4D Image Matching

This is a draft framework for the “4D” project under the NSF’s ICEBerg Cyber-Infrastructure grant. The purpose of the 4D Image Matching algorithm is to match an effective group of keypoints between two landscape images taken at different times, possibly separated by decades or more. Presumably, one of the images (the “target” image) is from a modern platform and is geo-located, allowing the geolocation and ortho-rectification of the “source” image with respect to the target image.

The eventual goal is to geo-locate large numbers of historical airborne photos against a database of modern, high-resolution, geo-located satellite images. Separately, by generating 3D topography from the historic airborne photos, we can create 4-dimensional maps (3D, over time) of change in terrain over important geophysical landscapes on Earth.

## Overview

In its current state, the 4D algorithm works in four different phases, with code organized into categories, by order of execution. Other phases may need to be added when scaling up from 2 images to searches across multiple images and multiple airborne flight lines. These are high-level descriptors: specific details and usage are available in the “Readme.md” included in the code sub-directory of each individual phase.

### Phase 1: Image Pre-processing

Image pre-processing that must take place before keypoint-matching begins:

* Tile Planner: Images are divided into “tiles,” the details of which are stored in a “Tiles” CSV file in a Scratch directory. Tiles are optimized to stay within a range of sizes, minimize “no data” portions of the image, and avoid “sliver” tiles at the edges of images. Individual tiles may be written to disk. Either way, their information is saved for later use in the “Tiles\_XXX.csv” files in a folder in the Scratch Directory.
* Tile Contrast Enhancer. Tile may have their contrast enhanced to aid the keypoint matching algorithms, and/or to optimize matching in particular areas of the image (example: rock) over others (snow and/or water).
* Pyramid Builder (optional): Before tiling, images may be downscaled into pyramids, to perform image matching on lower-resolution versions of the image (useful for initial processing). Pyramid images are saved in the same Scratch directory as the tiles.
* Scratch Directory Manager: A utility for organizing sub-folders in the Scratch directory. These routines are useful to save temporary results for images that may be re-used (i.e. an airborne “source” photo that will be searched over multiple satellite “target” images, and vice-versa). Tiles may be written once and re-used on multiple runs before being deleted.
* Run Case Planner: Take the lists of tiles output by the “Tile Generator” for both the source and target images, and matches them into pairs to be executed against each other in the next phase. Saves the output to a new CSV file. If the user/coder only wishes to select certain subsets of tiles to run against each other (only at certain pyramid levels, or only one “target” tile against which to search all the “source” tiles, e.g.), this is the place where those selections are made.

### Phase 2: Keypoint Generation

The execution of finding keypoints between image tiles. (It should be noted that many of the terms here refer to the ASIFT algorithm, although other algorithms are now also being used either in conjunction with or instead of ASIFT. Specific references to ASIFT are legacy.) The following steps are included.

* ASIFT Executable: A standalone set of functions for matching keypoints between two individual tiles. This is the code that does the majority of the work. Results are saved in the Scratch sub-Directory of the “source” image, including a “data\_matches.csv” file. This version of the code uses the “fast\_imas\_IPOL” executable, contained in the “fast\_imas\_IPOL” directory; however, other implementations (such as CUDA functions on a GPU node) can be swapped out here.
* ASIFT Class Definitions: A utility of class definitions used by the ASIFT Executable script.
* ASIFT Scheduler: A set of functions to handle the execution of all the “run cases” output by the “Run Case Planner” in the last phase. On a desktop workstation these could be run in serial, for instance. In an HPC environment, individual cases would be farmed out to individual nodes until execution is complete.
* ASIFT Performance Monitor: Measures performance metrics of the ASIFT Executable (time, memory), useful for internal optimization

### Phase 3: RANSAC Filter

Outputs from the ASIFT Executable have a large # of “false positive” matching keypoints. An implementation of the RANSAC algorithm filters these points to generate a subset of keypoints with higher confidence.

(NOTE: Some RANSAC results, especially in poorly-fit images, generate non-optimal homography matrices that require unrealistic distortions between source & target images. These can be filtered out, but this is not yet implemented).

* RANSAC Filter: Starting with the “data\_matches.csv” output from the ASIFT Executable in the last phase, this runs a single RANSAC filter on the results, saving the outputs in a “data\_matches\_FILTERED.csv” file.
* Matches Reader: A small utility for reading the “data\_matches.csv” outputs from the ASIFT Executable and creating OpenCV Keypoint objects for the RANSAC filter.
* DrawMatches: A utility for drawing image pairs and writing them to disk, using filtered or unfiltered results. Useful for visualization of results.
* KeyPoint Recombiner (not yet implemented): Takes RANSAC results between individual tiles of the “source” and “target” images, recombines them, transforms the coordinates back into whole-image-space. Performs another RANSAC filter to maintain internal geometric consistency between the image matches. Outputs a final matches CSV to output back to the user. (Optional: If the target image is geo-located, provides image coordinates as well as geographic coordinates of keypoints in the source image.)

### Utilities

Some basic utility functions are provided in the “UTILITIES” directory, useful to multiple scripts in different phases:

* CSV Auto Reader: Reads a simple CSV (comma-separated text) file with a top-line header, auto-detects the data type in each column (text, int, float), and returns a numpy array with named columns containing the data in the CSV file.
* CSV Writer: Takes a numpy array with named columns, or a list of arrays and a separated list of column names, and writes the output to a CSV text file.
* Eprint: Simple function for printing output to stderr rather than stdout.

## Installation:

(Notes to install & run 4D use case.)

## Running:

(Notes to run whole-system code for the 4D use case.)

Options for running individual components of each phase are contained in “Readme.md” for each of the code subdirectories for that phase.

## Dependencies

The code requires Python 2.7+ to execute. (A Python 3 version may be forthcoming soon.)

Code within the phases depends upon the following external Python libraries:

* GDAL v2.2+ : the GeoSpatial Data Abstraction Library ([www.gdal.org](http://www.gdal.org)), and associated Python “gdal” and “osgeo” bindings
* OpenCV v3.3+ : Open Computer Vision libraries and associated Python “cv2” bindings

## License

(TODO: Fill in the open-source license to be used here.)

## Acknowledgements

This project is funded by the NSF’s ICEBERG Cyber-Infrastructure grant.

Acknowledgements to Mariano Rodriguez for the “fast\_imas\_IPOL” executable (<http://github.com/rdguez-mariano/fast_imas_IPOL>), which CPU-versions of this code rely upon.

## Revisions:

1.0 – 2019.02.10 – Draft document by Mike MacFerrin