Aether Lab Synthetic Image Generation Code Manual

Git Commit: e4d66c1c5d

Author

Matthew Giarra
Matthew.giarra@gmail.com

Last modified

2014-12-31

System Requirements

1. Mac, PC, or Linux machine running Matlab 2013a or later
2. Matlab Coder toolbox is required for generating and running compiled Mex functions

Overview

`piv-image-generation` is a library of Matlab codes used to generate synthetic photographs for testing planar particle image velocimetry codes (PIV). Particle positions are generated pseudo-randomly, and then advected according to known displacement fields (for full-field images) or coordinate transformations (for Monte–Carlo analysis). The code also saves additional Mat files that contain the ground-truth solutions describing the particle motions, vorticity, etc. These solution files are the basis for assessing the performance of PIV codes. Particle images are rendered using the algorithm of Olsen and Adrian [1], and the specific details of the implementation can be found in Brady *et al.* [2].

The general workflow used to operate these codes is as follows.

1. The user-defined parameters for generating images are specified in text-base "job files." Job files are Matlab functions that return a structure or list of structures whose fields specify the parameters relevant to image generation. These parameters include the paths at which to save images, the number of images to be generated, the sizes of the images to be generated, etc.

2. These structures are passed as arguments to functions that generate and save the synthetic images.

3. Particle displacements in the synthetic images are measured using some external PIV program.

4. The PIV-estimated displacements, vorticity, etc., are compared against the ground-truth solution files that were generated concurrently with the synthetic PIV images.

Installation

The PIV Image Generation code repository is maintained on Github. If Git is installed on your system, clone the repository using:

```
$ git clone https://github.com/matthewgiarra/piv-image-generation
```

Otherwise, to download the code in your browser, navigate to

https://github.com/matthewgiarra/piv-image-generation

Once here, click on "Download ZIP" or "Clone in Desktop."

Note that because this is a Github project, updates will only be accepted through Git. Git clients with graphical interfaces can be found at https://github.com. A great 15-minutes interactive tutorial to Git can be found at:

https://try.github.io/levels/1/challenges/1

Keeping Up to Date

Installing and using Git is the easiest way to make sure your copy of piv-image-generation is up to date. To pull recent changes that have been committed to the Github repository, navigate to your local copy of piv-image-generation and issue the command `git pull`:

```
$ cd /path/to/piv-image-generation/
$ git pull
```

Alternatively, use your favorite Git client with a graphical interface to pull changes from this repository.

Contributing Changes

One great feature of Github repositories is the ability of users besides the code's author to submit changes to the code. Users can contribute changes through a "pull request." Instructions for creating pull requests can be found on Github's website at https://help.github.com/articles/creating-a-pull-request/

Using piv-image-generation

Compiling codes

Compiling several of the functions from this library into Matlab Mex functions can increase their computational speed by roughly one hundred times. This section identifies those functions and provides instructions for compiling them into Mex functions.

The two functions that can be compiled into Mex functions for increased performance are `generateImagePair_mc.m` and `generateParticleImage.m`. In order to realize this performance increase, the codes must be compiled on the same machine (or on a different machine with similar hardware and operating system) on which they will be run. In other words, Mex functions compiled using Linux will not run on Windows, and Mex functions compiled on a machine with an AMD processor may not run on a machine with an Intel processor, etc.

These codes handle the rendering of particle images, and contain several nested loops that execute faster when compiled than they do when run directly as Matlab functions. The code `generateImagePair_mc.m` is called when generating Monte Carlo image sets, and the code `generateParticleImage.m` is called when generating "full-field" images.

These instructions should not be platform specific, and should work on any installation of Matlab with Matlab Coder installed.

Compiling code for Monte-Carlo image generation

The function `generateImagePair_mc.m` renders particles for Monte Carlo image generation. To compile this code, navigate to the piv-image-generation directory and call the code `compile_generateImagePair_mc.m`:

```
>> cd /path/to/piv-image-generation;
>> compile_generateImagePair_mc;
```

Compiling code full-field image generation

The function `generateParticleImage.m` renders particles for full-field image generation. To compile this code, navigate to the piv-image-generation directory and call the code `compile_generateParticleImage.m`:

```
>> cd /path/to/piv-image-generation;
>> compile_generateParticleImage;
```

Generating images

Lamb Vortex Full-Field Images

This section provides a guided example for generating synthetic PIV images of a Lamb vortex ring.

1. Navigate to the piv-image-generation directory.

   ```
   >> cd /path/to/piv-image-generation/directory
   ```

2. Add the jobfiles directory to the Matlab path.

   ```
   >> addpath jobfiles;
   ```

3. Open the PIV image generation job file.

   ```
   >> open lambOseenVortexImageGenerationJobFile;
   ```

4. Modify the jobfile according to your preferences. For this example, all that should need to be changed is the path to the location at which to save the images.

5. Run the jobfile and return its output to a structure variable called JobFile.

   ```
   >> JobFile = lambOseenVortexImageGenerationJobFile;
   ```

6. Pass the structure JobFile to the function generateImages_LambVortex

   ```
   >> generateImages_LambVortex(JobFile);
   ```

7. Inspect the images to make sure they were output properly.

Monte–Carlo Images

This section is forthcoming.

Reading ground-truth solutions

Lamb Vortex Full-Field Images

The ground-truth Eulerian displacement can be calculated at the center of each PIV interrogation region (IR) by running the Lamb vortex velocity function and inputting the coordinates of the IR centers as the initial computational particle positions. The following code serves as a template example for extracting the ground-truth Eulerian displacement and vorticity fields at the centers of PIV interrogation regions.

```
% Create a List of the image numbers used in each correlation pair.
firstImageNumbers = startImage : frameStep : endImage;
secondImageNumbers = firstImageNumbers + correlationStep;

% Specify the path to the image generation parameters file
% that was saved concurrently with the synthetic images
parametersPath = '/path/to/image/generation/parameters/file.mat';

% Load the image generation parameters file.
imageGenerationParameters = load(parametersPath);

% Extract the vortex parameters
% from the image generation parameters file.
vortexParameters = imageGenerationParameters.VortexParameters;

% Extract the image times from the parameters file.
% Image times in sort-of physical units (i.e. "seconds")
imageTimes = imageGenerationParameters.T;

% Determine the solution times corresponding to the frame numbers
firstImageTimes  = imageTimes(firstImageNumbers);
secondImageTimes = imageTimes(secondImageNumbers);

% Simulated time elapsed between the two images
interFrameTime = secondImageTimes(1) - firstImageTimes(1);

% Set the solution time to halfway between the image times.
% This results in a second-order accurate estimate
% of the Eulerian displacement.
solutionTimes = firstImageTimes + (secondImageTimes - firstImageTimes) / 2;

% This is the number of solution times calculated.
number_of_solution_times = length(solutionTimes);

% These are column vectors corresponding to the
% row (Y) and column (X) coordinates
% of the centers of the PIV interrogation regions
% The variables X and Y are expected to come from
% the external PIV software's solution file.
gridPointsX = X(:);
gridPointsY = Y(:);

% regionWidth and regionHeight are the width and height
% of each interrogation region in pixels. In this example,
% the IRs will be assumed to be 64x64.
regionWidth  = 64;
regionHeight = 64;
```

```
% These are the coordinates of the geometric centroids of each PIV window.
% This corresponds to the physical location in the flow that the PIV
% correlation is supposed to measure. For odd sized windows, the geometric
% centroid of the window is at the center pixel. For even-sized windows, it
% is 0.5 pixels to the right of the pixel located at (regionHeight/2) or
% (regionWidth/2).
xCenter_01 = gridPointsX + 0.5 * (1 - mod(regionWidth,  2));
yCenter_01 = gridPointsY + 0.5 * (1 - mod(regionHeight, 2));

% Vector containing all the x and y grid points.
% This is an input to the vortex velocity function,
% and this format (a single column) is required by ODE45.
gridPointsVector = cat(1, gridPointsX, gridPointsY);

% These lines allocate matrices for the analytical
% velocity fields calculated at each solution time.
%
% Horizontal velocity component
uTrue = zeros([size(X), number_of_solution_times]);

% Vertical velocity component
vTrue = zeros([size(X), number_of_solution_times]);

% Out of plane component of vorticity
rTrue = zeros([size(X), number_of_solution_times]);

% This loops over the different solution times
% and calculates the analytical displacement at each time.
for t = 1 : number_of_solution_times

    % This calculates the true Eulerian velocity field for the t'th field.
    [trueVelocities, trueVorticity] = ...
    lambOseenVortexRingVelocityFunction(solutionTimes(t), ...
    [xCenter_01; yCenter_01], vortexParameters);

    % This is the ground-truth horizontal component
    % of the velocity field expressed as a vector.
    uTrueVect = interFrameTime * trueVelocities(1 : length(trueVelocities) / 2);

    % This is the ground-truth vertical component
    % of the velocity field expressed as a vector.
    vTrueVect = interFrameTime * trueVelocities(length(trueVelocities)/2 + 1 : end);

    % This is the ground-truth out-of-plane component
    % of the vorticity field expressed as a vector.
    rTrueVect = interFrameTime * trueVorticity(:);
```

```
    % This reshapes the horizontal component
    % of the ground-truth velocity field into
    % a matrix of the same size as the
    % input coordinates.
    uTrue(:, :, t) = flipud(reshape(uTrueVect, size(X)));

    % This reshapes the vertical component
    % of the ground-truth velocity field into
    % a matrix of the same size as the
    % input coordinates.
    vTrue(:, :, t) = -1 * flipud(reshape(vTrueVect, size(X)));

    % This reshapes the out-of-plane component
    % of the ground-truth vorticity field into
    % a matrix of the same size as the
    % input coordinates.
    rTrue(:, :, t) = flipud(reshape(rTrueVect, size(X)));

end
```

Monte–Carlo Images

This section is forthcoming.

References

1. Olsen, M.G. and R.J. Adrian, *Out-of-focus effects on particle image visibility and correlation in microscopic particle image velocimetry.* Experiments in Fluids, 2000. **29**(1): p. S166-S174.
2. Brady, M.R., S.G. Raben, and P.P. Vlachos, *Methods for Digital Particle Image Sizing (DPIS): Comparisons and improvements.* Flow Measurement and Instrumentation, 2009. **20**(6): p. 207-219.