)

llist.sort(order='Epsilon')
llist = llist[-10:]

print('After apply effective area:')
print(llist)
#[(1.8635   , nan, 1.6238816e-18, nan, 26, 24,    47, 1)
# (1.8532   , nan, 1.7191134e-18, nan, 26, 24, 10089, 6)
# (1.8622   , nan, 1.9665764e-18, nan, 26, 24, 10242, 3)
# (1.863    , nan, 4.7473049e-18, nan, 26, 24, 10247, 2)
# (1.8611   , nan, 5.2934158e-18, nan, 26, 24,    48, 1)
# (1.8659   , nan, 7.3193854e-18, nan, 26, 24, 10248, 3)
# (1.8554125, nan, 1.2779108e-17, nan, 26, 25,     6, 1)
# (1.8595169, nan, 1.3680673e-17, nan, 26, 25,     5, 1)
# (1.8681941, nan, 2.2867946e-17, nan, 26, 25,     2, 1)
# (1.8503995, nan, 6.0738072e-17, nan, 26, 25,     7, 1)]
```

Return line emissivity

To calculate the emissvitiy of a specific line, you can use `session.return_line_emissivity` along with the ion, element, upper and lower levels of the transition. You can supply a single or a range of temperatures, and a dictionary will be returned containing much of the information along with emissivity (epsilon) you requested.

```
import pyatomdb
import numpy
import pylab


# declare the Collisional Ionization Equilibrium session
sess = pyatomdb.spectrum.CIESession()
```

(continues on next page)

(continued from previous page)

```
# set response
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

# get the emissivity of the resonance line of O VII at 5 different temperatures

kTlist = numpy.array([0.1,0.3,0.5,0.7,0.9])
Z=8
z1=7
up=7
lo = 1
ldata = sess.return_line_emissivity(kTlist, Z, z1, up, lo)

fig = pylab.figure()
fig.show()
ax= fig.add_subplot(111)

ax.semilogy(ldata['Te'], ldata['epsilon'])

ax.set_xlabel("Temperature (keV)")
ax.set_ylabel("Epsilon ph cm$^3$ s$^{-1}$")

pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('spectrum_session_examples_6_1.pdf')
fig.savefig('spectrum_session_examples_6_1.svg')
```

Fig. 6: Emissivity of the resonance line of O VII.

1.2.2.2 NEISession Class

Derived from the CIESession class, this handles non-equilibrium spectra. As such, all of the calls to it are exactly the same. Adding response, setting abundances obtaining spectra, etc all work the same way. Therefore I will only outline what is different.

The main difference is that a non equilibrium plasma requires 2 more pieces of information to define it. The current electron temperature is as with the CIE case. The other two are:

tau
: The ionization timescale ($n_e t$) in cm^{-3} s^{1} since the plasma left its previous equilibrium

init_pop
: The initial population the plasma was in at tau=0.

Tau is a single number in all cases. init_pop can be be defined in a range of ways:

float
: If it is a single number, this is assumed to be an electron temperature. The ionization fraction will be calculated at this T_e.

dict
: If a dictionary is provided, it should be an explicit ionization fraction for each element. So dict[6]=

[0.0, 0.1, 0.2, 0.3, 0.3, 0.1, 0.0] would be the ionization fraction for carbon, dict[8]= [0.0, 0.0, 0.1, 0.1, 0.1, 0.15,0.3,0.2,0.05] would be for oxygen etc.

'ionizing'
If the string ionizing is provided, set all elements to be entirely neutral. This is the default.

'recombining'
If the string recombining is provided, set all elements to be fully ionized.

Making a Spectrum

As an example, this will plot a simple recombining spectrum within initial temperature of 1.0keV.

```
import pyatomdb
import numpy
import pylab


# declare the Non-Equilibrium Ionization session. I am only using
# oxygen and neon for simplicity
sess = pyatomdb.spectrum.NEISession(elements=[8,10])

# set the response (raw keyword tells pyatomdb it is not a real response file)
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')

kT = 0.4 # temperature in keV
tau = 1e10 # n_e * t in cm^-3 s
spec = sess.return_spectrum(kT, tau, init_pop=1.0)

# Returned spectrum has units of photons cm^5 s^-1 bin^-1
fig = pylab.figure()
fig.show()
ax = fig.add_subplot(111)

offset = max(spec)
ax.plot(sess.ebins_out, numpy.append(0,spec), drawstyle='steps', label='mild recombining
↪')


spec = sess.return_spectrum(kT, tau, init_pop='ionizing')
ax.plot(sess.ebins_out, offset+numpy.append(0,spec), drawstyle='steps', label='ionizing')

spec = sess.return_spectrum(kT, tau, init_pop='recombining')
ax.plot(sess.ebins_out, 2*offset+numpy.append(0,spec), drawstyle='steps', label=
↪'recombining')



# manually specified initial population
init_pop_dict = {}
init_pop_dict[8] = numpy.zeros(9) # 9 stages of oxygen (include neutral an fully␣
↪stripped)
init_pop_dict[10] = numpy.zeros(11)
```

(continues on next page)

(continued from previous page)

```
#put everything in the H-like stage
init_pop_dict[8][8] = 1.0
init_pop_dict[10][10] = 1.0


spec = sess.return_spectrum(kT, tau, init_pop=init_pop_dict)
ax.plot(sess.ebins_out, 3*offset+numpy.append(0,spec), drawstyle='steps', label='all H-
↪like')


# label the figure
ax.set_xlabel('Energy (keV)')
ax.set_ylabel('Intensity (ph cm$^5$ s$^{-1}$ bin$^{-1}$)')

# zoom in a bit
ax.set_xlim([0.6,1.2])
ax.legend(loc=0)
# make plot appear
pylab.draw()

zzz=input("Press enter to continue")

# save image files
fig.savefig('spectrum_session_examples_7_1.pdf')
fig.savefig('spectrum_session_examples_7_1.svg')
```

Fig. 7: Recombining spectrum specified by an initial temperature

Making a Linelist

This is again exactly the same as the CIESession case, except with extra parameters `tau` and `init_pop`. The only additional option is the ability to separate the linelist by driving ion. This is the ion which gives rise to the emission. For example, if the emission is largely from ionization into an excited state and then subsequent cascade, then that part of the emissivity has a different driving ion than that driven by excitation of the ion's ground state.

Separating out these features can be turned on and off using the `by_ion_drv` keyword,

```
import pyatomdb
import numpy
import pylab


# declare the Non-Equilibrium Ionization session. I am only using
# oxygen and neon for simplicity
sess = pyatomdb.spectrum.NEISession()

# set the response (raw keyword tells pyatomdb it is not a real response file)
sess.set_response('aciss_meg1_cy22.grmf', arf = 'aciss_meg1_cy22.garf')
```

(continues on next page)

(continued from previous page)

```
kT = 4.0 # temperature in keV
tau = 1e11 # n_e * t in cm^-3 s
llist = sess.return_linelist(kT, tau, [1.7,1.9],init_pop=1.0)

# filter lines to strongest 20
llist.sort(order='Epsilon')
llist= llist[-20:]

print("strongest 20 lines")
print(llist.dtype.names)
print(llist)

llist = sess.return_linelist(kT, tau, [1.7,1.9],by_ion_drv=True,init_pop=1.0)

# filter lines to strongest 20
llist.sort(order='Epsilon')
llist= llist[-20:]

print("Strongest 20 lines, separated by driving ion")
print(llist.dtype.names)
print(llist)


# Expected output
# strongest 20 lines
# ('Lambda', 'Lambda_Err', 'Epsilon', 'Epsilon_Err', 'Element', 'Ion', 'UpperLev', 'LowerLev')
# [(1.8827   , nan, 4.3892664e-19, nan, 26, 22, 20009, 1)
#  (1.8622   , nan, 4.5014698e-19, nan, 26, 24, 10242, 3)
#  (1.8824   , nan, 4.8421002e-19, nan, 26, 22,   483, 1)
#  (1.8825   , nan, 5.8088691e-19, nan, 26, 22,   482, 1)
#  (1.8851   , nan, 5.9669826e-19, nan, 26, 22, 20008, 2)
#  (1.8747   , nan, 6.6686722e-19, nan, 26, 24,    44, 1)
#  (1.8737   , nan, 6.9597891e-19, nan, 26, 23, 20126, 3)
#  (1.863    , nan, 1.0866564e-18, nan, 26, 24, 10247, 2)
#  (1.8756001, nan, 1.0874650e-18, nan, 26, 23, 20122, 4)
#  (1.8706   , nan, 1.3853237e-18, nan, 26, 24,    46, 1)
#  (1.8738   , nan, 1.4878691e-18, nan, 26, 24,    45, 1)
#  (1.8659   , nan, 1.6188786e-18, nan, 26, 24, 10248, 3)
#  (1.857    , nan, 1.9076071e-18, nan, 26, 24,    50, 1)
#  (1.8554125, nan, 2.3919818e-18, nan, 26, 25,     6, 1)
#  (1.8595169, nan, 2.8080335e-18, nan, 26, 25,     5, 1)
#  (1.8635   , nan, 3.6176967e-18, nan, 26, 24,    47, 1)
#  (1.8704   , nan, 7.2666667e-18, nan, 26, 23,   309, 1)
#  (1.8681941, nan, 8.8379641e-18, nan, 26, 25,     2, 1)
#  (1.8611   , nan, 1.1892281e-17, nan, 26, 24,    48, 1)
#  (1.8503995, nan, 1.4463351e-17, nan, 26, 25,     7, 1)]
# Strongest 20 lines, separated by driving ion
# ('Lambda', 'Lambda_Err', 'Epsilon', 'Epsilon_Err', 'Element', 'Elem_drv', 'Ion', 'Ion_drv',
↪'UpperLev', 'LowerLev')
# [(1.8622   , nan, 4.5014698e-19, nan, 26, 26, 24, 25, 10242, 3)
#  (1.8824   , nan, 4.8421002e-19, nan, 26, 26, 22, 22,   483, 1)
#  (1.8825   , nan, 5.6233666e-19, nan, 26, 26, 22, 22,   482, 1)
```

(continues on next page)

(continued from previous page)

```
#  (1.8851   , nan, 5.9669826e-19, nan, 26, 26, 22, 23, 20008, 2)
#  (1.8747   , nan, 6.5364458e-19, nan, 26, 26, 24, 24,    44, 1)
#  (1.8737   , nan, 6.9597891e-19, nan, 26, 26, 23, 24, 20126, 3)
#  (1.863    , nan, 1.0866564e-18, nan, 26, 26, 24, 25, 10247, 2)
#  (1.8756001, nan, 1.0874650e-18, nan, 26, 26, 23, 24, 20122, 4)
#  (1.8706   , nan, 1.3512313e-18, nan, 26, 26, 24, 24,    46, 1)
#  (1.8738   , nan, 1.4619381e-18, nan, 26, 26, 24, 24,    45, 1)
#  (1.8659   , nan, 1.6188786e-18, nan, 26, 26, 24, 25, 10248, 3)
#  (1.857    , nan, 1.8978882e-18, nan, 26, 26, 24, 24,    50, 1)
#  (1.8554125, nan, 2.3303760e-18, nan, 26, 26, 25, 25,     6, 1)
#  (1.8595169, nan, 2.7651213e-18, nan, 26, 26, 25, 25,     5, 1)
#  (1.8681941, nan, 3.3107048e-18, nan, 26, 26, 25, 25,     2, 1)
#  (1.8635   , nan, 3.6059715e-18, nan, 26, 26, 24, 24,    47, 1)
#  (1.8681941, nan, 5.3853784e-18, nan, 26, 26, 25, 24,     2, 1)
#  (1.8704   , nan, 7.2068723e-18, nan, 26, 26, 23, 23,   309, 1)
#  (1.8611   , nan, 1.1865456e-17, nan, 26, 26, 24, 24,    48, 1)
#  (1.8503995, nan, 1.4405425e-17, nan, 26, 26, 25, 25,     7, 1)]
#
```

1.2.3 Getting Atomic Data

The `atomdb` module is designed to read data from the database and either return raw opened files (e.g. all the energy levels of an ion) or individual useful data (e.g. a rate coefficient).

AtomDB stores data sorted by ion in a series of files in the `$ATOMDB/APED`. There are several files for each ion, covering different data types:

- 'IR' - ionization and recombination
- 'LV' - energy levels
- 'LA' - radiative transition data (lambda and A-values)
- 'EC' - electron collision data
- 'PC' - proton collision data
- 'DR' - dielectronic recombination satellite line data
- 'PI' - XSTAR photoionization data
- 'AI' - autoionization data
- 'ALL' - reads all of the above. Does not return anything. Used for bulk downloading.

Or, for non-ion-specific data (abundances and bremstrahlung coeffts):

- 'ABUND' - abundance tables
- 'HBREMS' - Hummer bremstrahlung coefficients
- 'RBREMS' - relativistic bremstrahlung coefficitients
- 'IONBAL' - ionization balance tables
- 'EIGEN' - eigenvalue files (for non-equilibrium ionization)

The raw atomic data files can be downloaded using `atomdb.get_data`.

Note: Datacache

Often, when using AtomDB data you will want to use the same file over and over again. However, the whole database is too large to be loaded into memory as a matter of course. To get around this, the `datacache` keyword is used in several routines. This is a dictionary which is populated with any data files that you have to open. The atomdb function which ultimately opens the raw data files, `get_data`, will always check for an already existing data file here before going to the disk to open the file anew. If there is no data there, then the file is opened and its data is stored in the datacache so that next time the routine is called it will already be there.

If you need to save memory, you can empty the cache by declaring the datacache as a new dictionary, i.e. `datacache={}`.

```
import pyatomdb, time

# Example of using get data. I am going to get a range of different atomic data
#
# This routine returns the raw opened FITS data. No other processing is done.

Z = 8
z1 = 7

# get the energy levels of O VII

a = pyatomdb.atomdb.get_data(Z,z1,'LV')

# print out the first few lines
# note that for all the atomic data files, the useful data is in HDU 1.

print(a[1].data.names)

print(a[1].data[:5])

# get radiative transition information
b = pyatomdb.atomdb.get_data(Z,z1,'LA')

# find the transitions from level 7

bb = b[1].data[ (b[1].data['UPPER_LEV']==7)]
print()
print(bb.names)
print(bb)

# and so on.

# For non ion-specific data, Z and z1 are ignored.

abund = pyatomdb.atomdb.get_data(0,0,'ABUND')
print()
print(abund[1].data.names)
print(abund[1].data[:2])
```

(continues on next page)

(continued from previous page)

```
# How the datacache works
# The datacache variable stores opened files in memory, preventing
# having to go and re-open them each time. This reduces disk access.

# make sure file is downloaded for a fair test:
a = pyatomdb.atomdb.get_data(Z,z1,'EC')

# case 1: no datacache
t1 = time.time()
for i in range(10):
  a = pyatomdb.atomdb.get_data(Z,z1,'EC')
t2 = time.time()

# case 2: using datacache
datacache = {}
t3 = time.time()
for i in range(10):
  a = pyatomdb.atomdb.get_data(Z,z1,'EC', datacache=datacache)
t4 = time.time()

print("Time without / with datacache: %f / %f seconds"%(t2-t1, t4-t3))
```

1.2.3.1 Getting Rate Coefficients

Ionization, recombination, and excitation rate coefficients can all be obtained using `get_maxwell_rate`.

```
import pyatomdb, numpy, pylab

# We will get some maxwellian rates for a O 6+ 7->1

Z = 8
z1 = 7
up = 7
lo = 1

# excitation

Te = numpy.logspace(6, 7, 15)

#  >>> # (1) Get excitation rates for row 12 of an Fe XVII file
#  >>> colldata = pyatomdb.atomdb.get_data(26,17,'EC')
#  >>> exc, dex = get_maxwell_rate(Te, colldata=colldata, index=12)

#  >>> # (2) Get excitation rates for row 12 of an Fe XVII file
#  >>> exc, dex = get_maxwell_rate(Te, Z=26,z1=17, index=12)

#  >>>  (3) Get excitation rates for transitions from level 1 to 15 of FE XVII
#  >>> exc, dex = get_maxwell_rate(Te, Z=26, z1=17, dtype='EC', finallev=15, initlev=1)

datacache = {}
```

(continues on next page)

(continued from previous page)

```
# get data by specifying ion, upper and lower levels
exc, dexc = pyatomdb.atomdb.get_maxwell_rate(Te, Z=Z, z1=z1, initlev = lo, finallev=up,␣
↪dtype='EC', datacache=datacache)

fig= pylab.figure()
fig.show()
ax = fig.add_subplot(111)
ax.loglog(Te, exc, label='excitation')
ax.loglog(Te, dexc, label = 'deexcitation')


# preload data and find a specific transition
ecdat = pyatomdb.atomdb.get_data(8,7,'EC', datacache=datacache)

i = numpy.where( (ecdat[1].data['Upper_Lev']==up) &\
                 (ecdat[1].data['Lower_Lev']==lo))[0][0]

exc, dex = pyatomdb.atomdb.get_maxwell_rate(Te,colldata=ecdat, index=i,␣
↪datacache=datacache)

ax.loglog(Te, exc, 'o', label='excitation')
ax.loglog(Te, dexc, 'o', label = 'deexcitation')

ax.legend(loc=0)
ax.set_xlabel("Temperature (K)")
ax.set_ylabel("Rate Coefficient (cm$^3$ s$^{-1}$)")

pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('atomdb_examples_2_1.pdf')
fig.savefig('atomdb_examples_2_1.svg')

# you can also obtain ionization or recombination rates (see get_maxwell_rates writeup␣
↪for options)
# in most cases you need to specify initlev and finallev as 1 to get the ion to ion␣
↪rates.

ion= pyatomdb.atomdb.get_maxwell_rate(Te,Z=Z, z1=z1, initlev=1, finallev=1, dtype='CI',␣
↪datacache=datacache)
print(ion)
#
# However it is recommended that you use get_ionrec_rate instead - it gives you␣
↪everythign in one go

# combining the different types
ion, rec = pyatomdb.atomdb.get_ionrec_rate(Te, Z=Z, z1=z1, datacache=datacache)

# as separate entities
CI, EA, RR, DR=pyatomdb.atomdb.get_ionrec_rate(Te, Z=Z, z1=z1, datacache=datacache,␣
↪separate=True)
```

(continues on next page)

(continued from previous page)

```
print("Collisional ionization: ",CI)
print("Excitation Autoionization: ",EA)
print("Radiative Recombination: ",RR)
print("Dielectroni Recombination: ",DR)
ax.cla()

ax.loglog(Te, ion, label='Ionization')
ax.loglog(Te, rec, label='Recombination')

ax.loglog(Te, CI, '--',label='CI')
ax.loglog(Te, EA, '--',label='EA')
ax.loglog(Te, RR, '--',label='RR')
ax.loglog(Te, DR, '--',label='DR')

ax.legend(loc=0)
ax.set_xlabel("Temperature (K)")
ax.set_ylabel("Rate Coefficient (cm$^3$ s$^{-1}$)")

pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('atomdb_examples_2_2.pdf')
fig.savefig('atomdb_examples_2_2.svg')


#get_maxwell_rate(Te, colldata=False, index=-1, lvdata=False, Te_unit='K', \
#                     lvdatap1=False, ionpot = False, \
#                     force_extrap=False, silent=True,\
#                     finallev=False, initlev=False,\
#                     Z=-1, z1=-1, dtype=False, exconly=False,\
#                     datacache=False, settings=False, ladat=False):
```

Fig. 8: Excitation rates for O^{6+}.

Fig. 9: Ionization and recombination rates for O^{6+} to O^{7+}. Note that for the components breakdown, there is no excitation autionization contribution in the files.

1.2.4 Calculating Plasma Conditions

The apec module is used for calculating plasma related phenomena. For example, ionization fractions, non-equilibrium ionization, level populations, emission spectra. Currently, it is mostly written in such a way that it is difficult to use it outside of running the entire APEC code. A rewrite will happen soon. However, for now, here are a couple of useful routines which should be useable.

1.2.4.1 Getting Charge State Distribution

```
import pyatomdb, numpy, pylab

# Calculate an equilibrium ionization balance for an element
kT = 0.5
Z = 8

# going to declare a datacache to speed up the calculations as we
# will be repeating things
datacache = {}

ionfrac_fast = pyatomdb.apec.return_ionbal(Z, kT, teunit='keV', datacache=datacache)

# The "fast" keyword makes the code open up a precalculated set of
# ionization balance data, a grid os 1251 values from 10^4K <= T <= 10^9K.
# By default it is true. Here, we turn it off for comparison
ionfrac_slow = pyatomdb.apec.return_ionbal(Z, kT, teunit='keV', datacache=datacache,
                                    fast=False)

fig = pylab.figure()
fig.show()
ax = fig.add_subplot(111)

ax.semilogy(range(0,Z+1), ionfrac_fast, 'o', label='fast')
ax.semilogy(range(0,Z+1), ionfrac_slow, '^', label='slow')
ax.set_ylim([1e-6,1])
ax.set_xlabel('Ion Charge')
ax.set_ylabel('Fraction')
ax.legend(loc=0)

pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('apec_examples_1_1.pdf')
fig.savefig('apec_examples_1_1.svg')


# now to do the same for several different ways of defining non-equilbrium:
tau = 1e11
# 1: from neutral
init_pop='ionizing' # this is the default
nei1 = pyatomdb.apec.return_ionbal(Z, kT, tau = tau, init_pop = init_pop,
                                   teunit='keV', datacache=datacache)
```

(continues on next page)

(continued from previous page)

```
# 2: from fully stripped
init_pop='recombining'
nei2 = pyatomdb.apec.return_ionbal(Z, kT, tau = tau, init_pop = init_pop,
                                   teunit='keV', datacache=datacache)


# 3: from a given Temperature
init_pop=0.1
nei3 = pyatomdb.apec.return_ionbal(Z, kT, tau = tau, init_pop = init_pop,
                                   teunit='keV', datacache=datacache)


# 4: from a given array
init_pop=numpy.array([0.1, #O0+
                      0.1, #O1+
                      0.1, #O2+
                      0.1, #etc
                      0.3,
                      0.1,
                      0.1,
                      0.1,
                      0.0])
nei4 = pyatomdb.apec.return_ionbal(Z, kT, tau = tau, init_pop = init_pop,
                                   teunit='keV', datacache=datacache)

# 5: from a dict of arrays
# (this is useful if you have lots of elements to do)
init_pop = {}
init_pop[8] = numpy.array([0.5, #O0+
                      0.0, #O1+
                      0.0, #O2+
                      0.0, #etc
                      0.0,
                      0.0,
                      0.0,
                      0.0,
                      0.5])
nei5 = pyatomdb.apec.return_ionbal(Z, kT, tau = tau, init_pop = init_pop,
                                   teunit='keV', datacache=datacache)

ax.cla()
ax.semilogy(range(Z+1), nei1, 'o', label='ionizing')
ax.semilogy(range(Z+1), nei2, '<', label='recombining')
ax.semilogy(range(Z+1), nei3, 's', label='0.1keV')
ax.semilogy(range(Z+1), nei4, '^', label='array')
ax.semilogy(range(Z+1), nei5, 'x', label='dict')

ax.set_ylim([1e-6,1])
ax.set_xlabel('Ion Charge')
ax.set_ylabel('Fraction')
ax.legend(loc=0)
```

(continues on next page)

(continued from previous page)

```
pylab.draw()
zzz=input("Press enter to continue")
# save image files
fig.savefig('apec_examples_1_2.pdf')
fig.savefig('apec_examples_1_2.svg')
```

Fig. 10: Ionization balance for oxygen

Fig. 11: Charge state distribution calculated many different ways for non equilibrium ionization.

1.2.5 Including APEC models in PyXSPEC

```
from apec_xspec import *

# declare a new model
# inital import creates pyapec, pyvapec, pyvvapec, analagous to apec, vapec, vvapec

m1 = xspec.Model('pyapec')

m1.show()

m1.pyapec.kT=5.0


# let's plot a spectrum
xspec.Plot.device='/xs'

xspec.Plot('model')

# you can fiddle with some aspects of the model directly
cie.set_eebrems(False) # turn off electron-electron bremsstrahlung

xspec.Plot('model')

# it is also possible to go in and change broadening, etc. The interface can be
# adjust quite easily by looking at apec_xspec.py to add extra parameters
```

There are other wrapper files which work in the same way for a model with a separate temperature set for line broadening, and others will be added in the `wrapper` directory.

1.2.6 Individual Use Cases

These are typically scripts created to answer user's questions which might be interesting. As a result, sometimes the exact response files etc may not be available to you. Please swap in what you need to.

1.2.6.1 Get PI Cross Sections

Extract the PI cross section data: `photoionization_data.py`

```
import pyatomdb, numpy,os, pylab
try:
  import astropy.io.fits as pyfits
except:
  import pyfits

# This is a sample routine that reads in the photoionization data
# It also demonstrates using get_data, which should download the data you
# need automagically from the AtomDB site.
#
# It also shows how to get the raw XSTAR PI cross sections.

# going to get PI cross section from iron 16+ to 17+ (Fe XVII-XVIII)
Z = 26
z1 = 17

# get the AtomDB level data
lvdata = pyatomdb.atomdb.get_data(Z, z1, 'LV')

# get the XSTAR PI data from AtomDB
pidata = pyatomdb.atomdb.get_data(Z, z1, 'PI')

# set up the figure
fig = pylab.figure()
fig.show()
ax = fig.add_subplot(111)


# to calculate the cross section (in cm^2) at a given single energy E (in keV)
# does not currently work with vector input, so have to call in a loop if you
# want multiple energies [I will fix this]

E = 10.

# get the ground level (the 0th entry in LV file) data
lvd = lvdata[1].data[0]

# This is the syntax for calculating the PI cross section of a given line
# This will work for non XSTAR data too.
sigma = pyatomdb.atomdb.sigma_photoion(E, Z, z1,lvd['phot_type'], lvd['phot_par'], \
               xstardata=pidata, xstarfinallev=1)


# To get the raw XSTAR cross sections (units: energy = keV, cross sections = Mb)
```

(continues on next page)

(continued from previous page)

```
# for level 1 -> 1 (ground to ground)
pixsec = pyatomdb.atomdb.__sort_pi_data(pidata, 1,1)
ax.loglog(pixsec['energy'], pixsec['pi_param']*1e-18, label='raw xstar data')

# label the plot
ax.set_title('Plotting raw XSTAR PI cross sections. Fe XVII gnd to Fe XVIII gnd')
ax.set_xlabel("Energy (keV)")
ax.set_ylabel("PI cross section (cm$^{2}$)")

pylab.draw()
zzz=input('press enter to continue')
```

1.2.6.2 Make Cooling Curve

Make a cooling curve, total emissivity in keV cm3 s-1, for each element in a specfied spectral range (e.g. 2 to 10 keV).

```
import pyatomdb, numpy, os, pylab

"""
This code is an example of generating a cooling curve: the total power
radiated in keV cm3 s-1 by each element at each temperature. It will
generate a text file with the emission per element at each temperature
from 1e4 to 1e9K.

This is similar to the atomdb.lorentz_power function, but with a few
normalizations removed to run a little quicker.

Note that the Anders and Grevesse (1989) abundances are built in to
this. These can be looked up using atomdb.get_abundance(abundset='AG89'),
or the 'angr' column of the table at
https://heasarc.nasa.gov/xanadu/xspec/xspec11/manual/node33.html
↪#SECTION00631000000000000000

Adjustable parameters (energy range, element choice) are in the block
marked #### ADJUST THINGS HERE

Note that any warnings of the nature:
"kT = 0.000862 is below minimum range of 0.000862. Returning lowest kT spectrum available
↪"
should be ignored. This is returning the temperature at our lowest tabulated value but is
a rounding error making it think it is outside our range.

Usage: python3 calc_power.py
"""

def calc_power(Zlist, cie, Tlist):

  """
  Zlist : [int]
    List of element nuclear charges
  cie : CIESession
```

(continues on next page)

(continued from previous page)

```
    The CIEsession object with all the relevant data predefined.
  Tlist : array(float)
    The temperatures at which to calculate the power (in K)
  """

  res = {}
  res['power'] = {}
  res['temperature'] = []
  cie.set_abund(numpy.arange(1,31), 0.0)
  kTlist = Tlist*pyatomdb.const.KBOLTZ
  en = (cie.ebins_out[1:]+cie.ebins_out[:-1])/2
  for i, kT in enumerate(kTlist):


    print("Doing temperature iteration %i of %i"%(i, len(kTlist)))
    T = Tlist[i]

    res['temperature'].append(T)
    res['power'][i] = {}

    for Z in Zlist:

      if Z==0:
        # This is the electron-electron bremstrahlung component alone

        #set all abundances to 1 (I need a full census of electrons in the plasma for e-
↪e brems)
        cie.set_abund(Zlist[1:], 1.0)
        # turn on e-e bremsstrahlung
        cie.set_eebrems(True)
        spec = cie.return_spectrum(kT, dolines=False, docont=False, dopseudo=False)
      else:
        # This is everything else, element by element.

        # turn off all the elements
        cie.set_abund(Zlist[1:], 0.0)
        # turn back on this element
        cie.set_abund(Z, 1.0)
        # turn off e-e bremsstrahlung (avoid double counting)
        cie.set_eebrems(False)

        spec = cie.return_spectrum(kT)
      # if Z = 1, do the eebrems (only want to calculate this once)
#      if Z == 1:
#        cie.set_eebrems(True)
#      else:
#        cie.set_eebrems(False)


      # get spectrum in ph cm3 s-1
      #spec = cie.return_spectrum(kT)
```

(continues on next page)

(continued from previous page)

```
      # convert to keV cm3 s-1, sum
      res['power'][i][Z] = sum(spec*en)

  return res

if __name__=='__main__':

  ############################
  #### ADJUST THINGS HERE ####
  ############################

  # Elements to include
  #Zlist = range(31) <- all the elements
  Zlist = [0,1,2,6,7,8,10,12,13,14,16,18,20,26,28] #<- just a few
  # Note that for this purpose, Z=0 is the electron-electron bremsstrahlung
  # continuum. This is not a general AtomDB convention, just what I've done here
  # to make this work.

  # specify photon energy range you want to integrate over (min = 0.001keV, max=100keV)
  Elo = 0.001 #keV
  Ehi = 100.0 #

  # temperatures at which to calculate curve (K)
  Tlist = numpy.logspace(4,9,51)

  # specify output file name (default output.txt)
  outfile = 'output.txt'

  ################################
  #### END ADJUST THINGS HERE ####
  ################################

  # set up the spectrum
  cie = pyatomdb.spectrum.CIESession()
  ebins = numpy.linspace(Elo, Ehi, 10001)
  cie.set_response(ebins, raw=True)
  cie.set_eebrems(True)
  # crunch the numbers
  k = calc_power(Zlist, cie, Tlist)

  # output generation
  o = open(outfile, 'w')

  # header row
  s = '# Temperature log10(K)'
  for i in range(len(Zlist)):
    s += ' %12i'%(Zlist[i])
  o.write(s+'\n')

  # for each temperature
  k['totpower'] = numpy.zeros(len(k['temperature']))
```

(continues on next page)

(continued from previous page)

```
  for i in range(len(k['temperature'])):
    s = '%22e'%(numpy.log10(k['temperature'][i]))
    for Z in Zlist:
      s+=' %12e'%(k['power'][i][Z])
      k['totpower'][i]+=k['power'][i][Z]
    o.write(s+'\n')

  # notes
  o.write("# Total Emissivity in keV cm^3 s^-1 for each element with AG89 abundances,␣
↪between %e and %e keV\n"%(Elo, Ehi))
  o.write("# To get cooling power, multiply by Ne NH")
  o.write("# Z=0 component is electron-electron bremsstrahlung component, only␣
↪significant at high T")
  o.close

  fig = pylab.figure()
  fig.show()
  ax = fig.add_subplot(111)

  ax.loglog(k['temperature'], k['totpower']*pyatomdb.const.ERG_KEV)

  ax.set_xlabel('Temperature (K)')
  ax.set_ylabel('Radiated Power (erg cm$^3$ s$^{-1}$)')

  ax.set_xlim(min(Tlist), max(Tlist))

  # draw graphs
  pylab.draw()

  zzz=input("Press enter to continue")

  # save image files
  fig.savefig('calc_power_examples_1_1.pdf')
  fig.savefig('calc_power_examples_1_1.svg')
```

Fig. 12: The total emissivity of the plasma between 0.001 and 100 keV.

1.3 Module Documentation

1.3.1 apec module

This modules contains the APEC code. It calls many different subroutines from throughout the PyAtomDB module.

It is expected (as of May 2020) that this code will go through a major rewrite to make it more pythonic and to make many helper routines private to declutter the module. Be wary of embedding too many links to this module in your code.

The apec module contains routines crucial for the APEC code. This also includes some interfaces to external C libraries (or will, eventually).

Version 0.1 - initial release Adam Foster September 16th 2015

pyatomdb.apec.**calc_brems_gaunt**(*E*, *T*, *z1*, *brems_type*, *datacache=False*, *settings=False*)

calculate the bremstrahulung free-free gaunt factor

Parameters

E
[float] Energy (in keV) to calculate gaunt factor

T
[float] Temperature (in K) of plasma

z1
[int] Ion charge +1 of ion (e.g. 6 for C VI)

brems_type
[int] Type of bremstrahlung requested: 1 = HUMMER = Non-relativistic: 1988ApJ…327..477H 2 = KELLOGG = Semi-Relativistic: 1975ApJ…199..299K 3 = RELATIVISTIC = Relativistic: 1998ApJ…507..530N 4 = BREMS_NONE = no bremstrahlung

settings
[dict] See description in atomdb.get_data

datacache
[dict] Used for caching the data. See description in atomdb.get_data

Returns

gaunt_ff
[float] The gaunt factor for the free-free process.

pyatomdb.apec.**calc_cascade_population**(*matrixA*, *matrixB*)

pyatomdb.apec.**calc_ee_brems**(*E*, *T*, *N*)

calculate the electron-electron bremsstrahlung.

Parameters

E
[array (float)] energy grid (keV)

T
[float] Electron temperature (keV)

N
[float] electron density (cm^-3)

Returns

array(float)
ee_brems in photons cm^s s^-1 keV-1 at each point E. This should be multiplied by the bin width to get flux per bin.

References

Need to check this!

pyatomdb.apec.**calc_full_ionbal**(*Te*, *tau=False*, *init_pop='ionizing'*, *Te_init=False*, *Zlist=False*, *teunit='K'*, *extrap=True*, *cie=True*, *settings=False*, *datacache=False*)

Calculate the ionization balance for all the elements in Zlist.

One of init_pop or Te_init should be set. If neither is set, assume all elements start from neutral.

Parameters

Te
[float] electron temperature in keV or K (default K)

tau
[float] N_e * t for the non-equilibrium ioniziation (default False, i.e. off)

init_pop
[dict of float arrays, indexed by Z] initial populations. E.g. init_pop[6]=[0.1,0.2,0.3,0.2,0.2,0.0,0.0]

Te_init
[float] initial ionization balance temperature, same units as Te

Zlist
[int array] array of nuclear charges to include in calculation (e.g. [8,26] for oxygen and iron)

teunit
[{'K' , 'keV'}] units of temperatures (default K)

extrap
[bool] Extrappolate rates to values outside their given range. (default False)

cie
[bool] If true, collisional ionization equilbrium calculation (tau, init_pop, Te_init all ignored)

Returns

final_pop
[dict of float arrays, indexed by Z] final populations. E.g. final_pop[6]=[0.1,0.2,0.3,0.2,0.2,0.0,0.0]

pyatomdb.apec.**calc_ioniz_popn**(*levpop*, *Z*, *z1*, *z1_drv*, *T*, *Ne*, *settings=False*, *datacache=False*, *do_xi=False*)

Calculate the level population due to ionization into the ion

Parameters

levpop: array(float)
The level population of the parent ion. Should already have abundance and ion fraction built in.

Z: int
z1: int
z1_drv: int
T: float
Ne: float
settings: dict
datacache: dict
do_xi: bool
Include collisional ionization

Returns

levpop_out: array(float)
The level populations of the Z,z1 ion

pyatomdb.apec.**calc_recomb_popn**(*levpop*, *Z*, *z1*, *z1_drv*, *T*, *dens*, *drlevrates*, *rrlevrates*, *settings=False*, *datacache=False*, *dronly=False*, *rronly=False*)

Calculate the level population of a recombined ion

Parameters

levpop: array(float)
Level populations, already taking into account elemental abundance and ion fraction of z1_drv

Z: int
z1: int
z1_drv: int
T: electron temperature (K)
dens: electron density (cm^-3)
drlevrates: array(float)
Rates into each level from DR calculations

rrlevrates: array(float)
Rates into each level from RR calculations

Returns

array(float)
Level population

pyatomdb.apec.**calc_satellite**(*Z*, *z1*, *T*, *datacache=False*, *settings=False*)

Calcaulate DR satellite lines

Parameters

Z: int
The nuclear charge of the element Z: int

z1
[int] Recombined Ion charge +1 of ion (e.g. 5 for C VI -> C V)

te: float
The electron temperature (K)

settings: dictionary
The settings read from the apec.par file by parse_par_file

Returns

array(linelist)
List of DR lines

array(levlistin)
Rates into each lower level, driven by DR

pyatomdb.apec.**calc_total_coco**(*cocodata*, *settings*)

Calculate the total emission in erg cm^3 s^-1

pyatomdb.apec.**compress_continuum**(*xin*, *yin*, *tolerance*, *minval=0.0*)

Compress the continuum into linear interpolatable grids

Parameters

xin
[array(float)] The bin edges (keV)

yin
[array(float)] The continuum in photons (or ergs) cm^3 s^-1 bin^-1. Should be 1 element shorter then xin

tolerance
[float] The tolerance of the final result (if 0.01, the result will always be within 1% of the original value)

Returns

xout
[array (float)] The energy points of the compressed energy grid (keV)

yout
[array (float)] The continuum, in photons(or ergs) cm^3 s^-1 keV^-1

pyatomdb.apec.**continuum_append**(*a*, *b*)

Join two continuum arrays together, expanding arrays as necessary

Parameters

a: numpy.array(dtype=continuum)
The first array

b: numpy.array(dtype=continuum)
The second array

Returns
c: numpy.array(dtype=continuum)
The two arrays combined, with continuum arrays resized as required.

pyatomdb.apec.**create_chdu_cie**(*cocodata*)

pyatomdb.apec.**create_cparamhdu_cie**(*cocodata*)

pyatomdb.apec.**create_lhdu_cie**(*linedata*)

pyatomdb.apec.**create_lhdu_nei**(*linedata*)

pyatomdb.apec.**create_lparamhdu_cie**(*linedata*)

pyatomdb.apec.**do_brems**(*Z*, *z1*, *T*, *abund*, *brems_type*, *eedges*)

Calculate the bremstrahlung emission in units of photon cm^3 s^-1 bin^-1

Parameters

Z
[int] nuclear charge for which result is required

z1
[int] ion charge +1

T
[float] temperture (Kelvin)

abund
[float] elemental abundance (should be between 1.0 and 0.0)

brems_type
[int] Type of bremstrahlung requested: 1 = HUMMER = Non-relativistic: 1988ApJ...327..477H 2 = KELLOGG = Semi-Relativistic: 1975ApJ...199..299K 3 = RELATIVISTIC = Relativistic: 1998ApJ...507..530N 4 = BREMS_NONE = no bremstrahlung

eedges
[array(float)] The energy bin edges for the spectrum (keV)

Returns

array(float)
bremstrahlung emission in units of photon cm^3 s^-1 bin^-1

pyatomdb.apec.**do_lines**(*Z*, *z1*, *lev_pop*, *N_e*, *datacache=False*, *settings=False*, *z1_drv_in=-1*)

Convert level populations into line lists

Parameters

Z: int
The nuclear charge of the element

z1
[int] Ion charge +1 of ion (e.g. 6 for C VI)

lev_pop
[array(float)] The level population for the ion. Should already have elemental abundance and ion fraction multiplied in.

N_e
[float] Electron Density (cm^-3)

datacache
[dict] Used for caching the data. See description in atomdb.get_data

settings
[dict] See description in atomdb.get_data

z1_drv_in
[int] the driving ion for this calculation, if not z1 (defaults to z1)

Returns

linelist: numpy.dtype(linetype)
The list of lines and their emissivities. see generate_datatypes

twophot: array(float)
The two-photon continuum on the grid specified by the settings If settings['TwoPhoton'] is False, then returns a grid of zeros.

pyatomdb.apec.**extract_gauntff**(*Z*, *gamma2*, *gaunt_U*, *gaunt_Z*, *gaunt_Ng*, *gaunt_g2*, *gaunt_gf*)

Extract the appropriate Gaunt free-free factor from the relativistic data tables of Nozawa, Itoh, & Kohyama, 1998 ApJ, 507,530

Parameters

Z
[int] Z for which result is required

gamma2
[array(float)] gamma^2 in units of Z^2 Rydbergs/kT

gaunt_U
[array(float)] u=E/kT

gaunt_Z
[array(int)] nuclear charge

gaunt_Ng
[array(int)] number of gamma^2 factors

gaunt_g2
[array(float)] gamma^2 factors

gaunt_gf
[array(float)] ff factors

Returns

array(float)
Gaunt factors.

References

Nozawa, Itoh, & Kohyama, 1998 ApJ, 507,530

pyatomdb.apec.**gather_rates**(*Z*, *z1*, *te*, *dens*, *datacache=False*, *settings=False*, *do_la=True*, *do_ai=True*, *do_ec=True*, *do_pc=True*, *do_ir=True*)

fetch the rates for all the levels of Z, z1

Parameters

Z: int
The nuclear charge of the element

z1
[int] ion charge +1

te
[float] temperture (Kelvin)

dens: float
electron density (cm^-3)

settings
[dict] See description in atomdb.get_data

datacache
[dict] Used for caching the data. See description in atomdb.get_data

Returns

up: numpy.array(float)
Initial level of each transition

lo: numpy.array(float)
Final level of each transition

rate: numpy.array(float)
Rate for each transition (in s-1)

pyatomdb.apec.**generate_apec_headerblurb**(*settings*, *linehdulist*, *cocohdulist*)

Generate all the headers for an apec run, and apply them to the HDUlist.

Parameters

settings: dict
The output of read_apec_parfile

hdulist
[list or array of fits HDUs] The hdus to have headings added.

Returns

None

pyatomdb.apec.**generate_cie_outputs**(*settings*, *Z*, *linelist*, *contlist*, *pseudolist*)

Convert a linelist and continuum values into an equilibrium AtomDB fits output

Parameters

settings: dictionary
The settings read from the apec.par file by parse_par_file

Z: int
The nuclear charge of the element

linelist: numpy.array(dtype=linelisttype)
The list of lines, separated by ion

contlist: dict
Dictionary with the different continuum contributions from each ion. Each is an array of ph cm^3 s^-1 bin^-1

pseudolist: dict
Dictionary with the different pseudocontinuum contributions from each ion. Each is an array of ph cm^3 s^-1 bin^-1

Returns

None

pyatomdb.apec.**generate_datatypes**(*dtype*, *npseudo=0*, *ncontinuum=0*)

returns the various data types needed by apec

Parameters

dtype
[string] One of "linetype", "cielinetype", "continuum"

npseudo
[int (default=0)] Number of pseudocontinuum points for "continuum" type

ncontinuum
[int (default=0)] Number of continuum points for "continuum" type

Returns

numpy.dtype
The data dtype in question

pyatomdb.apec.**generate_nei_outputs**(*settings*, *Z*, *linelist*, *contlist*, *pseudolist*, *ionfrac_nei*)

Convert a linelist and continuum values into a non-equilibrium AtomDB fits output

Parameters

settings: dictionary
The settings read from the apec.par file by parse_par_file

Z: int
The nuclear charge of the element

linelist: numpy.array(dtype=linelisttype)
The list of lines, separated by ion

contlist: dict
Dictionary with the different continuum contributions from each ion. Each is an array of ph cm^3 s^-1 bin^-1

pseudolist: dict
Dictionary with the different pseudocontinuum contributions from each ion. Each is an array of ph cm^3 s^-1 bin^-1

Returns

None

pyatomdb.apec.**kurucz**(*uin*, *gam*)

Correction factors to Kellogg bremstrahlung calculation by Bob Kurucz

Parameters

uin
[array(float)] energy grid, units of E/kT (both in keV)

gam
[array(float)] Z**2/T, in units of Rydbergs

Returns

array(float)
gaunt factors at high gam (> 0.1)

pyatomdb.apec.**make_vector**(*linear*, *minval*, *step*, *nstep*)

Create a vector from the given inputs

Parameters

linear: boolean
Whether the array should be linear or log spaced

minval: float
initial value of the array. In dex if linear==False

step: float
step between points on the array. In dex if linear==False

nstep: int
number of steps

Returns

array(float)
array of values spaced out use the above parameters

pyatomdb.apec.**make_vector_nbins**(*linear*, *minval*, *maxval*, *nstep*)

Create a vector from the given inputs

Parameters

linear: boolean
Whether the array should be linear or log spaced

minval: float
initial value of the array. In dex if linear==False

maxval: float
maximum value of the array. In dex if linear==False

nstep: int
number of steps

Returns

array(float)
array of values spaced out use the above parameters

pyatomdb.apec.**parse_par_file**(*fname*)

Parse the apec.par input file for controlling APEC

Parameters

fname
[string] file name

Returns

dict
The settings in "key:value" pairs.

pyatomdb.apec.**return_ionbal**(*Z*, *Te*, *init_pop=False*, *tau=False*, *teunit='K'*, *filename=False*, *datacache=False*, *fast=True*, *settings=False*, *debug=False*, *extrap=True*)

Solve the ionization balance for a element Z.

Parameters

Z
[int] atomic number of element

Te
[float or array] electron temperature(s), default in K

init_pop
[float array] initial population of ions for non-equlibrium calculations. Will be renormalised to 1.

tau
[float or array] N_e * t for the non-equilibrium ioniziation, in cm^3 s^-1.

Te_init
[float] initial ionization balance temperature, same units as Te

teunit
[{'K' , 'keV'}] units of temperatures (default K)

filename
[string] Can optionally point directly to the file in question, i.e. to look at older data look at $HEADAS/../spectral/modelData/eigenELSYMB_v3.0.fits. If not set, download from AtomDB FTP site.

datacache
[dict] Used for caching the data. See description in atomdb.get_data

fast
[bool] If true, use precalculated eigenvector files to obtain CIE and NEI results

Returns

final_pop
[float array] final populations.

pyatomdb.apec.**run_apec**(*fname*)

Run the entire APEC code using the data in the parameter file fname

Parameters

fname
[string] file name

Returns

None

pyatomdb.apec.**run_apec_element**(*settings*, *te*, *dens*, *Z*)

Run the APEC code using the settings provided for one element

Parameters

settings: dictionary
The settings read from the apec.par file by parse_par_file

te: float
The electron temperature (K)

dens: float
The electron density (cm^-3)

Z: int
The nuclear charge of the element

Returns

None

pyatomdb.apec.**run_apec_ion**(*settings*, *te*, *dens*, *Z*, *z1*, *ionfrac*, *abund*)

Run the APEC code using the settings provided for an individual ion.

Parameters

settings: dictionary
The settings read from the apec.par file by parse_par_file

te: float
The electron temperature (K)

dens: float
The electron density (cm^-3)

Z: int
The nuclear charge of the element

z1: int
The ion charge +1 of the ion

ionfrac: float
The fractional abundance of this ion (between 0 and 1)

abund: float
The elemental abundance of the element (normalized to H)

Returns

linelist
[numpy array] List of line details and emissivities

continuum
[array] Continuum emission in photons bin-1 s-1. This is a 3-item dict, with "rrc", "twophot", "brems" entries for each continuum source

pseudocont
[array] Pseudo Continuum emission in photons bin-1 s-1

pyatomdb.apec.**run_wrap_run_apec**(*fname*, *Z*, *iTe*, *iDens*)

After running the APEC code ion by ion, use this to combine into FITS files.

Parameters

fname
[string] file name of par file

Z: int
The atomic numbers

iTe: int
The temperature index

iDens: int
The density index

Returns

None

pyatomdb.apec.**solve_ionbal**(*ionrate*, *recrate*, *init_pop=False*, *tau=False*)

solve_ionbal: given a set of ionization and recombination rates, find the equilibrium ionization balance. If init_pop and tau are set, do an non-equilibrium calculation starting from init_pop and evolving for n_e * t = tau (cm^-3 s)

Parameters

ionrate
[float array] the ionization rates, starting with neutral ionizing to +1

recrate
[float array] the recombination rates, starting with singly ionized recombining to neutral

init_pop
[float array] initial population of ions for non-equlibrium calculations. Will be renormalised to 1.

tau
[float] N_e * t for the non-equilibrium ioniziation

Returns

final_pop
[float array] final populations.

Notes

Note that init_pop & final_pop will have 1 more element than ionrate and recrate.

pyatomdb.apec.**solve_level_pop**(*init*, *final*, *rates*, *settings*)

Solve the level population

Parameters

init
[array(int)] The initial level for each transition

final
[array(int)] The initial level for each transition

rates
[array(float)] The rate for each transition

settings: dictionary
The settings read from the apec.par file by parse_par_file

Returns

array(float)
The level population

pyatomdb.apec.**wrap_ion_directly**(*fname*, *ind*, *Z*, *z1*)

pyatomdb.apec.**wrap_run_apec**(*fname*, *readpickle=False*, *writepickle=False*)

After running the APEC code ion by ion, use this to combine into FITS files.

Parameters

fname
[string] file name

readpickle
[bool] Load apec results by element from pickle files, instead of regenerating

Returns

None

pyatomdb.apec.**wrap_run_apec_element**(*settings*, *te*, *dens*, *Z*, *ite*, *idens*, *writepickle=False*, *readpickle=False*)

Combine wrap_run_apec_ion results for an element

Parameters

settings: dictionary
The settings read from the apec.par file by parse_par_file

te: float
The electron temperature (K)

dens: float
The electron density (cm^-3)

Z: int
The nuclear charge of the element

ite: int
The temperature index

idens: int
The density index

writepickle: bool
Dump data into a pickle file. Useful for rapidly combining data after runs.

readpickle: bool
Read data from a pickle file. Useful for rapidly combining data after runs. Usually the result of a previous call using writepickle=True

Returns

None

1.3.2 atomic module

This modules contains basic atomic parameters (i.e. atomic numbers, element symbols)

atomic.py contains routines related to basic atomic data, e.g. converting integer nuclear charge to element symbols, etc.

Version -.1 - initial release Adam Foster July 17th 2015

`pyatomdb.atomic.Z_to_mass(Z, raw=False)`

Converts element symbol to atomic mass, e.g. "C" -> 12.0107

Isotope fractions based on those found in earth's crust samples, your astrophysical object may vary.

Parameters

Z
[int] nuclear charge, e.g 6 for C

raw
[bool] if true, ignore Z, and return the entire mass list as an array with a 0 at the beginning so ret[12] = mass of carbon.

Returns
——-
float
mass in a.m.u. for the element. (e.g. 12.0107 for C)

References

Atomic masses are taken from: Pure Appl. Chem. 81 NO 11, 2131-2156 (2009) Masses for Technetium, Promethium, Polonium, Astatine, Radon, Francium, Radium & Actinum are estimates. If you need these you probably aren't doing astronomy...

`pyatomdb.atomic.Ztoelname(Z)`

Returns element name of element with nuclear charge Z.

Parameters

Z
[int] nuclear charge of element (e.g. 6 for carbon)

Returns

str
element name (e.g. "Carbon" for carbon)

pyatomdb.atomic.**Ztoelsymb**(*Z*)

Returns element symbol of element with nuclear charge Z.

Parameters

Z - nuclear charge of element (e.g. 6 for carbon)

Returns

element symbol (e.g. "C" for carbon)
Version 0.1 28 July 2009
Adam Foster

pyatomdb.atomic.**config_to_occup**(*cfgstr*, *nel=-1*, *shlmax=-1*, *noccup=[-1]*)

pyatomdb.atomic.**elsymb_to_Z**(*elsymb*)

Converts element symbol to nuclear charge, e.g. "C" -> 6

Parameters

elsymb
[str] Element symbol, e.g. "C". Case insensitive.

Returns

int
Z for the ion. (e.g. 6 for C)

pyatomdb.atomic.**elsymb_to_z0**(*elsymb*)

Converts element symbol to nuclear charge, e.g. "C" -> 6 (wrapper to elsymb_to_Z, retained for consistency)

Parameters

elsymb
[str] Element symbol, e.g. "C". Case insensitive.

Returns

int
Z for the ion. (e.g. 6 for C)

pyatomdb.atomic.**get_maxn**(*cfgstr*)

pyatomdb.atomic.**get_parity**(*cfgstr*)

pyatomdb.atomic.**int2roman**(*number*)

pyatomdb.atomic.**int_to_roman**(*input*)

Convert an integer to Roman numerals.

pyatomdb.atomic.**occup_to_cfg**(*occlist*)

pyatomdb.atomic.**occup_to_config**(*occup*)

pyatomdb.atomic.**parse_config**(*cfgstr*)

pyatomdb.atomic.**parse_eissner**(*cfgstr*, *nel=0*, *levelmap=None*)

pyatomdb.atomic.**roman_to_int**(*input*)

Convert a roman numeral to an integer.

pyatomdb.atomic.**shorten_config**(*cfgstr*, *nel=0*)

Shorten the configuration as required

Parameters

cfgstr
[string] configuration string. Should be simplified already e.g. '1s2 2s2 3p1'

Returns

cfgshrt
[string] shortened configuration, e.g. '3p1'

pyatomdb.atomic.**spectroscopic_name**(*Z*, *z1*)

Converts Z,z1 to spectroscopic name, e.g. 6,5 to "C V"

Parameters

Z
[int] nuclear charge (e.g. 6 for C)

z1
[int] ion charge +1 (e.g. 5 for C4+)

Returns

str
spectroscopic symbol for ion (e.g. "C V" for C+4)

pyatomdb.atomic.**spectroscopictoz0**(*name*)

Converts spectroscopic name to Z, z1, e.g. "C V" to 6,5

Parameters

name
[str] Ion name, e.g. "C V"

Returns

int, int
Z, z1 for the ion. (e.g. 6,5 for C V)

pyatomdb.atomic.**z0_to_mass**(*z0*)

Converts element symbol to atomic mass, e.g. "C" -> 12.0107

(wrapper to Z_to_mass, retained for consistency)

Isotope fractions based on those found in earth's crust samples, your astrophysical object may vary.

Parameters

z0
[int] nuclear charge, e.g 6 for C

Returns

float
mass in a.m.u. for the element. (e.g. 12.0107 for C)

References

Atomic masses are taken from: Pure Appl. Chem. 81 NO 11, 2131-2156 (2009) Masses for Technetium, Promethium, Polonium, Astatine, Radon, Francium, Radium & Actinum are estimates. If you need these you probably aren't doing astronomy...

pyatomdb.atomic.**z0toelname**(*z0*)

Returns element name of element with nuclear charge z0. (wrapper to Ztoelname for compatibility purposes)

Parameters

z0
[int] nuclear charge of element (e.g. 6 for carbon)

Returns

str
element name (e.g. "Carbon" for carbon)

pyatomdb.atomic.**z0toelsymb**(*z0*)

Returns element symbol of element with nuclear charge z0. (wrapper to Ztoelsymb for compatibility purposes)

Parameters

z0
[int] nuclear charge of element (e.g. 6 for carbon)

Returns

str
element symbol (e.g. "C" for carbon)

1.3.3 atomdb module

This modules is designed to interact with the main atomic database, extracting real values of coefficients and so on.

1.3.4 const module

A series of physical constants and constants relevant to running the APEC code.

This contains a list of constants, both physical and apec code related.

Version 0.1 - initial release Adam Foster July 17th 2015

1.3.5 spectrum module

This module contains codes for creating spectra from the AtomDB emissivity files

1.3.6 util module

This modules contains simple utility codes (sorting, file handling etc) that pyatomdb relies on.

util.py contains a range of miscellaneous helper codes that assist in running other AtomDB codes but are not in any way part of a physical calculation.

Version -.1 - initial release Adam Foster July 17th 2015

exception pyatomdb.util.**NotImplementedError**

Bases: ValueError

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyatomdb.util.**OptionError**

Bases: ValueError

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyatomdb.util.**ReadyError**

Bases: ValueError

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception pyatomdb.util.**UnitsError**

Bases: ValueError

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

pyatomdb.util.**check_version**()

Checks if there is a more recent version of the database to install.

Parameters

None.

Returns

None

pyatomdb.util.**convert_spec**(*spec*, *specunit*, *specunitout*)

Convert spectral ranges from specunit to specunitout

Parameters

spec
[array] The units to return

specunit
[string] The input spectral unit ('keV', 'A')

specunitout
[string] The output spectral unit ('keV', 'A')

Returns

specout
[array] spec, converted to specunitout

pyatomdb.util.**convert_temp**(*Te*, *teunit*, *teunitout*)

Convert temperature (Te) from units teunit to teunitout

Parameters

Te
[float] The temperature

teunit
[string] units of Te

teunitout
[string] output temperature units

pyatomdb.util.**download_atomdb_emissivity_files**(*adbroot*, *userid*, *version*)

Download the AtomDB equilibrium emissivity files for AtomDB"

This code will go to the AtomDB FTP site and download the necessary files. It will then unpack them into a directory adbroot. It will not overwrite existing files with the same md5sum (to avoid pointless updates) but it will not know this until it has downloaded and unzipped the main file.

Parameters

adbroot
[string] The location to install the data. Typically should match $ATOMDB

userid
[string] An 8 digit ID number. Usually passed as a string, but integer is also fine (provided it is all numbers)

version
[string] The version string for the release, e.g. "3.0.2"

Returns

None

pyatomdb.util.**download_atomdb_nei_emissivity_files**(*adbroot*, *userid*, *version*)

Download the AtomDB non-equilibrium emissivity files for AtomDB"

This code will go to the AtomDB FTP site and download the necessary files. It will then unpack them into a directory adbroot. It will not overwrite existing files with the same md5sum (to avoid pointless updates) but it will not know this until it has downloaded and unzipped the main file.

Parameters

adbroot
[string] The location to install the data. Typically should match $ATOMDB

userid
[string] An 8 digit ID number. Usually passed as a string, but integer is also fine (provided it is all numbers)

version
[string] The version string for the release, e.g. "3.0.2"

Returns

None

pyatomdb.util.**figcoords**(*lowxpix*, *lowypix*, *highxpix*, *highypix*, *lowxval*, *lowyval*, *highxval*, *highyval*, *xpix*, *ypix*, *logx=False*, *logy=False*)

pyatomdb.util.**generate_equilibrium_ionbal_files**(*filename*, *settings=False*)

Generate the eigen files that XSPEC uses to calculate the ionizatoin balances

Parameters

filename
[string] file to write

settings
[dict] This will let you override some standard inputs for get_data:

- settings['filemap']: the filemap to use if you do not want to use the default $ATOMDB/filemap
- settings['atomdbroot']: If you have files in non-standard locations you can replace $ATOMDB with this value

Returns

none

pyatomdb.util.**generate_isis_files**(*version=''*, *outfile='atomdb_VERSION_lineid.tar.bz2'*)

Generate the atomic data necessary solely for identifying lines in AtomDB. Useful in ISIS, for example.

Parameters

version
[string] version number to generate line ID tarball for. Defaults to version in $ATOMDB/VERSION

outfile
[string] the file to be generated. Defaults to atomdb_VERSION_lineid.tar.bz2

Returns
——-
none

pyatomdb.util.**generate_web_fitsfiles**(*version=''*, *outdir=''*)

Split the linelist files into many small files and make an index for them

Parameters

version
[string] version number to generate this for. Defaults to version in $ATOMDB/VERSION

outdir
[string] Output files will be placed in this directory. Defaults to 'webonly"

Returns
——-
none

pyatomdb.util.**generate_xspec_ionbal_files**(*Z*, *filesuffix*, *settings=False*)

Generate the eigen files that XSPEC uses to calculate the ionizatoin balances

Parameters

Z
[int] atomic number of element

filesuffix
[string] the filename will be eigenELSYMB_filesuffix.fits

settings
[dict] This will let you override some standard inputs for get_data:

- settings['filemap']: the filemap to use if you do not want to use the default $ATOMDB/filemap
- settings['atomdbroot']: If you have files in non-standard locations you can replace $ATOMDB with this value

Returns

none

pyatomdb.util.**initialize**()

Initialize your AtomDB Setup

This code will let you select where to install AtomDB, get the latest version of the filemap, and download the emissivity files needed for various functions to work.

Parameters

None.

Returns

None

pyatomdb.util.**keyword_check**(*keyword*)

Returns False is the keyword is in fact false, otherwise returns True

Parameters

keyword: any
The keyword value

Returns

bool
True if the keyword is set to not False, otherwise False

pyatomdb.util.**load_user_prefs**(*adbroot='$ATOMDB'*)

Loads user preference data from $ATOMDB/userdata

Parameters

adbroot
[string] The AtomDB root directory. Defaults to environment variable $ATOMDB.

Returns

dictionary
keyword/setting pairs e.g. settings['USERID'] = "12345678"

pyatomdb.util.**make_linelist**(*linefile*, *outfile*)

Create atomdb linelist file from line.fits file

Parameters

linefile
[string] The filename of the line file

outfile
[string] The output filename of the string

Returns

none

pyatomdb.util.**make_release_filetree**(*filemapfile_in*, *filemapfile_out*, *replace_source*, *destination*, *versionname*)

Take an existing filemap, copy the files to the atomdbftp folder as required.

Parameters

filemapfile_in
[string] The existing filemap file for the new release

filemapfile_out
[string] The filename for the produced filemap

replace_source
[string] All new files are in this directory.

destination
[string] The folder to store the files in

versionname
[string] The version string for the new files (e.g. 3_0_4)

Returns

None

Notes

This code searches for any files which don't have $ATOMDB in the filename and assumes they are new.

It updates the file name to be $ATOMDB/elname/elname_ion/elname_ion_FTYPE_versionname.fits

Versionname will have its last number stsripped and replaced with "a". So 3_0_4_2 becomes 3_0_4_a. This reflects that 4-number versions are for revisions of a file under development, while 3 number + letter are for released data.

And then copies it to the destination folder, compressing it with gzip.

pyatomdb.util.**make_release_tarballs**(*ciefileroot*, *neifileroot*, *filemap*, *versionname*, *releasenotes*, *parfile*, *neiparfile*, *makelinelist=False*)

Create tarball for exmissivity files for a new release.

Parameters

ciefileroot
[string] The path to the CIE line and coco files, with the _line.fits and _coco.fits ommitted.

neifileroot
[string] The path to the NEI line and coco files, with the _line.fits and _comp.fits ommitted.

filemap
[string] The filemap file

versionname
[string] The version string for the new files (e.g. 3.0.4).

releasenotes
[string] The file name for the release notes.

parfile
[string] The parameter file used to create the data

neiparfile
[string] The parameter file used to create the NEI data

makelinelist
[bool] Remake the line list from the line file. If not specified, assumes linelist file already exists.

Returns

None

pyatomdb.util.**make_vec**(*d*)

Create vector version of d, return True or false depending on whether input was vector or not

Parameters

d: any scalar or vector
The input

Returns

vecd
[array of floats] d as a vector (same as input if already an iterable type)

isvec
[bool] True if d was a vector, otherwise False.

pyatomdb.util.**md5Checksum**(*filePath*)

Calculate the md5 checksum of a file

Parameters

filepath
[str] the file to calculate the md5sum of

Returns

string
the hexadecimal string md5 hash of the file

References

Taken from http://joelverhagen.com/blog/2011/02/md5-hash-of-file-in-python/

pyatomdb.util.**mkdir_p**(*path*)

Create a directory. If it already exists, do nothing.

Parameters

path
[string] The directory to make

Returns

none

pyatomdb.util.**question**(*question*, *default*, *multichoice=[]*)

Ask question with default answer provided. Return answer

Parameters

question
[str] Question to ask

default
[str] Default answer to question

multichoice
[str] if set, answer must be one of these choices

Returns

str
The answer.

pyatomdb.util.**record_upload**(*fname*)

Transmits record of a file transfer to AtomDB

This simply transmits the USERID, filename, and time to AtomDB. If USERID=0, then the user has chosen not to share this information and this is skipped

Parameters

fname
[string] The file name being downloaded.

Returns

None

pyatomdb.util.**switch_version**(*version*, *force=False*)

Changes the AtomDB version. Note this will overwrite several links on your hard disk, and will *NOT* be repaired upon quitting python.

The files affect are the VERSION file and the soft links $ATOMDB/apec_line.fits, $ATOMDB/apec_coco.fits, $ATOMDB/filemap and $ATOMDB/apec_linelist.fits

Parameters

version: string
The version of AtomDB to switch to. Should be of the form "2.0.2"

force
[bool] If True, force a re-download of all the relevant files regardless of whether they already exist or not.

Returns

None

pyatomdb.util.**unique**(*s*)

Return a list of the elements in s, but without duplicates.

For example, unique([1,2,3,1,2,3]) is some permutation of [1,2,3], unique("abcabc") some permutation of ["a", "b", "c"], and unique(([1, 2], [2, 3], [1, 2])) some permutation of [[2, 3], [1, 2]].

For best speed, all sequence elements should be hashable. Then unique() will usually work in linear time.

If not possible, the sequence elements should enjoy a total ordering, and if list(s).sort() doesn't raise TypeError it's assumed that they do enjoy a total ordering. Then unique() will usually work in O(N*log2(N)) time.

If that's not possible either, the sequence elements must support equality-testing. Then unique() will usually work in quadratic time.

Parameters

s
[list type object] List to remove the duplicates from

Returns

list type object
... with all the duplicates removed

References

Taken from Python Cookbook, written by Tim Peters. http://code.activestate.com/recipes/52560/

pyatomdb.util.**write_ai_file**(*fname*, *dat*, *clobber=False*)

Write the data in list dat to fname

Parameters

fname
[string] The file to write

dat
[list] The data to write. Should be a list with the following keywords:

- Z : int: nuclear charge
- z1 : int: ion charge + 1
- comments : iterable of strings: comments to append to the file
- data : numpy.array : stores all the individual level data, with the following types:
  - ion_init : int : Inital ion state of transition
  - ion_final : int : Final ion state of transition
  - level_init : int : Initial level of transition
  - level_final : int : Final level of transition
  - auto_rate : float : Autoionization rate (s-1)
  - auto_err : float : Error in autoionization rate (s-1)
  - auto_ref : string(20) : Autoionization rate reference (bibcode)

clobber
[bool] Overwrite existing file if it exists.

Returns

none

pyatomdb.util.**write_develop_data**(*data*, *filemapfile*, *Z*, *z1*, *ftype*, *folder*, *froot*)

pyatomdb.util.**write_dr_file**(*fname*, *dat*, *lvdat=None*, *clobber=False*)

Write the data in list dat to fname

Parameters

fname
[string][The file to write]

dat
[list][The data to write. Should be a list with the following keywords:]

- Z : int : nuclear charge
- z1 : int: ion charge + 1
- comments : iterable of strings: comments to append to the file
- data : numpy.array: stores all the individual level data, with the following types
  - upper_lev : int : upper level of transition
  - lower_lev : int : lower level of transition
  - wavelen : float : Wavelength of transtion (A)
  - wave_obs : float : Observed wavelength of transition (A)
  - wave_err : float Error in wavelength (A)
  - dr_type : int : DR data type. 1=Jaconelli, 2 = Safranova
  - e_excite : float : transition excitation energy (keV)
  - eexc_err : float : error in transition excitation energy (keV)
  - satelint : float : intensity factor (s-1)
  - satinterr : float : error in intensity factor (s-1)
  - params : float(10) : parameters
  - drrate_ref : string(20) : DR rate reference (usually bibcode)
  - wave_ref : string(20) : wavelength reference (bibcode)
  - wv_obs_ref : string(20) : observed wavelength reference (bibcode)

clobber
[bool] Overwrite existing file if it exists.

Returns

none

pyatomdb.util.**write_ec_file**(*fname*, *dat*, *clobber=False*)

Write the data in list dat to fname

Parameters

fname
[string] The file to write

dat
[list] The data to write. Should be a list with the following keywords:

- Z : int : nuclear charge
- z1 : int : ion charge + 1
- comments : iterable of strings: comments to append to the file
- data : numpy.array : stores all the individual level data, with the following types:
  - lower_lev : int : Lower level of transition
  - upper_lev : int : Upper level of transition
  - coeff_type : int : Coefficient type

- min_temp : float : Minimum temperature in range (K)
- max_temp : float : Maximum temperature in range (K)
- temperature : float(20) : List of temperatures (K)
- effcollstrpar : float(20) : Effective collision strength parameters
- inf_limit : float (OPTIONAL - if type 1.2.0) : High temperature limit point, if provided.
- reference : string(20) : Collisional excitation reference (bibcode)

clobber
[bool] Overwrite existing file if it exists.

Returns

none

pyatomdb.util.**write_ionbal_file**(*Te*, *dens*, *ionpop*, *filename*, *Te_linear=False*, *dens_linear=False*)

Create ionization balance file

Parameters

Te
[array(float)] temperatures (in K)

dens
[array(float)] electron densities (in cm^-3)

ionpop
[dict of arrays] one entry for each element: ionpop[2] = numpy.array(nion,nte, ndens)

filename
[str] filename to write to

Te_linear
[bool] if true, temperature grid is linear

dens_linear
[bool] if true, density grid is linear

pyatomdb.util.**write_ir_file**(*fname*, *dat*, *clobber=False*)

Write the data in list dat to fname

Parameters

fname
[string] The file to write

dat
[list] The data to write. Should be a list with the following keywords:

- Z : int : nuclear charge
- z1 : int : ion charge + 1
- comments : iterable of strings : comments to append to the file
- ionpot : float : ionization potential (eV)
- ip_dere : float : ionization potential (eV) (from dere, optional)
- data : numpy.array : stores all the individual level data, with the following types:
  - element : int : Nuclear Charge

- ion_init : int : Initial ion stage
- ion_final : int : Final ion stage
- level_init : int : Initial level
- level_final : int : Final level
- tr_type : string(2) : Transition type:

```
CI = collisional excitaion
EA = excitation autoionization
RR = radiative recombination
DR = dieclectronic recombination
XI = ionization, excluded from total rate calculation
XR = recombination, excluded from total rate calculation
(XR and XI are used to populate level directly)
```

- tr_index : int : index within the file
- par_type : int : parameter type, i.e. how the data is stored
- min_temp : float : Minimum temperature in range (K)
- max_temp : float : Maximum temperature in range (K)
- temperature : float(20) : List of temperatures (K)
- ionrec_par : float(20) : Ionization and recombination rate parameters
- wavelen : float : Wavelength of emitted lines (A) [not used]
- wave_obs : float : Observed wavelength of emitted lines (A) [not used]
- wave_err : float : Error in these wavelengths (A) [not used]
- br_ratio : float : Branching ratio of this line [not used]
- br_rat_err : float : Error in branching ratio [not used]
- label : string(20) : Label for the transition
- rate_ref : string(20) : Rate reference (bibcode)
- wave_ref : string(20) : Wavelength reference (bibcode)
- wv_obs_ref : string(20) : Observed wavelength reference (bibcode)
- br_rat_ref : string(20) : Branching ratio reference (bibcode)

clobber
[bool] Overwrite existing file if it exists.

Returns

none

pyatomdb.util.**write_la_file**(*fname*, *dat*, *clobber=False*)

Write the data in list dat to fname

Parameters

fname
[string] The file to write

dat
[list] The data to write. Should be a list with the following keywords:

- Z : int : nuclear charge
- z1 : int : ion charge + 1
- comments : iterable of strings : comments to append to the file
- data : numpy.array: stores all the individual level data, with the following types:
  - upper_lev : int : Upper level of transition
  - lower_lev : int : Lower level of transition
  - wavelen : float : Wavelength of transition (A)
  - wave_err : float : Error in wavelength (A)
  - einstein_a : float : Einstein A coefficient (s-1)
  - ein_a_err : float : Error in A coefficient (s-1)
  - wave_ref : string(20) : wavelength reference (bibcode)
  - ein_a_ref : string(20) : A-value reference (bibcode)

clobber
[bool] Overwrite existing file if it exists.

Returns

none

pyatomdb.util.**write_lv_file**(*fname*, *dat*, *clobber=False*)

Write the data in list dat to fname

Parameters

fname
[string][The file to write]

dat
[list][The data to write. Should be a list with the following keywords:]

- Z : int : nuclear charge
- z1 : int: ion charge + 1
- comments : iterable of strings: comments to append to the file
- data : numpy.array: stores all the individual level data, with the following types
  - elec_config : string (40 char max) : Electron configuration strings
  - energy : float: Level energy (eV)
  - e_error : float : Energy level error (eV)
  - n_quan : int : N quantum number
  - l_quan : int : L quantum number
  - s_quan : float : S quantum number
  - lev_deg : int : level degeneracy
  - phot_type : int : photoionization data type:

```
-1. none

 0. hydrogenic

 1. Clark

 2. Verner

 3. XSTAR data
```

- phot_par : float(20) : photoionization paramters (see specific PI type for definition)
- Aaut_tot : float (optional) : the total autoionization rate out of the level (s^-1)
- Arad_tot : float (optional) : the total radiative rate out of the level (s^-1)
- energy_ref : string(20) : energy reference (usually bibcode)
- phot_ref : string(20) : photoionization reference (bibcode)
- Aaut_ref : string(20) : total autoionization rate reference (bibcode)
- Arad_ref : string(20) : total radiative decay rate reference (bibcode)

clobber
[bool] Overwrite existing file if it exists.

Returns

none

pyatomdb.util.**write_user_prefs**(*prefs*, *adbroot='$ATOMDB'*)

Write user preference data to $ATOMDB/userdata. This will overwrite the entire file.

Therefore you should use "load_user_prefs", then add in additional keywords, the call write_user_prefs.

Parameters

prefs: dictionary
keyword/setting pairs e.g. settings['USERID'] = "12345678"

adbroot
[string] The AtomDB root directory. Defaults to environment variable $ATOMDB.

Returns

None

1.4 License

that use of the software would not infringe privately owned rights; (4) do not warrant that the software is error-free or will be maintained, supported, updated or enhanced; (5) will not be liable for any indirect, incidental, consequential special or punitive damages of any kind or nature, including but not limited to lost profits or loss of data, on any basis arising from contract, tort or otherwise, even if any of the parties has been warned of the possibility of such loss or damage.

1.5 Contact

This module is still in very active development. If you have feature requests, bug reports or any other questions, please contact us through the project's GitHub page.

CHAPTER

TWO

OUTLINE

PyAtomDB is a selection of utilities designed to interact with the AtomDB database . These utilities are under constant development. Please get in touch with any issues that arise.

There are several different modules currently. These are:

atomdb
: A series of codes for interacting with the AtomDB atomic database

atomic
: Basic atomic data routines - e.g. converting element symbols to atomic number, etc.

const
: Physical and code related constants

spectrum
: Routines for generating spectra from the published AtomDB line and continuum emissivity files

util
: Utility codes (sorting etc) that pyatomdb relies on.

apec
: The full APEC code

To report bugs or make feature requests, email the code authors or raise an issue at the github page

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX