Hands-on Learning: Building a Photomosaic

EXCELLABUST Tutorial

Ricard Campos (rcampos@eia.udg.edu)
Rafael Garcia (rafa@eia.udg.edu)

Vicorob - CIRS

0. Table of Contents

1. Installation

Before starting with the tutorial, we need to compile and prepare the code. First, please download and unzip the code from `coronis.udg.edu/excellabust/excellabust_tutorial__mosaicing_code.zip`.

Then, we will start by compiling the g^2o library in the "c++" folder, containing a modified version of the original library, with the homography-related factors that we developed. To do so, we have to go to the "c++" folder, and run the "install.sh" file:

```
$ chmod +x ./install.sh
$ ./install.sh
```

This script is responsible for downloading all the dependencies, in case you don't have them already, and then compile and install the modified version of g^2o in the folder.

If everything went well, the next step is to move to matlab and initialize the toolbox. So, within Matlab, we change to the "matlab" directory in the project, and run:

```
>> initialize_mosaicking_toolbox
```

1.1. Possible Issues

In addition to adding the necessary paths to the Matlab path, the "initialize_mosaicking_toolbox" script will try to run the g^2o executable. The normal output of this test should be:

```
# Using CSparse poseDim -1 landMarkDim -1 blockordering 0
No input data specified
```

If you find some problems with the g^2o library, here are some suggestions:

- **Problems finding the library**: The g^2o library is installed in `/usr/local/lib`. In case you don't have it already, you need to add this path to the LD_LIBRARY_PATH environment variable:

  ```
  $ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
  ```

  If you want, it is a good option to add this line to your ".bashrc" file:

  ```
  $ echo export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib >> ~/.bashrc
  ```

- **Problems with the versions of other libraries**: Matlab uses its own versions of some of the standard libraries. This results in some external code not working when called from Matlab, as it tries to use these old versions of the libraries instead of the (newer) system ones. If you get something similar to the following, it is likely that you have this problem:

  ```
  bool g2o::DlWrapper::openLibrary(const string&) Cannot open library: /usr/
      local/MATLAB/MATLAB_Production_Server/R2013a/sys/os/glnxa64/
      libgfortran.so.3: version `GFORTRAN_1.4' not found (required by /usr/lib/
      liblapack.so.3)
  ```

  The most common libraries giving problems for g^2o are libstdc++ and libgfortran. You need to change the matlab libraries to point to the system ones. We prepared a bash script named "relink_matlab_libraries.sh" to ease this task. This script asks the user to agree on the commands that it intends to perform. Since we will be touching libraries with superuser rights, you need

to be careful and understand what the script is doing, in order to detect possible errors in the commands it will execute. This script has been tested on our machines, but it may not work on yours, so USE AT YOUR OWN RISK. In the following, you will find a description of the behaviour of the script.

Taking libstdc++ as example, what it does first is to look for the matlab library:

```
$ find /usr/local/MATLAB/ -iname libstdc++*
```

In the system where we tested this tutorial, the returned path was:

```
/usr/local/MATLAB/MATLAB_Production_Server/R2013a/sys/os/glnxa64/libstdc++
    .so.6
```

Just for safety reasons, the script creates a backup of the library:

```
$ sudo cp /usr/local/MATLAB/MATLAB_Production_Server/R2013a/sys/os/glnxa64/
    libstdc++.so.6 /usr/local/MATLAB/MATLAB_Production_Server/R2013a/sys/os/
    glnxa64/libstdc++.so.6_bak
```

Then, it looks for the path of the system library, which in Ubuntu is usually located in a subfolder of `/usr/lib`:

```
$ find /usr/lib -iname libstdc++*
```

For instance, we found it in `/usr/lib/x86_64-linux-gnu/libstdc++.so.6`.

Now that we know both paths, we link the Matlab library to point to the system one:

```
$ sudo ln -sf /usr/lib/x86_64-linux-gnu/libstdc++.so.6 /usr/local/MATLAB/
    R2012a/bin/glnxa64/libstdc++.so.6
```

Finally, the script does the same for libgfortran.

2. Toolbox Description

The mosaic is defined as a structure, and each function in the toolbox fills or modifies parts of it. The main parts of this structure are the following:

- **Header**: General information regarding the mosaic, such as size, resolution, camera calibration, etc.
- **Nodes**: The nodes in the mosaic. A node represents an image view. That is, the image itself and the pose of the camera/vehicle. The parts of this structure are the following:
  - **Name**: The image file name.
  - **Homography**: The absolute homography, relating this image to the mosaic.
  - **Pose**: A 3D pose for the node, in the format [*UtmX*, *UtmY*, *Altitude*, *Roll*, *Pitch*, *Yaw*].
  - **FeatPos**: 2D position of the feature points detected (size num_features $\times$ 2).
  - **Desc**: SURF Descriptors [1] of the feature points (size num_features $\times$ 64).
  - **Edges**: Links between nodes, representing a substantial match between them. Each edge contains other fields:
    - *NodeInd*: Index of the node linked.
    - *Homography*: Relative homography computed from the matches found for this pair of nodes.
    - *Matches*: The matches used to compute the homography (size 4 $\times$ num_matches).

Once created, you may explore it more deeply with the variable explorer of Matlab.
Regarding the functions in the toolbox, they are divided into the following folders:

- **main**: Contains the functions needed to build the mosaic structure and render it.
- **motion**: Contains the functions related to the (robust) computation of pairwise 2D motion (i.e., homographies).
- **plotting**: Contains debug functions to pre-visualize the mosaic before the final rendering.
- **g2o**: Contains functions to convert the mosaic structure into the g^2o format to apply the global alignment optimization and read the results back.
- **_3rd_party**: 3rd party toolboxes needed by some of the functions.

The way to specify some of the parameters is with an external parameters file. The parameters file must be named "mosaicing_parameters.m" and be located in the same folder from where we are running a given function. In case of this file not existing, the default parameters in the toolbox's file "default_mosaicing_parameters.m" will be used, and a warning will raise on Matlab's command window. The "default_mosaicing_parameters.m" file is self-explanatory and contains documentation for all the available parameters.

Note that, for further reference, every function in the toolbox is documented, so you can check their meaning/use by simply typing:

```
>> help <name_of_the_function>
```

3. Step-by-step Tutorial

We begin by downloading the example datasets from `coronis.udg.edu/excellabust/excellabust_tutorial__mosaicing_examples.zip`. This file contains the "coral" dataset, which is the example we will use in this section, as well as the "g500" data, which corresponds to a dataset we collected in CIRS' water tank.

In this section we will overview the basic steps to build a mosaic for the "coral" dataset. We will use the script "script_for_mosaicing.m" in that folder, which is a modified version of the script that you can find in the toolbox. After initializing the toolbox, change Matlab's working dir to the folder of the "coral" dataset. We will now follow the different steps composing the "script_for_mosaicing.m" file.

We begin by creating the base mosaic structure, by specifying the images' folder/extension, and the camera calibration.

```
%% Parameters
InputImagesFolder = './images' ;
InputImagesExtension = 'jpg' ;
load K ;

%% First, we will create a structure containing the information for each image
Mosaic = initialize_mosaic( InputImagesFolder, InputImagesExtension, K ) ;
```

With this, we have an empty mosaic structure containing a node for each image. The next step is to extract and describe feature points for each image:

```
%% Extract features for all the images
Mosaic = feature_extraction( Mosaic ) ;
```

This will fill the FeatPos and Desc elements of each Node in the Mosaic structure. With this information, we can try to find the matching pairs in the sequence. We do this by using RANSAC [2] to compute a given homography model between candidate matching pairs (see the [Motion] part of the parameters file). Just for the sake of demonstration, we will start by visualizing the different steps of the pairwise image registration process. To do so, we need to call the "pairwise_matching" function with the demo flag set to true:

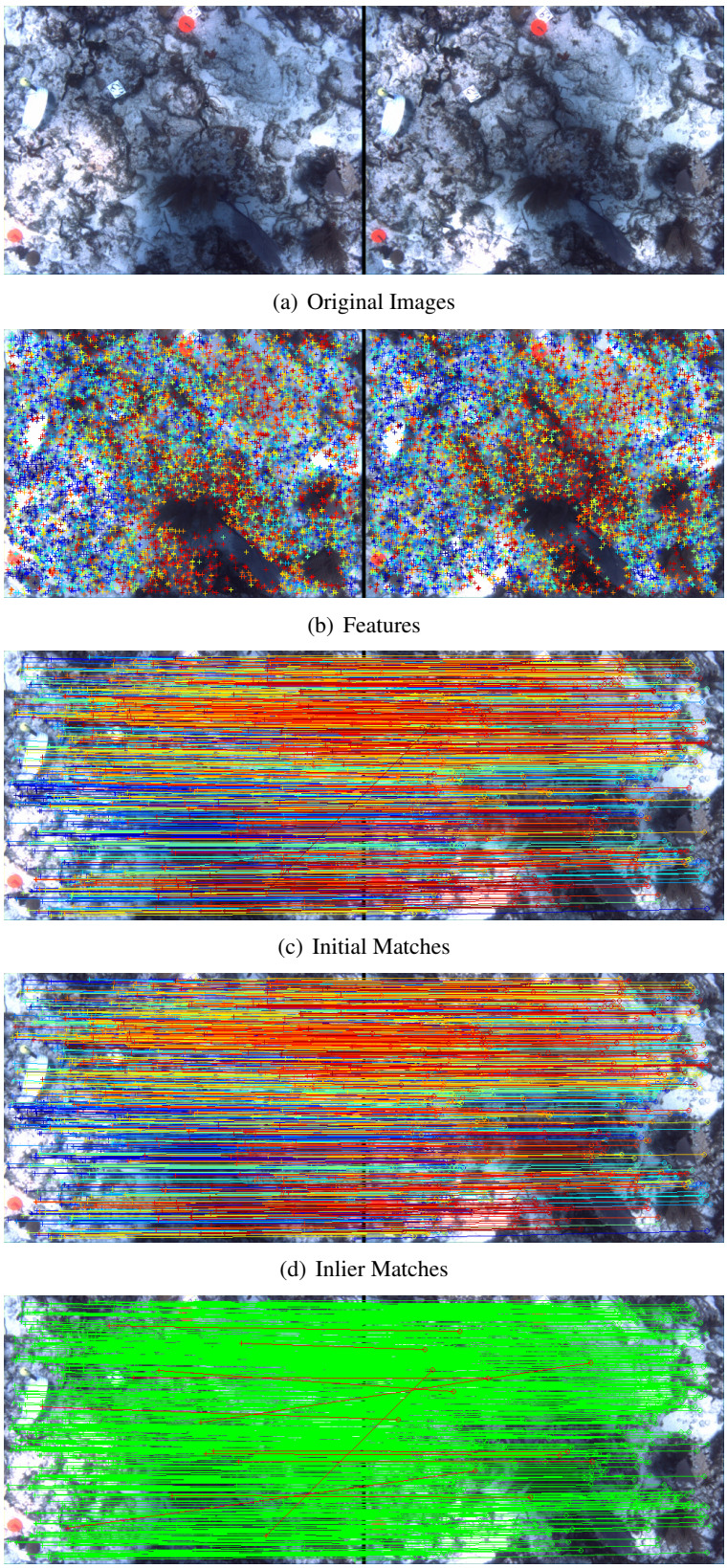
```
%% Match a pair (debug)
Mosaic = pairwise_matching( Mosaic, [3 2], true )
```

This function is the responsible of matching pairs of nodes in the Mosaic structure, where those nodes are specified, row-wise, in the second parameter, and with the pairs of indices in decreasing order (i.e., to match images 1 and 2, we need to pass the indices as [2, 1]). By setting the debug flag to true, each step will generate a series of figures with the different steps of the matching process. These figures, which should be similar to the ones you will obtain after running the function, can be seen in Figure 1.

After seeing how the matching process works, we will now try to find matching pairs in the images set. We don't have any information regarding the possible topology of the mosaic, but we know that the images are part of a sequence. Thus, we will start by computing the matches between sequential pairs. This can be done by running again the "pairwise_matching" function, this time just with the Mosaic structure as parameter:

```
%% Match the sequential pairs
Mosaic = pairwise_matching( Mosaic ) ;
```

If the "pairwise_matching" function is used in this way, a sequence is supposed on the input images, and they are tested for pairwise matching according to their order in the structure. However, as



(a) Original Images

(b) Features

(c) Initial Matches

(d) Inlier Matches

(e) Inlier(green)/Outlier(red) Matches

Figure 1: The different steps of the pairwise registration pipeline for edge [3, 2] of the coral dataset.

we have seen above, a second parameter can be passed to the function, listing the possible pairs that we want to test for matching (we will see this later on).

Since we don't have an initial estimation for the poses of the cameras, we will try to compute a base mosaic by cascading the relative homographies between pairs in the sequence. In order to reduce drift, we force the recomputation of these homographies to follow a simpler Euclidean motion model. From the computed absolute homographies, we will infer the pose of the cameras, in order to give an initial guess to the optimization to be performed later on:

```
%% Compute the absolute homographies by multiplying the relative ones
% Recompute relative homographies with an euclidean motion, since it is less sensitive to drift
Mosaic = compute_relative_from_corresp( Mosaic, 'euc' ) ;
% Cascade the relative homographies in sequence to recover the absolute ones
Mosaic = compute_absolute_from_chain( Mosaic ) ;
% Optionally, compute the pose out of the absolute homographies and known intrinsics
Mosaic = compute_pose_from_absolute( Mosaic ) ;
```

As an intermediate step, we can now pre-visualize the mosaic. Since rendering it would take a lot of resources, we check that it looks correct by just plotting the images as a wireframe (you should get something similar to what is shown in Figure 2(a)):

```
%% Preview the mosaic as a wireframe
preview_mosaic( Mosaic) ;
```

Now that we have the initial estimate provided by the cascading of sequential homographies, we will optimize the mosaic using just these constraints (i.e., the pairwise relative homographies) found in the matching step:

```
%% Optimize the mosaic
Mosaic = optimize_mosaic( Mosaic ) ;

%% Preview the optimized mosaic as a wireframe
preview_mosaic( Mosaic) ;
```

Note how the pre-visualization now shows a slightly different shape for the mosaic in Figure 2(b). This is because now the motion model is more general, and not limited to Euclidean motion. Now, we can render this first version of the mosaic. To do so, we have to give it a resolution first using the "set_mosaic_resolution" function. If the second parameter (the desired resolution) is not specified, the function will provide a guess for the proper resolution to set. Note that, if the mosaic was georeferenced (i.e., if we had imposed some navigation information), the resolution would be in meters/pixel.

```
%% Set the desired resolution for the mosaic
Mosaic = set_mosaic_resolution( Mosaic, 1 ) ;
```

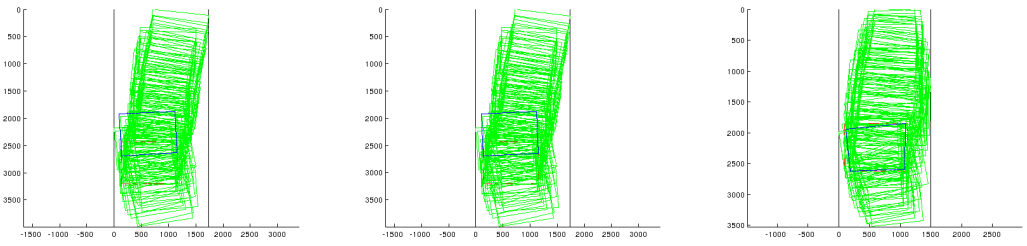
We can then render the mosaic and save it to disk:

```
%% Render the mosaic (as an image, and as a video)
ImMosaicCons = render_mosaic( Mosaic, 'last', 'Mosaic_ConsecutiveMatches.avi' ) ;
imshow( ImMosaicCons ) ;

%% Save the mosaic as an image
imwrite( ImMosaicCons, 'Mosaic_ConsecutiveMatches.png', 'png' ) ;
```

The function "render_mosaic" returns the rendered mosaic as an image, and, optionally, saves a video of the rendering process (by setting its 3rd and 4th parameters).

Now we have an initial estimation of the mosaic. However, enforcing just sequential constraints will lead to drift in the mosaic. You can better observe this drift in the generated video ("Mosaic_ConsecutiveMatches.avi"): while the sequential coherence is respected, the coherence in areas



(a) Cascade of Homographies (b) Sequential Matches, Globally Aligned (c) Non-sequential Matches, Globally Aligned

Figure 2: Differences between the initial estimation of the mosaic constructed by cascading the sequential homographies (a) the same after global alignment (b) and the final version taking into account all the sequential and non-sequential matching constraints (c).

separated in time is not enforced. Thus, we need to add some non-consecutive constraints in the global alignment. In this case, we can take advantage of the initial mosaic estimation we have so far to guess new possible matching pairs. We will look for potential pairs based on the absolute homographies:

```
%% Find non-consecutive matches
NonConsPairs = get_possible_pairs_from_absolute( Mosaic, 'overlap', 0.01 ) ;
Mosaic = pairwise_matching( Mosaic, NonConsPairs ) ;
```

The function “get_possible_pairs_from_absolute”, when called with the ’overlap’ flag, will try to find all pairs of images whose projection on the mosaic overlap. With the new set of matches, we optimize all the constraints (see expected result in Figure 2(c)):

```
%% Optimize again with the new matches
Mosaic = optimize_mosaic( Mosaic ) ;

%% Preview the mosaic as a wireframe
preview_mosaic( Mosaic ) ;
```

Finally, we render the mosaic again:

```
%% Set the desired resolution for the mosaic
Mosaic = set_mosaic_resolution( Mosaic, 1 ) ;

%% Render the final mosaic
ImMosaic = render_mosaic( Mosaic, 'last', 'Mosaic.avi' ) ;
imshow( ImMosaic ) ;

%% Save the mosaic as an image
imwrite( ImMosaic, 'Mosaic.png', 'png' ) ;
```

The result you get should be similar to the one presented in Figure 3.

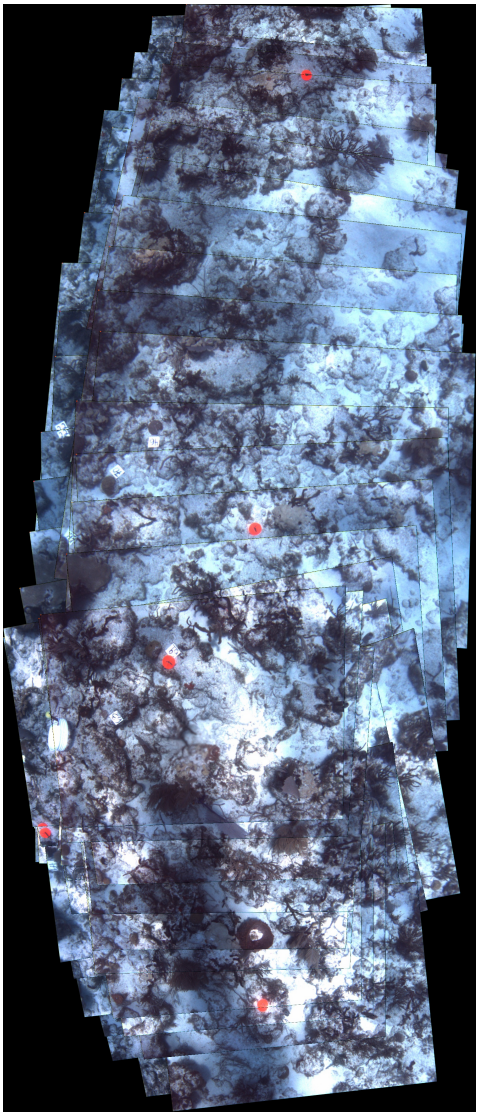


Figure 3: The resulting mosaic for the “coral” dataset.

4. Exercise

Based on the script reviewed in the previous section, build a mosaic with the images collected on the pool with Girona 500. However, in this case you have available navigation data that you can use. Modify the previously presented "script_for_mosaicing.m" to account for this new information, and use it to create an initial estimation of the mosaic instead of the cascading of sequential homographies.

HINTS: check the functions: "get_pose_from_nav" and "compute_absolute_from_pose", and the 'distance' mode of "get_possible_pairs_from_absolute".

5. References

[1] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-up robust features (surf)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346 – 359, 2008, similarity Matching in Computer Vision and Multimedia. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314207001555

[2] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981. [Online]. Available: http://doi.acm.org/10.1145/358669.358692