The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XLII-3/W3, 2017
Frontiers in Spectral imaging and 3D Technologies for Geospatial Solutions, 25–27 October 2017, Jyväskylä, Finland

SOFTWARE FRAMEWORK FOR HYPERSPECTRAL DATA EXPLORATION AND PROCESSING IN MATLAB

Matti A. Eskelinen

University of Jyväskylä, Faculty of Information Technology, Jyväskylä, Finland - matti.a.eskelinen@student.jyu.fi

Commission III, WG III/4

KEY WORDS: Software, Hyperspectral Imaging, Data Exploration, MATLAB

ABSTRACT:

This paper presents a user introduction and a general overview of the MATLAB software package `hsicube` developed by the author for simplifying the data manipulation and visualization tasks often encountered in hyperspectral analysis work, and the design principles and software development methods used by the author. The framework implements methods for slicing, masking, visualization and application of existing functions to hyperspectral data cubes without the need to use explicit indexing or reshaping, as well as enabling expressive syntax for combining these operations on the command line for highly efficient data analysis workflows. It also includes utilities for interfacing with existing file reader scripts for easy access to files using the framework. The `hsicube` framework is released as open source to promote the free use and peer review of the code and enable collaborative development.

1. INTRODUCTION

Hyperspectral imaging data presents a challenge for data exploration due to it's high dimensionality, size and very application-specific features of interest. Programming algorithms and processing pipelines for analysis and visualization of hyperspectral data requires the programmer to keep track of multiple variables besides the data, such as wavelength information and regions of interest. Care must be taken when translating between data exploration and batch processing workflows to ensure correct results, and again when visualizing results of the batch processing to prevent erroneous presentations. Automation of the steps of such workflows — while highly desirable — is often a complex programming task due to the management of many datasets of different dimensions along with their metadata. Consider for example the following workflow:

1. Read a hyperspectral cube and its metadata from disk,
2. Extract data for region(s) of interest,
3. Select data in a specific wavelength region(s),
4. Compare select spectra visually with reference data,
5. Apply a model on the selected data and collect the results,
6. Visualize the results overlaid on the original dataset.

Steps 1 and 5 are cases that existing MATLAB libraries such as `ENVIreader/writer` (Totir and Howat, 2010) and `Hyperspectral ToolBox` (Gerg, 2016) are built to tackle, while the rest of the steps are usually constructed piece-by-piece using the basic MATLAB functionality. However, since MATLAB does not provide any smart datatypes for combining metadata with multi-dimensional arrays, manual bookkeeping is necessary in many of the steps if one wishes to keep the metadata synchronized with the operations done to the array. Such bookkeeping is a major source of errors and a large time expenditure for the programmer, and easily results in repetetive code if not properly abstracted. This in turn makes working inside a REPL (read-eval-print-loop) such as the MATLAB command line or iPython prohibitively cumbersome, which is detrimental to rapid prototyping of new algorithms.

Object-based datatypes that encapsulate both the data and metadata along with methods to operate on the data are a common abstraction for problems of this kind in many languages. For example, open source solutions like *Spectral Python* (Boggs, 2016) or *xarray* (Hoyer and Hamman, 2017) exist for manipulating hyperspectral or general multidimensional data in Python, but no similar datatype implementations exist for MATLAB.

This paper showcases a framework for simplifying hyperspectral data analysis workflows in a similar way using MATLAB. The framework package is mostly feature complete for the author's current workflow and a development version has already been used for the hyperspectral analysis in (Salmi et al., 2017). General design principles of the framework are detailed in section 2, with the internal implementation along with the main properties of the class listed in section 3. The implemented methods for simplifying the steps of the example workflow are presented in section 4. Section 5 contains a note about the testing methodology used for ensuring code quality, and section 6 has notes on acquiring and using the code.

2. GENERAL DESIGN

The guiding principle in the access to the internal object properties is that of minimal privilege, meaning that the programmer using the data type is purposely unable to access the object in a way that could de-synchronize the metadata and the data (at least without considerable effort). This is a design choice of the author to favor data integrity over malleability, and in contrast with the approach taken by many libraries which often allow most unsafe operations, but recommend not using them. However, the author feels that recommendations rarely deter programmers looking to

This contribution has been peer-reviewed.
https://doi.org/10.5194/isprs-archives-XLII-3-W3-47-2017

Table 1. Data type object properties

Property	Description	Example content
Data	Data array	`ones(10,10,100)`
Files	File(s) of origin	`{'data.hdr'}`
Quantity	Quantity of the data	`'Reflectance'`
WavelengthUnit	Unit of the wavelengths	`'nm'`
Wavelength	Vector of wavelengths	`[400 ... 500]`
FWHM	Vector of FWHMs	`[3, 4, 3, ..., 3]`
History	History of operations	`{'Read from file'}`
Version	Class version	`'0.8.0'`

do the easy thing instead of the right thing, and as such prefers the implemented technical solution.

Following from this, all the object properties are accessible read only, and can only be assigned new values during after object construction through the object various object methods that validate the input for the specific purpose. In addition, methods that manipulate the data but cannot generate meaningful default metadata for the results, require the user to supply valid metadata instead of generating non-meaningful defaults.

There has also been some effort to standardize the format of class method arguments to improve the user experience. As a notable divergence from the usual MATLAB syntax, the class methods use the syntax `[x,y]` for indexing horizontal and vertical pixel coordinates instead of the usual MATLAB syntax of `[y,x]`. This makes it easier to relate indices to the MATLAB image visualizations.

3. INTERNALS

The main content of the software package is a MATLAB object class named `Cube` which contains as properties the hyperspectral data and relevant metadata and provenance for the data. The properties implemented in the current version are listed in table 1. The properties are read-only after the creation of the object, and can only be changed using the object methods to ensure validity and synchronization during operations.

The object class is implemented as a MATLAB value class, which means that each object method returns a copy of the object instead of a reference (along with any other return values). This makes the state of the object easier to reason about, but relies on the MATLAB memory management to optimize memory use. The internal object methods do however try to minimize the creation of temporary extra copies of the data where possible.

The argument parsing for the Cube class constructor (see 4.1) is implemented flexibly using the class CubeArgs, which is subclassed from the MATLAB InputParser class. Due to the amount of development effort required to implement this kind of functionality in MATLAB, the "Name, value" syntax is currently only implemented for the constructor method.

Some utility functions, which are not specific to, but used by the Cube class are separated into a separate namespace in the static class Utils. ENVI file format reading functionality is similarly implemented as a static class ENVI which contains wrappers around the existing `ENVIreader/writer` functions.

4. FUNCTIONALITY

This section will detail some of the main features of the class and provide examples of their usage. It is not intended as a full API reference, but as more of cursory look at common MATLAB operations on multidimensional data and their implementation in the Cube class.

4.1 Constructing Cubes

Given a hyperspectral data cube (or any 3-dimensional array), the construction of a Cube object is straightforward. For a simple example, the syntax

```
c = Cube(ones(10,10,16));
```

creates a Cube object with the given array and default metadata (vector of band indices 1 to 16 for `Wavelength`, vector of zeros for `FWHM`, and `Quantity` set to the string "Unknown"). Metadata can be specified to the constructor using the usual "Name, value" syntax used by many MATLAB functions. As an example, one could construct a mockup radiance Cube with actual wavelength metadata as

```
mockup = Cube(ones(10,10,25), ...
     'qty', 'Radiance', ...
     'wlu', 'nm', ...
     'wl', 501:525, ...
     'fwhm', 5*ones(1,25));
```

The possible arguments can be found from the documentation of the `Cube()` constructor method or by directly inspecting the argument parser class CubeArgs.

4.2 Slicing

Instead of the usual MATLAB slicing syntax for multidimensional matrices, the Cube class implements different methods for slicing different dimensions. While slightly more verbose, this makes code much more readable, especially when combining multiple operations on a single line.

- For spatial slicing, the current implementation has the method `crop(tl,br)`, which takes in the top left and bottom right corners of the desired area as 2-element vectors of the pixel coordinates, and returns the rectangular region defined by the corners. For more advanced region selection, users should employ the mask and unmask functionality shown in section 4.5. For a simple spatial crop of 20 pixels on each side of the image, one could supply the following syntax:

  ```
  cropped = cube.crop(...
    [20,20], ...
    [cube.Width-20, cube.Height-20])
  ```

  Note here the usage of the Cube properties `Width` and `Height` to extract the dimensions, which makes the code very readable at only a slight cost to terseness.

- The method `px([x,y])` selects individual pixels based on their pixel coordinates. It is possible to supply multiple coordinates, in which case the resulting Cube will be a list ($N \times bands$ matrix) of the spectra in those coordinates.

- For band selection, the method `bands()` takes in a vector of band indices (between 1 and the number of bands in the data) and returns a Cube object with the specified bands. The syntax allows for some more manipulations besides just selection of existing bands: For instance, it is possible to replicate bands by supplying the same index multiple times. There is currently no direct support for selecting

wavelengths explicitly. Instead, `bands()` also accepts logical vectors for selecting bands, which allows selection of certain wavelengths by e.g. the following syntax using the `Wavelength` property of the Cube:

```
longer_wavelengths = ...
    cube.bands(cube.Wavelength > 1000);
```

Both syntaxes will result in Cubes that have their other properties appropriately modified, with only the wavelength and FWHM information corresponding to the selected band left preserved in the metadata.

For comparison, listing 1 demonstrates the amount bookkeeping required for selecting wavelengths from a dataset without the Cube class, and keeping the metadata synchronized at the same time. It also demonstrates the namespace pollution that is hard to avoid without more specialized data containers that reduce the need for separate variables for in-memory data management.

Listing 1. Selecting data by wavelength with plain MATLAB

```
wl   = 501:1000;            % mock wavelengths
fwhm = 5*ones(500);         % mock fwmh data
g    = rand(100, 100, 500); % random datacube

% Selecting bands corresponding to 600-800 nm range
idx = wl >= 600 & wl <= 800;
g2 = g(:, :, idx);
wl2 = wl(idx);
fwhm2 = fwhm(idx);
```

4.3 Arithmetic

Basic arithmetic (namely the operators `+,-,*,/`) is currently implemented by overloading the operators for the Cube class. The `Quantity` parameters of the argument cubes are by default naively combined to denote the new quantity, i.e. dividing a radiance cube by a radiance cube will produce a cube with quantity "radiance / radiance". However, since MATLAB allows extra arguments to overloaded functions, it is possible to use the syntax `oper(a,b,quantity)` to supply the result quantity as an extra argument. As an example with the radiance Cubes, one might wish to calculate reflectance using the following syntax:

```
refl_cube = div(cube, white_ref_cube, ...
        'Reflectance');
```

The resulting Cube would then contain the data in `cube` divided elementwise by that of `white_ref_cube`, with the `Quantity` set to the string "Reflectance".

In the current implementation, the operators are explicitly restricted to Cubes of the exact same dimensions (also erroring on any arrays that are not Cubes). This is in contrast to the normal MATLAB operators on numerical arrays, which in the 2017a version do automatic expansion of the argument arrays (using `bsxfun`). This is due to the fact that `bsxfun` does not necessarily replicate the arguments along a dimension that is sensible for a hyperspectral data cube. For instance, if one were to add a constant vector (like a bandwise correction to the spectrum) to a full data cube, the correctness of the result would depend on which of the three dimensions of the cube would match the length of the vector first, and could result in errors for cubes with two dimensions with the same length.

Other mathematical operations that one might want to apply on a single cube (such as multiplication by a scalar, logarithms etc.) can be applied by using the existing functions with the family of map functions detailed in section 4.4.

4.4 Function application

For more functionally inclined programmers, the Cube class implements methods for applying given functions on the Cube data without the need for explicit deconstruction of the Cube object. For applying functions expecting various dimensions of data, three different functions are implemented:

- The method `map(f, ...)` may be used to apply a given function `f` on the whole data cube. If the function changes the data in a meaningful way, the user is expected to supply the new metadata for the result as separate parameters using the Name, value syntax of the Cube constructor.
- `mapSpectra(f, ...)` applies the function `f` on the data after reshaping the hyperspectral data cube into a list of the spectra, and after application reshapes the result into the original spacial dimensions of the data (with possible change in the number of bands). This is equivalent to the functionality demonstrated in section 4.5, but using the full data cube instead of selected parts of it.
- `mapBands(f)` applies the function `f` to each band of the data cube by looping through each layer in turn, applying `f` and collecting the results. This provides an easy way to apply filtering operations that are not dependent on the wavelength, but due to the looping (which there is in this general form no easy way to avoid), it may be preferable to implement more costly filter directly on cubes and apply them using `map`.

4.5 Masking and unmasking

For selecting regions of interest (especially non-rectangular ones) from a Cube, the class implements a method called `mask`. Given a binary mask image (logical matrix) with a size matching the spatial dimensions of the Cube, it reduces the Cube to a $N \times bands$ matrix of pixel spectra selected by the mask, with N the number of True pixels in the mask image. The rationale for the list form of the output is compatibility with machine learning due to the fact that the $N \times bands$ matrix is of the form $samples \times features$ expected by most existing machine learning applications, which makes it very fast to use `mask()` to extract data for classification tasks.

The `unmask()` method is used for applying the reverse transformation: Given a mask image with N True pixels, it reorganizes an $N \times bands$ (or $samples \times features$) Cube into a full Cube with the spatial dimensions of the mask image, with zeros in the False mask regions and the data in the True region. If the data in the list is in the same order as returned by the corresponding `mask()` operation, applying `unmask()` after `mask()` with the same mask image results in having a mask applied on a data cube by zeroing any False entries on each layer.

This along with the `map()` methods (4.4) allows for very concise applications of existing algorithms on hyperspectral cubes. For instance consider the example workflow of applying PCA to masked part of the image in listing 2. For comparison, the same workflow without the abstractions provided by the Cube class is also shown.

Listing 2. Performing PCA on a subset of pixels

```
% g is a datacube with the h*w pixels and b bands
[h, w, b] = size(g);
```

```
% Construct a Cube from the data
c = Cube(g);
% Dummy mask that selects the diagonal
im = eye(c.Height, c.Width);

% Application of PCA using masks and map
c_scores = c.mask(im).map(scores).unmask(im);

% Helper function to extract the scores from PCA
function [y] = scores(x)
    [~, y] = pca(x);
end
```

4.6 File operations

For convenience, the implementation includes a separate class for wrapping `ENVIreader/writer` operations with those of the Cube class. The included ENVI class has methods for reading and writing ENVI files directly to Cube objects, filling in all the Cube metadata fields directly from the ENVI header file. A write wrapper method has also been implemented for directly writing both the data and metadata from a Cube object to an ENVI file. Other file formats can easily be added using similar wrappers for existing functionality.

4.7 Visualization

For visualization, the class implements a few plotting and image viewing methods to make data exploration easier. The main advantage over using the MATLAB visualizations directly is that the Cube class can utilize its metadata to assign most axis ticks, labels and titles appropriately in the produced figures. If found in path, they also employ the MATLAB version of Colorbrewer (Cobeldick, 2017) colormaps for better representation of the data. The main visualizations are provided by the following methods:

- The method `im(b)` displays a single band `b` as an image using the MATLAB function `imagesc`.
- `rgb(r,g,b)` displays the bands at indices `r, g` and `b` as a three-color image using MATLAB `imshow`.
- `plot()` plots the spectra using MATLAB `plot()`. It also automatically restricts the number of spectra to display in order to prevent MATLAB from freezing when trying to plot too many curves at once.
- `hist()` calculates histograms of each band layer in the Cube, combines them and displays them using `surf` in combination with a colormap for a versatile visualization.

The visualizations always return the Cube they were visualizing, which makes it easy to both extract the result of a Cube operation and visualize it using a single line of code. For example, the following line would visualize the first band of the cube, apply a crop and then display the same band in the cropped image in a new figure, while also extracting the result as a new Cube:

```
tl = [20,20]; % top left
br = [100, 100]; % bottom right
cropped = c.im(1).crop(tl, br).im(1);
```

This generally makes data exploration effortless and easy to integrate with follow-up scripting.

4.8 Provenance

Apart from the visualizations, all the Cube methods that operate on the data append a string representation of their action to the `History` property of the returned Cube. This allows those of us with less-than perfect memory of our command line usage to inspect the operations performed on the Cube objects in the MATLAB workspace by simply looking at the strings stored in the property of each object. In the current implementation, there is however no inbuilt way to store the history of each Cube in a file apart from saving the whole object to a file, which is in general a fragile procedure due to the way MATLAB handles object loading.

5. TESTING

The class constructor and methods have unit tests written for the using the MATLAB unit testing framework. Integration testing of method and constructor interoperations has not yet been implemented apart from some individual cases. The tests are included in the release version of the package, and can be run by users on their MATLAB installation to ensure compatibility with their given version of MATLAB. People interested in the framework are invited to submit test cases and bug reports to the author (preferably through pull requests to the Github repository, see section 6).

6. OBTAINING THE CODE

At the time of writing, version 0.8.0 of the `hsicube` framework is available from the authors Github page[1] under the MIT license. The package contains the methods needed to directly read ENVI files under the `ENVI` module, however this functionality requires one to have the `ENVIreader/writer` (Totir and Howat, 2010) in their MATLAB path, which is not included in the package and must be acquired separately. Similarly, for the nicer Colorbrewer colormaps (Cobeldick, 2017) the required package needs to be acquired separately and placed in path before utilizing the visualizations.

REFERENCES

Boggs, T., 2016. Spectral python. Software. Available from http://www.spectralpython.net.

Cobeldick, S., 2017. Colorbrewer: Attractive and distinctive colormaps. Software. Available from https://se.mathworks.com/matlabcentral/fileexchange/45208-colorbrewer–attractive-and-distinctive-colormaps.

Gerg, I., 2016. Matlab hyperspectral toolbox. Software. Available from https://github.com/isaacgerg/matlabHyperspectralToolbox.

Hoyer, S. and Hamman, J., 2017. xarray: N-d labeled arrays and datasets in python. *Journal of Open Research Software*.

Salmi, P., Eskelinen, M. A., Kremp, A. and Pölönen, I., 2017. Constructing absorbance spectra and abundance maps of micro- and nanoalgae by transmission hyperspectral imaging of liquid cultures on petri dishes. *PLOS One*. Submitted.

Totir, F. and Howat, I., 2010. Envi file reader/writer. Software. Available from https://se.mathworks.com/matlabcentral/fileexchange/27172-envi-file-reader-writer.

[1] `https://github.org/maaleske/hsicube`