UAV Processing README

Rob Holman

June, 2016, Version 2.0

1. Background:

UAV's offer a complementary method to collect Argus-like data without installing an Argus station. In some ways the process is much simpler and more flexible but it comes at a disadvantage of much increased book-keeping and case-by-case processing. This README offers some guidance and suggestions on how to handle this in an organized and predictable way so that the data locations and organization is obvious even well into the future after details have been forgotten. These notes describe the algorithm in the CIL toolbox UAVProcessing and the CIL standards for data storage and archiving.

Note that this would be a good case for using a gui, if someone would make one.

This README is also a living document. Send suggestions to Rob Holman.

2. Data Organization:

We elected to base our organization on the long-proven Argus organization. Thus, data are kept under /ftp/pub with a station name taken from the UAV. For illustration, we will use the case of Aerielle, a DJI Phantom 3 quadcopter. Folders under /ftp/pub/Aerielle contain lens calibration images and results as well as stationInfo, in this case the history of the UAV and usage and any other relevant details.

Keeping with Argus conventions, data are stored next under year/camera/day. So for example, /ftp/pub/Aerielle/2015/cx/280_Oct.07 contains derived image products (cx) for flights from October 7, 2015 including rectified image products like time exposures and time stacks if these have been created. Folder names can be personalized in the inputFile (described below). Similarly, the routine argusFilename takes care of constructing useful filenames and is included in the toolbox in the folder 'neededCILRoutines'. If you don't have your own version, please move this from that folder into CILProcessing (or somewhere on your path).

Typically, the actual data collection for a run will consist of a snapshot followed by a movie of sufficient length to perform cBathy analysis (hopefully 17.1 minute). The snapshot is taken as a simple record of the scene and also because it will be geotagged, so can be used as a source of estimated lat-long of the parked UAV. For a phantom 3 these geotags appear to be quite useful (~5-10 m accuracy) but for the Phantom 2 they are truncated at integer seconds so are only accurate to ~30 m. The movie is often too long to be stored as a single MP4 file so might be split into two consecutive files automatically by the Phantom. Note that DJI files are stored with simple names like DJI_0001 so it is the responsibility of the user to manually convert these into more useful names and store them in an appropriate location.

Because MP4 videos are usually multi-gigabyte files, they would rapidly overwhelm our online disk storage capabilities (in the same sense that we don't save 30 Hz full-frame video from Argus stations). Thus our choice is that these are stored offline on a LaCie USB disk under some obvious structure like /media/Lacie/Aerielle/bathyDuckDataCollects/100615/exampleFolder (where exampleFolder is the specific folder name for one data collection, usually including at least one snap and one or more mp4 movies). To access the individual 2 Hz frames from the 30 Hz movie to do Argus-like analysis, we use the routine loadAndPartitionMovies which partitions the movie into 2Hz frames using the command "every 15 soureFn destinationFn" where sourceFn and destinationFn are your source and destination filenames ("every" is a local Perl script that used the mplayer video player routines. Users are responsible for breaking out their own frames). For the CIL, we store frames as a series of png's in a scratch area, e.g. /scratch/temp/Holman/Aerielle/2015/localFolderName. This component of operations will vary to suit the needs and tools of each research group. However, we have found that using the Argus conventions for derived products greatly simplifies our logistics.

3. Processing Organization:

Data analysis involves two major components, reading in and geolocating each of the N frames, then the creation of image products and the extraction of pixel time series for a number of designed pixel instruments. The latter step is like Argus sampling. Because each collect is different from the previous, this requires specification of a number of inputs, described in the next section.

The analysis uses a number of structures to simplify the variable space. These include:

- gcp - ground control points, described in appendix A
- insts - pixel instruments, described in appendix B
- inputs - described below
- meta - metadata, built during analysis and saved for the record in cx (appendix C)
- stack - temporary stack data structure.
- Images - collection of regular rectified Argus image products.

Geo-location requires solving the image geometry for each frame. This is a too complicated to describe in depth here so the user is referred to texts like Hartley and Zisserman [2003] or a soon to be completed companion paper that will be included in the toolbox ("Surf Zone Characterization Using a Small Quadcopter: Technical Issues and Procedures"). But it is important to have at least a broad understanding of the process.

Solution requires knowing things about the physical camera and lens, called the intrinsic parameters, and also things about the installation, called the extrinsic parameters. The intrinsic parameters are found by lens calibration in the lab and are well handled by free toolboxes available on the web. We use the excellent Caltech Lens Calibration software (http://www.vision.caltech.edu/bouguetj/calib_doc/) with those results implemented through a lens calibration profile (lcp) structure that is created by the m-file makeLCPP3.m. You will need to modify this routine with entries for your camera.

There are six extrinsic parameters, the x, y, and z locations of the camera and the three viewing angles, azimuth (taken here as the compass-like rotation clockwise from the positive y-axis), tilt (zero at nadir, rising to 90° at the horizon), and roll (rotation about the look direction, positive in the counter-clockwise direction as viewed from the camera). If you know all of these parameters accurately, you can geo-locate object in images. If you do not, you need to do a least squares solution for those you don't know using ground control points (GCPs, points in the image that you know the world location of and can also locate by cursor in the image). Since each GCP located by clicking in the image provides 2 pieces of input (U and V coordinate), we must see enough GCPs so that twice that number exceeds the number of unknowns. Thus for six unknowns we require 4 GCPs (2*4 > 6). In fact, these points cannot also be collinear and they really should be spread across the image for a good solution. For terrestrial work this is usually possible, but for surf zone work we may see only one or two identifiable points on the shore, near the edge of the field of view. Thus we often must find alternate sources for these parameters so that we can reduce the number of degrees of freedom below 6 (and hopefully remove the non-collinear restriction).

It is rare to find sufficiently accurate information of the azimuth and tilt of an airborne camera so these variables almost always must be solved for. However the camera location is often available in the imagery, for example by using exiftool on a snapshot (usually taken just before a video). Latitude-longitude information on a Phantom 3 seems to be accurate to a few meters and is a good choice. Vertical position can be less accurate (for example, it is often expressed relative to the takeoff point, rather than in the ground coordinate system) but could be used if no better GCPs are available. Finally, it is reasonable to assume for a good stabilized gimbal as on the Phantom 3 that roll is stable and perhaps taken as equal to zero within a first approximation. Thus it is possible to reduce to as low as two unknowns which can be solved with just two GCPs anywhere on the image (in fact, the solution will be found with just one GCP, but not in a least squares sense).

The choice of which of the six unknowns is taken to be known is selected by the input variable inputs.knownFlags in which each element equals 1 for a known variable and 0 if that variable is to be found by least squares solution. Thus

```
knownFlags = [1 1 0 0 0 1]
```

would mean that the x and y camera locations and the roll would be considered fixed (supplied values in the input file are used exactly as supplied) while zCam, azimuth and tilt (variables 3, 4, 5) will be solved for (supplied values are used as seeds for the nonlinear search).

While clicking on GCPs in an image is a reasonable approach to solve for the geometry of one initial image, it would be exceedingly tedious to require this for every frame of a movie where the viewing angles vary slowly due to UAV drift. Instead, we would prefer to use control points that can be automatically found in each frame, for example something that is brighter than its surroundings. We call these reference points and let the user identify a number of these in the initialization process. This process is described below.

3.1 User Inputs:
The following are the required inputs for an analysis. This example file is from a test work case and has pathnames that are appropriate for CIL conventions rather than some of the demo inputs in the toolbox.

```
% Demo input file for UAV processing.
% The user is responsible for correcting content for each new analysis

% 1.  paths, names and time stamp info:
inputs.stationStr = 'Aerielle';
inputs.dateVect = [2015 10 08 10+4 57 0];      % date/time of first frame
inputs.dt = 0.5/(24*3600);         % delta t (s) converted to datenums
inputs.frameFn = '201510081057AzMovie';          % root of frames folder
name
inputs.gcpFn = '/ftp/pub/Aerielle/2015/cx/281_Oct.08/gcp20151008.mat';
inputs.instsFn =
'/home/holman/ruby/research/UAVTesting/bathyDuck2015/makeInstsSho
rtRuns';          % instrument m-file location

% 2.  Geometry solution Inputs:
% The six extrinsic variables, the camera location and viewing angles
% in the order [ xCam yCam zCam Azimuth Tilt Roll].
% Some may be known and some unknown.  Enter 1 in knownFlags for
known
% variable.  For example, knownFlags = [1 1 0 0 0 1] means that camX and
% camY and roll are known so should not be solved for.
% Enter values for all parameters below.  If the variable is known, the
% routine will use this data.  If not, this will be the seed for the
% nonlinear search.
inputs.knownFlags = [0 0 0 0 0 1];
lat = dms2degrees([36 10 54.36]);         % data from exiftool.
long = dms2degrees([-75 44 56.60]);
[x,y] = ll2Argus('argus02b',lat,long);     % convert to local coords
inputs.xyCam = [x y];
inputs.zCam = 64;          % based on last data run
inputs.azTilt = [0 70] / 180*pi;         % first guess
inputs.roll = 0 / 180*pi;

% 3.  GCP info
% the length of gcpList and value of nRefs must be >= length(beta0)/2
inputs.gcpList = [11 12 32 28];     % use these gcps for init beta soln
inputs.nRefs = 4;                   % number of ref points for stabilization
inputs.zRefs = 7;                   % assumed z level of ref points

% 4.  Processing parameters
inputs.doImageProducts = 1;                 % usually 1.
inputs.showFoundRefPoints = 0;              % to display ref points as check
inputs.rectxy = [50 0.5 500 400 0.5 1000];     % rectification specs
inputs.rectz = 0;                           % rectification z-level

% residual calculations - NO USER INPUT HERE
inputs = makeUAVPn(inputs);           % make the path to find init-file
inputs.dn0 = datenum(inputs.dateVect);
bs = [inputs.xyCam inputs.zCam inputs.azTilt inputs.roll];  % fullvector
inputs.beta0 = bs(find(~inputs.knownFlags));
inputs.knowns = bs(find(inputs.knownFlags));
```

The inputs are in four categories.
1. Paths, names and time stamps. StationStr is the name of the particular UAV while dateVect is a 1x6 vector of [yyyy mm dd hh mm ss] where the hour is forced to GMT (in this example from Duck, NC, during summer, this is a 4 hour correction), consistent with Argus standards of recording times in GMT (an option you may not want to use). dt is the frame sampling interval (1/2 second here, expressed as a matlab datenum). FrameFn is the filename of the folder in which the individual frames are stored. Our current standard is to append the numbers 1 or 2 to this root to specify the first and second half of split MP4 movies. The user must also specify gcpFn, a gcp file, and instsFn, a file creating the pixel instruments. Examples are included in Appendices A and B.

2. Geometry solution inputs. This section defines which of the six extrinsic camera parameters [xCam yCam zCam azimuth tilt roll] are known a priori and defines those values as well as initial guesses at the unknown parameters. In this case xyCam is defined using lat-long measurments from exiftool of the snapshot. These are then converted to Argus coordinates using ll2Argus (you will need an alternate routine). For this case, only the roll is assumed to be known (set to value 0°). The other variables are just initial values for the nonlinear search. In this case I have guessed that the azimuth and tilt are 0° (looking along the +y axis) and 70° (20° below horizontal). All angles must be converted to radians (hence the /180*pi).

3. GCP Info. This section would be better handled by a gui, if someone wanted to build one. As it is, you need to look at the first frame (or an equivalent snapshot) independently and decide which GCPs you can see. The vector gcpList refers to the selected GCPs by their number in the gcp structure. In this case I am using four GCPs (so I can solve for all six geometry variables if I want). The user must also identify a number of reference points that he will establish in the initialization process. Reference points are just virtual GCPs whose effective location is found by finding their image location then converting to an equivalent xyz location assuming the vertical location is defined by the variable zRefs. In this example I use a vertical level of 7 m, roughly the height of the pier and dune. Results are insensitive to this choice (as long as it is closer to the correct ground location that it is to the zCam).

4. Processing Parameters. This section allows you to choose to do image products (timex, brightest and darkest) and whether to show how well the reference point identification is working (for possible debugging). It also defines the rectification box in x and y (rectxy = [xmin dx xmax ymin dy ymax]) and the vertical level for rectification (usually mean sea level).

The final input group requires no user input, with one exception. makeUAVPn is a routine that creates standard path and folder names for CIL processing standards. These include 1) dayFn (an Argus standard day filename, for example '274_Oct.01' where all cx data for that day will be stored), 2) pnIn, the pathname for the input png image frames (the folder in which frameFn resides), and 3) pncx, the standard CIL pathname for storing cx results. The names you wish to use will likely be different from our default names so you may need to modify these routines.

3.1 Initialization

Prior to bulk processing, several initialization steps are required. First, the input structure must be initialized, for example by calling the m-file demoInputFile.m. At present the name of this routine is hardcoded into the sampling program on line 14 but this could be deleted and the input file name called before calling the sampling program. The routine then creates the cx output directory (if not already existing), initializes the instruments, creates the stack structure and finds all of the frames.

3.2 First Frame Processing

If this is the first time processing this video (i.e. there is no pre-existing metadata file in cx), initialization is done by the routine initUAVAnalysis. The first step is to show the image and digitize the selected gcps (input.gcpList) to solve for the geometry of the first frame. Figure 1 shows the demo example with a good fit (comparing clicked green symbols to red best fit locations).

The second step is to identify a number of visual reference points that can used to correct wander in the view of subsequent frames. GCPs are often very small (e.g. the first gcp in the demo list which is a tiny white dot), so are not appropriate for automated recognition. Instead the user should identify a number of features in the image that are brighter than their surroundings (darker reference points could also be implemented later). The number of points is specified in inputs and for a 6 dof solution should be at least 4 (although I believe that 3 is possible?). For each reference point the user specifies a small box surrounding the feature by clicking a top-left then a bottom-right location (see example white square in Figure 1). A new window then shows the selected region in false color (left panel, Figure 2) so that the user can specify an intensity level that will separate the bright target from the background, in this case 210 works well. The right panel then shows the thresholded version, in this case showing only the white parts of the target. A white symbol shows the center of mass (COM) of the selected area and is used as the center location of the reference point.

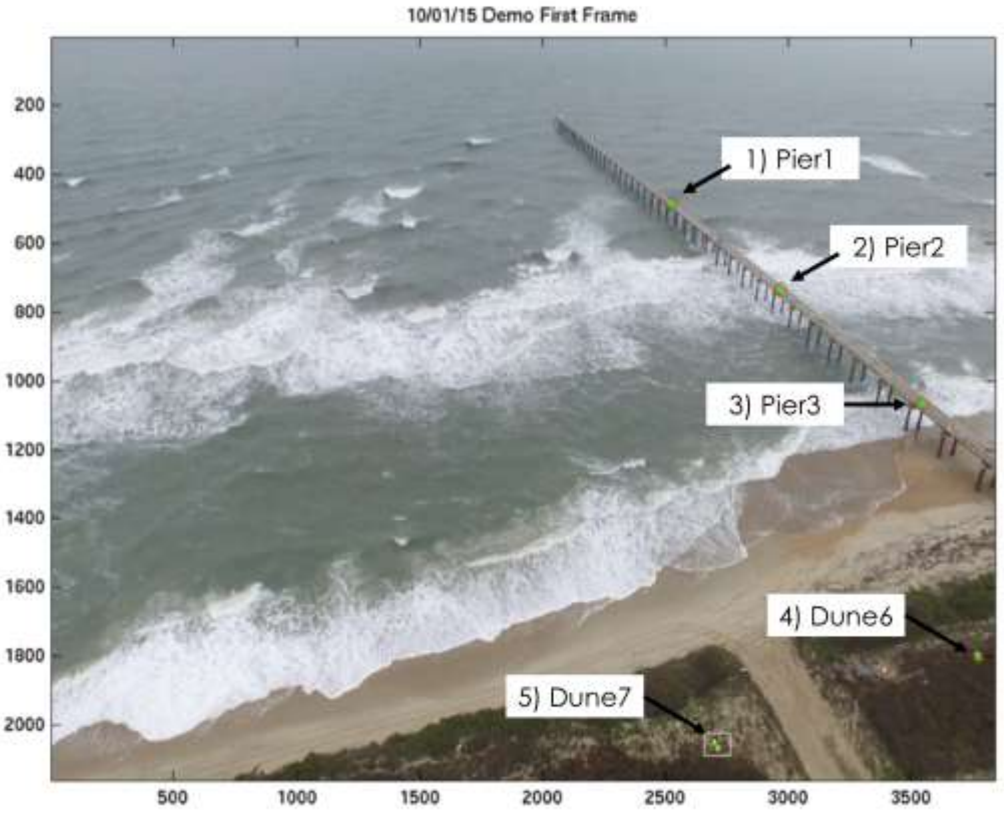

Figure 1. First frame of demo video showing the identified gcps (green asterisks show the five locations clicked by the user – gcps [1 2 3 6 7] in inputs.gcpList) and the best fit locations from the fit (red circles). The first gcp (seaward on the pier) is a tiny white point that is hard to see. The white square surrounding the bottom gcp indicates the selected sampling region for the automatic reference point search for this target (Figure 2).

NOTES – it is not important that the thresholded shape looks symmetric like this case, only that the chosen threshold isolates some feature repeatedly, for example not including occasional bits of bright surrounding background. Also, the first reference point should be chosen as the one that is best isolated from background clutter and it should have a border that is large enough to allow for expected inter-frame movement. In this case, the bottom left gcp happens to be a very identifiable target, but it could be any white feature, usually not a measured control point. The algorithm first looks for the new location (COM, or Center Of Mass) of the first reference point, then adjusts its search for the other points based on the first point displacement. Thus, other reference points can (and probably should) have much smaller bounding boxes. For example, I used the inner two gcps on the pier but had to keep the bounding boxes small to avoid including background white breakers. The white sign at x = 3400, y = 1850 would also be a good target. Reference points have virtual world locations assigned to them based on the inputs.zRefs vertical value and are subsequently treated as recognizable gcps. Note that the reference points should not lie along a single line (a problem often for beaches) unless the roll and/or camera position are declared known. The user should also be careful to not choose a reference point that might drift out of view due to aircraft drift.

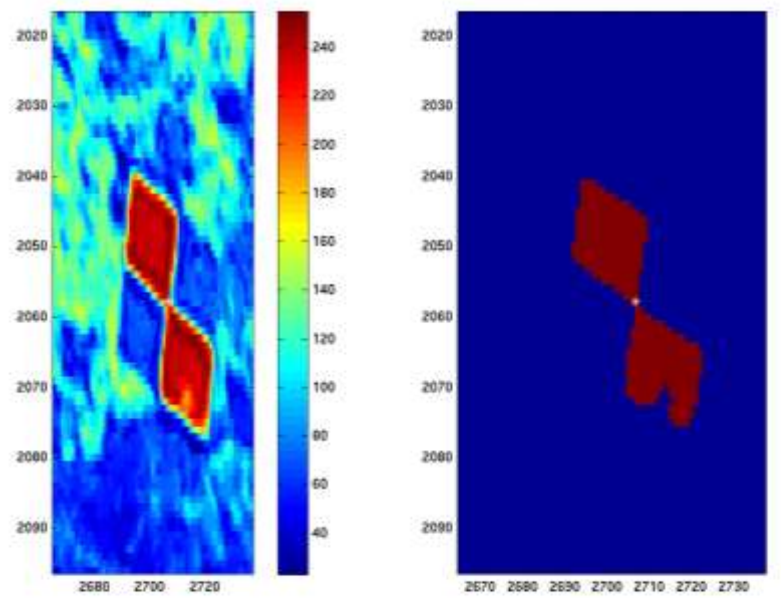

Figure 2. Left) False color blow up of the reference point selected in Figure 1 as the white box. Right) Thresholded version of the left panel. The small white symbol shows the center of mass of the selected pixels, a very accurate measure of the features location that is used for automatic co-registration of later images.

The final step of the initialization phase is to fill in the pixels of the pixels instruments and display them in a figure as a final check that the collection has come out the way that the user had hoped (Figure 3). If this is not true the user should terminate with a control-C and adjust accordingly. At this point, the initial metadata file is saved to disk.

3.3 Analysis of Remaining Frames

Analysis of the remaining frames is straight-forward. Each frame is read in and, if there is no prior geometry from a previous analysis (no pre-existing meta data file was found), a new geometry is found using routine findNewBeta. This automatically finds the reference points and solves for the new geometry, storing the result in a matrix called betas. It then samples the pixel instruments for this frame using the routine sampleDJIFrame. If image products (timex, etc) have been requested in the inputs, it then collects that data using buildRectProducts.

3.4 Closing Out the Analysis

Data analysis finishes when either all the frames have been analyzed or when the findNewBeta is no longer able to find the first reference point (usually because the operator turned the UAV to head for home but left the video running). At that point the final image products are made by routine makeFinalImages and data are organized to save in the cx directory. All important inputs are stored in a metadata file using standard Argus naming conventions. Note that since the geometries (betas) are saved, new pixel tool analysis can be done subsequently using much less CPU time, if new instruments are desired. This is done by changing the content of the instrument file then rerunning the sample program.

Time stack data are then saved in cx, again using standard Argus names including the instrument name selected by the user, for example vBar150. These can be plotted in standard ways, for example,

```
imagesc(stack.xyzAll(:,1), stack.dn, stack.data).
```

Figure 4 shows an example runup time stack, figure 5 shows a vBar time stack.

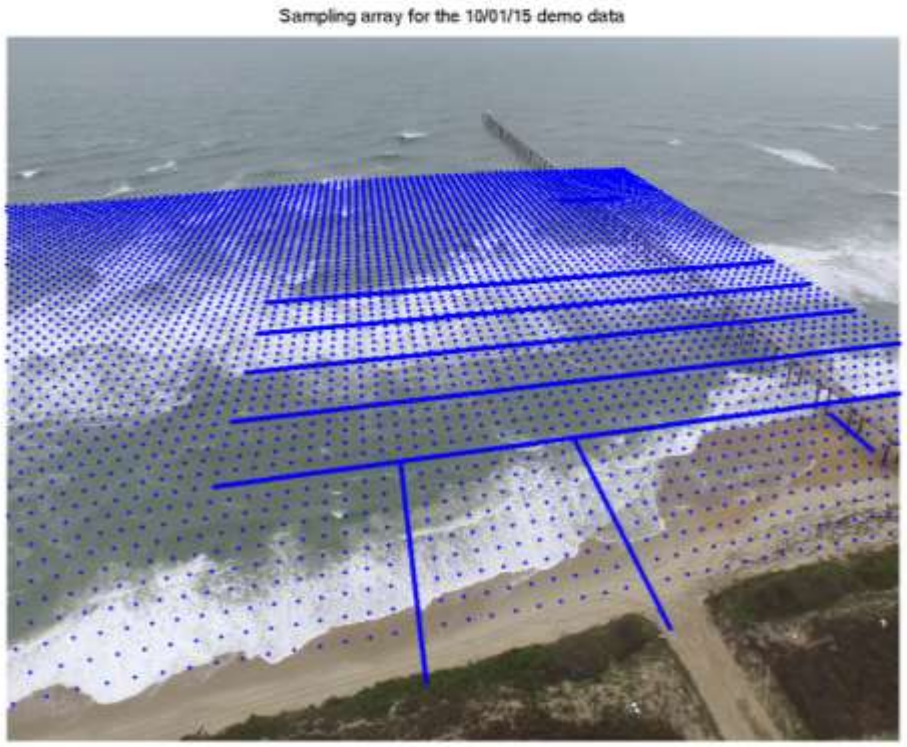

Figure 3. Sampling array for demo data. The five alongshore-oriented lines are the vBar arrays while the two cross-shore lines are for runup. The regular matrix is for cBathy. The short cross-shore line by the pier is designed to sample a pier leg so that residual vertical motion of the stabilized images can be detected (there is a similar alongshore line offshore that is less clear).

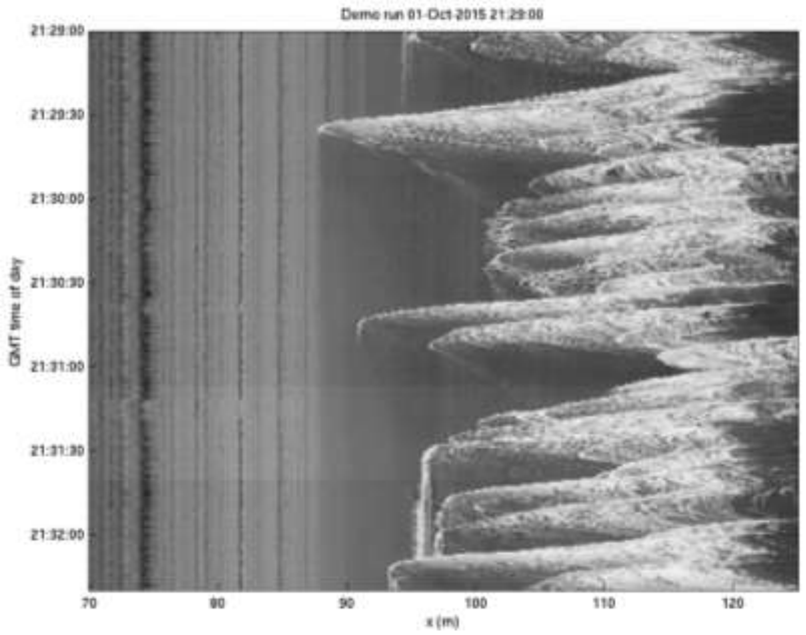

Figure 4. Example runup time stack for y = 600 for the demo run.

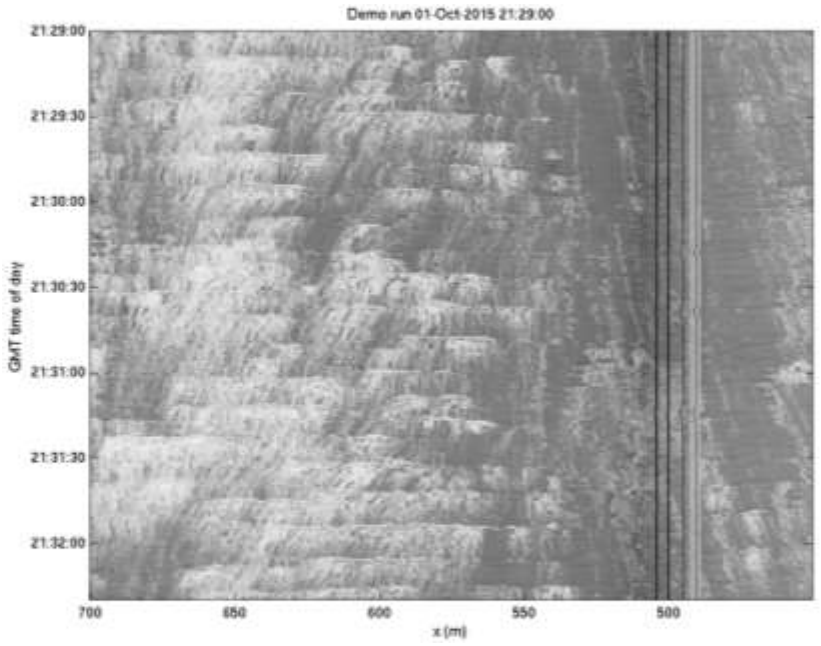

Figure 5. Example vBar stack for x = 175 for the demo run. Currents obviously flow north (down left) on the north side of the pier and to the south (down right) on the south of the pier.

Image products are saved as png's in cx using standard naming conventions and in matlab format using the routine 'printAndSaveImageProducts'. Matlab versions are saved in files with the image type 'imageProducts.mat'. These can be loaded and shown using, for example for Figure 6,

```
Imagesc(finalImages.x, finalImages.y, finalImages.timex)
```

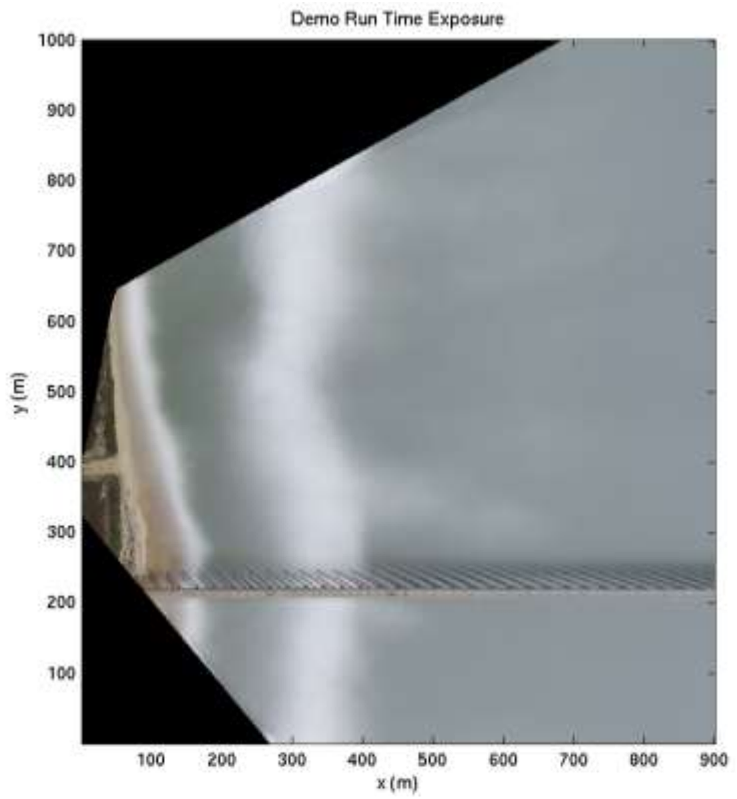

Figure 6. Rectified time exposure for the 10/01/15 demo data run.

4 Implementation Details

The toolbox is called UAVProcessing and includes a Contents.m file that describes all of the routines ('help UAVProcessingCodes'). The toolbox includes a demo data set and all routines are already set up to allow automatic processing of this 400 frame demo case (in local folder demoMovies) by typing 'sampleAerielleVideoDemo' while in the toolbox. Due to the size of the demo images (> 1GB) they are not managed using SVN but are available from http://cil-www.coas.oregonstate.edu/UAVdemoMovies.zip. This zip file should be unpacked in the toolbox directory, creating a demoMovies directory.

Some routines are listed in the help as being 'CIL routines' that are called and are located in a folder called neededCILRoutines. If you don't have CIL routines set up, you should either add this folder to your path or just copy the routines to the UAVProcessingCodes folder.

When you run the demo, you will have to click on the gcps shown in Figure 1, going from seaward to shoreward on the pier, then on the south (right), then north (left), checkerboard targets on the dune crest. All targets are checkerboards, but the seaward one is REALLY hard to see (it is just a tiny white dot). In choosing the reference points, chose a dune target as the first point with a box size similar to that shown in Figure 1. Of the four reference points, you can choose three points on the dune but at least one must not be in line with the others (on the pier is the obvious choice). Click a small surrounding box so there is no chance of getting background breakers in the target search. Figure 2 in your display will show the video frame by frame along with the found reference points (red) and geometry solution (black circles). This will allow you to determine if the tracking gets confused. If you want to see the reference point windows, set inputs.showFoundRefPoints to 1 and they will be displayed in figure window 11, 12 ….

You can change the instruments as much as you want.

If you run the analysis, results will be stored in a default output directory, determined by the contents of makeUAVPn (set by default to demoMovies/demoOutput). The default is ONLY for demonstration and should not be used for real analysis (set up a good output strategy like the one discussed above and in commented text in this routine). Note that once you have run the demo once, the default output folder will contain the results including the meta data file. Thus if a second person tries the demo, the program will find the meta data file and assume that you don't want to redo geometries. Delete files in the output directory to give a second person a fresh analysis. Look in the output directory for all of your output.

argusFilename (in neededCILRoutines) usually looks to the argus database for information. The version here has those references removed so times will be whatever you want but will say GMT. You will receive a warning occasionally that the station name was not found in the toolbox so the GMT time zone will be assumed. Ignore this.

'every' is a perl script to break out frames. It is included but likely won't work. You are responsible for breaking out your own frames.

Exiftool is a free tool that is available on the web or is commonly part of standard computer software. It provides, among other things, the GPS and vertical location of the camera for images (not for all cameras) that can be used for geometry solutions. The same info may be available from iminfo in matlab or in image library packages like Portfolio. Values must be accurate to fractions of a second in latitude and longitude to be useful.

Appendix A: example gcp file content.
This is a .mat file with a vector of gcp structure having the following fields, illustrated by the example below.

```
gcp(3) =

     num: 3
    name: 'pier3'
       x: 120.4070
       y: 516.6788
       z: 7.4230
```

gcp's are referred to by their numbers. This file must be created from a current survey, usually using an m-file specific to the purpose. Users must know where each gcp will show up (or not) in an image frame.

Appendix B: Example Instrument Design m-file

Pixel instruments are designed in a simple m-file, using xyz space. Below is an example. Instruments are currently of type line (contiguous, for example a vbar line) or matrix (an array of non-contiguous pixels, for example like cBathy). Instruments have fields

- type
- xyz
- name
- shortName
- x
- y
- z
- xyzAll

Type is 'line' or 'matrix'. Name and shortName are up to the user by we use a standard set of CIL names, for example vBar125 means an alongshore oriented vBar array located at x = 125 m. runup600 is a cross-shore line of pixels at y = 600, and mBW is a cBathy array (for historical reasons, dating from the BeachWizard days). Line instruments are specified by their two xyz end points. In this case, the x, y, and z fields are unused. A matrix is specified by those x and y fields [xmin dx xmax] and [ymin dy ymax] while the z field specifies a scalar sea level to sample at. For a matrix, the xyz field is unused. The last two instruments below are simply a cross-shore and alongshore slice through a known edge (of the pier) to test the stability of a fixed object after image stabilization.

```
function insts = makeDJIInsts201510011529
%   insts = makeDJIInsts
%
% creates pixel instruments for DJI video.  Types can be line or matrix

cnt = 1;

% vBar instruments
y = [450 700];
x = [125: 25: 225];
z = 0;
for i = 1: length(x)
   insts(cnt).type = 'line';
   insts(cnt).xyz = [x(i) y(1) z; x(i) y(2) z];
   eval(['insts(cnt).name = ''vBar' num2str(x(i)) ''';']);
   eval(['insts(cnt).shortName = ''vBar' num2str(x(i)) ''';']);
   cnt = cnt+1;
end

% some runup lines
x = [70 125];
```

```
y = [600:50:650];
z = 0;
for i = 1: length(y)
   insts(cnt).type = 'line';
   insts(cnt).xyz = [x(1) y(i) z; x(2) y(i) z];
   eval(['insts(cnt).name = ''runup' num2str(y(i)) ''';']);
   eval(['insts(cnt).shortName = ''runup' num2str(y(i)) ''';']);
   cnt = cnt+1;
end

% cBathy array
x = [80 5 400];   % determine sample region and spacing
y = [450 5 900];    % format is [min del max]
z = 0;
insts(cnt).type = 'matrix';
insts(cnt).name = 'cBathyArray';
insts(cnt).shortName = 'mBW';
insts(cnt).x = x;
insts(cnt).y = y;
insts(cnt).z = z;
cnt = cnt+1;

% make some slices to check stability
insts(cnt).type = 'line';
insts(cnt).xyz = [300 540 7; 300 500 7];
insts(cnt).name = 'x = 300 pier transect';
insts(cnt).shortName = 'x300Slice';
cnt = cnt+1;

insts(cnt).type = 'line';
insts(cnt).xyz = [100 520 3; 115 520 3];
insts(cnt).name = 'y = 520 Piling x-transect';
insts(cnt).shortName = 'y517Slice';
```

Hartley, R., and A. Zisserman (2003), *Multiple view geometry in computer vision*, second ed., 665 pp., Cambridge University Press.