EIS: The EUV Imaging Spectrometer on Hinode

EIS DATA ANALYSIS IN PYTHON

Version 0.01 | August 29, 2019

Contents

Introduction

The EUV Imaging Spectrometer — EIS[1] — was designed to study the solar atmosphere and answer fundamental questions on the heating of the solar corona, the origin of the solar wind, and the release of energy in solar flares. EIS observes two wavelength ranges in the extreme ultraviolet, 171—212 Å and 245—291 Å with a spectral resolution of about 22 mÅ and a plate scale of 1″ per pixel. Solar images can be made by stepping the slit over a region of the Sun and taking an exposure at each position. A detailed description of EIS is given in the instrument paper[2].

This document describes the basic elements of EIS data analysis using new HDF5 level-1 files and new routines written in the Python programming language. At the beginning of the Hinode mission the strategy was to release unprocessed level-0 FITS files and software routines written in IDL for processing these files into a format that could be used for data analysis. Additionally, all of routines for computing ancillary information, such as the offsets of the detectors or the magnitude of the instrumental broadening, were all written in IDL. Unfortunately, IDL is an expensive, proprietary language, little used outside of solar physics. Python, in contrast, is a free, open source language that has grown dramatically in popularity since the launch of Hinode, making it an obvious choice for future software development.

To accelerate the transition to Python we have created a new level-1 product that contains both the processed level-1 data and the ancillary information needed for data analysis. The alternative approach, to port all of the existing IDL software to Python, would be time consuming and create confusion about which routines are being actively supported during the transition. Distributing level-1 files removes this problem, but does make the user dependent on the team for reformatting all of the files as bugs are discovered. Since the mission has been going on for some time now, the number of bugs is likely to be small.

There are several other design decisions that merit some explanation

- The data and header information are stored in separate files. Since the data is large and unlikely to change, the time-consuming download of these files should only need to be done once. The header file is very small and can be updated easily.

[1] EIS is part of the Hinode mission and was sponsored by the Japan Aerospace Exploration Agency (JAXA), the United Kingdom Space Agency (UKSA), and National Aeronautics and Space Administration (NASA) with contributions from ESA and Norway. Hinode was launched on September 22, 2006 at 21:36 UTC from the Uchinoura Space Center in Japan and continues to operate.

[2] Culhane, J. L., Harra, L. K., James, A. M., et al. 2007, Sol. Phys., 243, 19

- HDF5 is used to store the data. This is a very widely used, high-performance file format that is well supported by both IDL and Python. The most attractive feature for this application is that data is stored in a self-documenting, directory-like tree structure instead of binary table extensions.
- The data is processed from raw "data numbers" to "photon events" or "counts". Conversion to absolute units is done using a calibration curve in the header file, and several different curves can be considered.

The remaining chapters of this document describe the contents of the level-1 files and illustrate how to use these files for data analysis in Python.

Prepping the Data and Writing the HDF5 Files

The level-0 fits files were prepped using the IDL routine `eis_prep` available via SolarSoft (Freeland & Handy, 1998) and the following options

```
default = 1
save = 1
quiet = 1
retain = 1
photons = 1
refill = 0
```

There are 400,000+ EIS level-0 files at present, but on a multi-core machine using the IDL bridge all of the files can be prepped in under 24 hours. We have prepped all of the available EIS files and saved them to standard fits files in the usual way. Some important points

units: As mentioned previously, the units for the output in these level-1 files is "photon events" or "counts." This means that the statistical uncertainty can be estimated as $\sqrt{N}$. Technically the read noise should be added in, but it becomes significant only at very low flux levels (1–2 counts).

retain: Note that the retain keywords preserves negative values. One of the jobs of `eis_prep` is to remove the pedestal from the CCD readout and any time dependent dark current. Since the spectral windows are generally narrow, the estimate of the background can be too high and the subtracted intensities of the continuum can be negative. This will be dealt with during the fitting.

refill: The warm pixel problem complicates the fitting of EIS line profiles. As discussed in software note #13 in SSW, interpolating the values of missing pixels appears to best reproduce the original data. This option is left off during `eis_prep` so that the level-1 fits file preserves the information on the missing pixels. As discussed below, the interpolation (via the refill option) is done during the read and this data is ultimately written to the HDF5 file. A mask indicating which pixels have been interpolated needs to be added to the HDF5 file.

Here is a code snippet related to reading the data by looping over the spectral windows.

```
for iwin=0, nwin-1 do begin
 d = eis_getwindata(eis_level1_filename, iwin, /refill, /quiet)
 eis_level1_data[iwin] = ptr_new(d)
endfor
```

Reading the HDF5 Level-1 Files

We assume that the reader is familiar with installing Python packages using either "pip install" or Anaconda. This software has been tested using a default Anaconda stack with Python version 3.7. In this chapter we will give a very brief overview of how to read the HDF5 files. Subsequent chapters will provide details on specific topics.

This code snippet illustrates how to read the level-1 data from spectral window 7 (containing Fe XII 195.12 Å) in Python. Later in this chapter we'll show how to map a wavelength to a window number.

```
import h5py
import numpy as np

file_data = 'eis_20190404_131513.data.h5'
file_head = 'eis_20190404_131513.head.h5'

f_data = h5py.File(file_data, 'r')
data = np.array(f_data['level1/win07'])
f_data.close()
```

The resulting array is `(512, 87, 24)`. That is, 512 pixels along the slit, 87 steps in the x direction, and 24 pixels in the dispersion direction.

Information from the header file is read in a similar way. For example,

```
f_head = h5py.File(file_head, 'r')
x_scale, = f_head['pointing/x_scale']
date_obs, = f_head['index/date_obs']
wave = np.array(f_head['wavelength/win07'])
radcal = np.array(f_head['radcal/win07_pre'])
f_head.close()
date_obs = date_obs.decode('utf-8')
```

Here `x_scale` is the number of arcsec per step in the raster. Most EIS rasters take more than 1 arcsec per step, which degrades the spatial resolution but increases the cadence. The variable `radcal` is the preflight calibration curve for this data window. It includes all of the factors for converting counts directly to erg cm^{-2} s^{-1} sr^{-1}. The `decode` on `date_obs` converts from a byte array to unicode.

We can make a quick image by summing the data in the dispersion direction. The raster is scaled logarithmically.

```
import matplotlib.pyplot as plt

raster = np.sum(data, axis=2)
range = np.percentile(raster, (1, 99))
range = range[1]*np.array([1.0E-2, 1.0])
scaled = np.clip(raster, range[0], range[1])
scaled = np.log10(raster)

plt.imshow(scaled, origin='lower', aspect=1/x_scale, cmap='gray')
plt.title(date_obs)
plt.show()
```

The following code illustrates how to display the spectrum from a single pixel. To convert from counts to calibrated units we simply multiply by the calibration curve. Note how the array is addressed differently in Python than in IDL.

```
ix = 48
iy = 326
spec = data[iy, ix, :]
spec_cal = spec*radcal

plt.subplot(1, 2, 1)
plt.plot(wave, spec)
plt.title('units = counts')

plt.subplot(1, 2, 2)
plt.plot(wave, spec_cal)
plt.title('units = ergs cm$^{-2}$ s$^{-1}$ sr$^{-1}$')

plt.show()
```

We usually don't want to know about the numbering of the data windows. It's more natural to want to read the data corresponding to a particularly wavelength. The data in the `wininfo` group contains the max and min wavelengths for each data window and allows us to map from wavelength to window number.

```
f_head = h5py.File(file_head, 'r')
nwin, = f_head['/wininfo/nwin']
dt = np.dtype([('line_id', 'U64'), ('wvl_min', 'f'), ('wvl_max','f')])
wininfo = np.zeros((nwin,), dtype=dt)
wininfo = wininfo.view(np.recarray)
for iwin in range(nwin):
   line_id, = f_head[f'/wininfo/win{iwin:02d}/line_id']
   wvl_min, = f_head[f'/wininfo/win{iwin:02d}/wvl_min']
   wvl_max, = f_head[f'/wininfo/win{iwin:02d}/wvl_max']
   wininfo[iwin].line_id = line_id.decode('utf-8')
   wininfo[iwin].wvl_min = wvl_min
   wininfo[iwin].wvl_max = wvl_max
f_head.close()
```

Given a wavelength we can find the window number using the wavelength ranges for each window. Note that we've converted `wininfo` to a numpy record array, which behaves similarly to an array of IDL

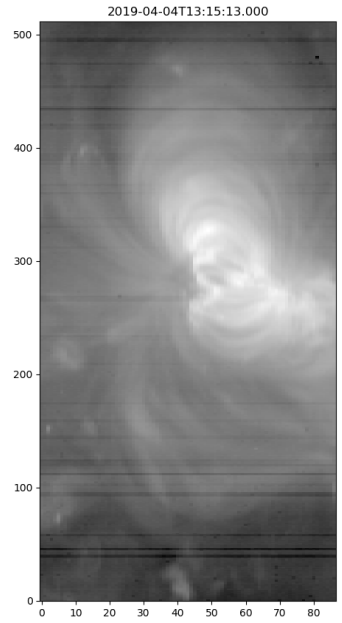

Figure 1: An example image formed by summing the data for the Fe XII spectral window in the dispersion direction. In a subsequent chapter we'll discuss fitting the spectra.

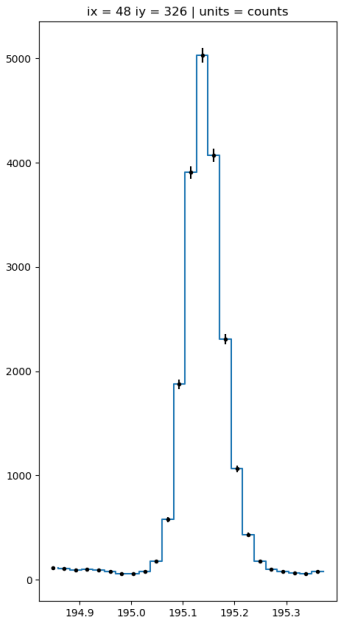

Figure 2: An example Fe XII 195.119 Å line profile from the raster.

structures.

```
wvl = 195.119
p = (wininfo.wvl_max - wvl)*(wvl - wininfo.wvl_min)
iwin, = np.where(p >= 0)
```

If the result is an empty list, the wavelength is not in the data.

Two final notes on reading the data. First, the contents of the HDF5 files can be displayed using `h5dump`, which is provided with Anaconda. For example,

```
> h5dump -n eis_20190404_131513.data.h5
FILE_CONTENTS {
group   /
group   /level1
dataset /level1/intensity_units
dataset /level1/win00
dataset /level1/win01
dataset /level1/win02
dataset /level1/win03
dataset /level1/win04
dataset /level1/win05
. . .
```

The actual data associated with each variable can be printed out using the `-d` option. For example,

```
> h5dump -d exposure_times/duration eis_20190404_131513.head.h5
HDF5 "eis_20190404_131513.head.h5" {
DATASET "exposure_times/duration" {
  DATATYPE H5T_IEEE_F32LE
  DATASPACE SIMPLE { ( 87 ) / ( 87 ) }
  DATA {
  (0): 40.0005, 40.0002, 40.0004, 40.0004, 39.9994, 40.0002, 39.9995, 40,
  (8): 40.0007, 39.9999, 40.0005, 40.0004, 39.9997, 40.0002, 39.9994,
  . . .
}
```

Second, there is a Python object `eis_read_raster.py` that can be used to read the data and head files. For example,

```
from eis_read_raster import eis_read_raster
filename = 'eis_20190404_131513.data.h5'
wave = 195.119
eis = eis_read_raster(filename, wave)
print(eis.data['index']['date_obs'])
print(eis.data['data'].shape)
```

In the chapters that follow we will use the object to do most of the heavy lifting. The routine `eis_display_window.py` illustrates how to use the object and matplotlib to click around the raster and display individual line profiles

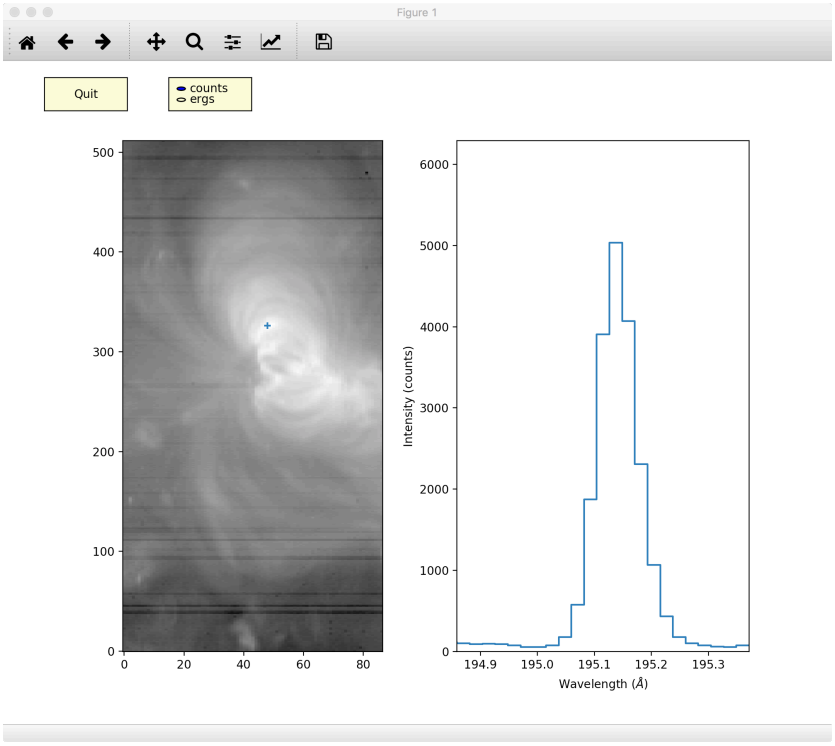

Figure 3: An example

Fitting the Spectra

Fitting of the spectra involves selecting a spectral line of interest (e.g. Fe XII 195.12 Å) from the spectral windows of the data and determining a guess on the fit parameters. The next ingredient for a fit is the selection of an optimization method[3].

For this we've created a set of fit templates for different spectral lines. An `h5dump` on the file shows that it contains a `/template` group for the initial guess on the fit parameters and a `/parinfo` group containing constraints on the parameters for `mpfit.py`.

```
h5dump -n fe_12_195_119.2c.template.h5
HDF5 "fe_12_195_119.2c.template.h5" {
FILE_CONTENTS {
 group   /
 group   /parinfo
 dataset /parinfo/fixed
 dataset /parinfo/limited
 dataset /parinfo/limits
 dataset /parinfo/tied
 dataset /parinfo/value
 group   /template
 dataset /template/component
 dataset /template/data_e
 dataset /template/data_x
 dataset /template/data_y
 dataset /template/fit
 dataset /template/fit_back
 dataset /template/fit_gauss
 dataset /template/line_ids
 dataset /template/n_gauss
 dataset /template/n_poly
 dataset /template/order
 dataset /template/wmax
 dataset /template/wmin
 }
```

The object `eis_read_template.py` can be used to read a template file and examine the contents.

```
from eis_read_template import eis_read_template
filename = 'fe_12_195_119.2c.template.h5'
template = eis_read_template(filename)
```

This produces the output below, showing the `/parinfo` group that contains parameters (peak, centroid, width, background) for a double

[3] Here we use a Python implementation of the well-known IDL method mpfit which solves the non-linear least squares problem using the Levenberg-Marquardt algorithm. The Python implementation mpfit.py is found on GitHub (https://github.com/segasai/astrolibpy/) and included in our analysis software.

Gaussian fit along with the parameter constraints. Note that this is specific to using the `mpfit` method (see the GitHub page for more info).

```
+ template file = fe_12_195_119.2c.template.h5
*PARAMETER CONSTRAINTS*
*          Value    Fixed        Limited           Limits          Tied
 p[0]   57514.6647     0       1       0     0.0000    0.0000
 p[1]    195.1179      0       1       1   195.0778  195.1581
 p[2]      0.0289      0       1       1     0.0191    0.0510
 p[3]   8013.4013     0       1       0     0.0000    0.0000
 p[4]    195.1779      0       1       1   195.1378  195.2181      p[1]+0.06
 p[5]      0.0289      0       1       1     0.0191    0.0510      p[2]
 p[6]    664.3349      0       0       0     0.0000    0.0000
```

Next you'll want to prep the data for fitting. Once you've read in a template file, you can use the central wavelength to find the desired spectral window in the data using `eis_read_raster` as shown in the previous chapter.

```
from eis_read_raster import eis_read_raster
from eis_read_template import eis_read_template

# input data and template files
file_data = 'eis_20190404_131513.data.h5'
file_template = 'fe_12_195_119.2c.template.h5'

# read fit template
template = eis_read_template(file_template)

# get central wavelength
wmin = template.template['wmin']
wmax = template.template['wmax']
wave = wmin + (wmax-wmin)*0.5

# read raster
raster = eis_read_raster(file_data, wave)
ints = raster.data['data']
wave = raster.data['wave']
corr = raster.data['wave_corr']
```

Prepping of the data can be handled at various levels of sophistication at the user's discretion, however, at a minimum it should include handling bad values[4] in the raster, correcting for the wavelength offsets[5], and computing the errors on the intensities[6].

```
# get dimensions
ndata = ints.shape
nx   = ndata[0]
ny   = ndata[1]
nz   = ndata[2]

# bad data correction
bad = np.where(ints<0)
ints[bad] = 0.0
```

[4] Negative values are a result of the background subtraction.

[5] This is from thermal shifts in the CCD and normal degradation of being in space over time?...

[6] The square root of the counts is a good first-order approximation.

```
# compute error on counts
errs = np.sqrt(ints)

# wavelength correction
newwave = np.zeros(ndata)
for i in range(nx):
   for j in range(ny):
      newwave[i,j,::] = wave-corr[i,j]
wave = newwave
```

Now on to the fitting! Now that you have a fit template and the data elements, you can perform a fit of the entire raster by calling `eis_fit_raster.py`[7]. The fit results can be saved and read back using `eis_save_fit.py` and `eis_read_fit.py`.

[7] Here's what's happening under the hood. The object eis-fit-raster calls eis-scale-guess to scale the initial parameter guess to the data, then calls eis-mpfit to implement the Levenberg-Marquardt fitting. The module eis-fit-deviates contains the callable function that returns the fit deviates computed from a model function for eis-mpfit.

```
from eis_fit_raster import eis_fit_raster
from eis_save_fit import save_fit
from eis_read_fit import read_fit

# fit profile
parinfo = template.parinfo
template = template.template
fit    = eis_fit_raster(wave, ints, errs, template, parinfo)

# save fit output
fit = fit.fit
file_fit = save_fit(fit, file_data)

# read fit output back from file
fit = read_fit(file_fit[0])
```

The output fit parameters are stored to a dictionary.

```
background float64  (512, 87, 1)
centroid  float64   (512, 87, 2)
chi2      float64   (512, 87)
component int64     1
e_background float64 (512, 87, 1)
e_centroid float64  (512, 87, 2)
e_int     float64   (512, 87, 2)
e_peak    float64   (512, 87, 2)
e_width   float64   (512, 87, 2)
int       float64   (512, 87, 2)
line_ids  object    (2,)
n_gauss   int16     1
n_poly    int16     1
params    float64   (512, 87, 7)
peak      float64   (512, 87, 2)
perror    float64   (512, 87, 7)
status    float64   (512, 87)
wavelength float64  (512, 87, 24)
width     float64   (512, 87, 2)
```

The above steps are illustrated in the example routine `eis_fit_example.py`, which produces a plot like the one shown below.

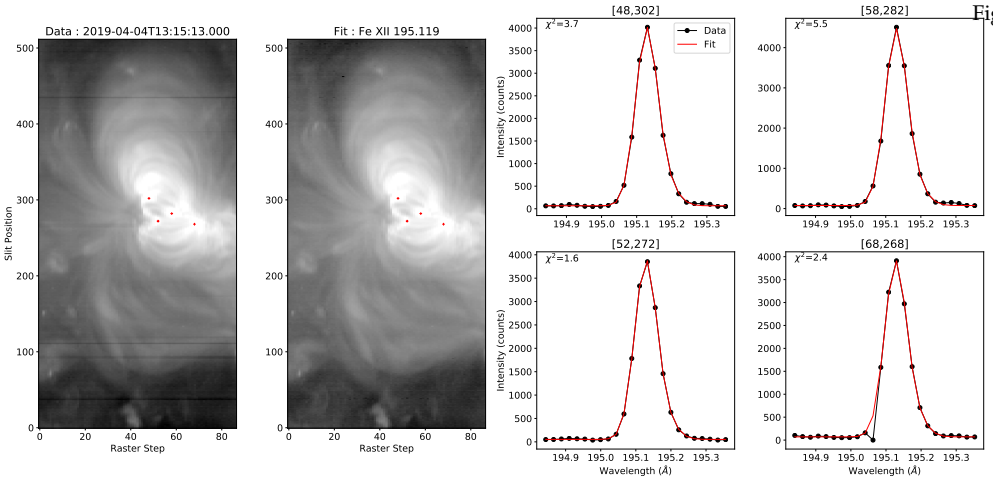

Figure 4: Fit Examples.

Bibliography

Culhane, J. L., Harra, L. K., James, A. M., et al. 2007, Sol. Phys., 243, 19

Freeland, S. L., & Handy, B. N. 1998, Sol. Phys., 182, 497