

Light Field Toolbox for Matlab

v0.4 released 12-Feb-2015

Copyright (c) 2013–2015 by Donald G. Dansereau

This is a set of tools for working with light field (aka plenoptic) imagery in Matlab. Features include decoding, calibration, rectification, colour correction, basic filtering and visualization of light field images. New in version 0.4 are some linear depth/focus and denoising filters.

Download the sample light field pack at http://www-personal.acfr.usyd.edu.au/ddan1654/LFToolbox0.3_Samples1.zip. Sample calibration datasets can be found at <http://marine.acfr.usyd.edu.au/plenoptic-imaging>.

Compatibility

LFToolbox 0.4 is reverse-compatible with LFToolbox 0.3. The original Lytro camera (the “F01”) and the Lytro Illum are supported under Lytro Desktop 4 and 3. Calibration and rectification of Illum imagery is experimental.

Users upgrading directly from LFToolbox 0.2 will need to re-run calibration and decoding and update some parameter names.

Contributing / Feedback

Suggestions, bug reports, code improvements and new functionality are welcome – email Donald.Dansereau+LFToolbox@gmail.com.

Acknowledgments

Parts of the code were taken with permission from the Camera Calibration Toolbox for Matlab by Jean-Yves Bouguet, with contributions from Pietro Perona and others; and from the JSONlab Toolbox by Qianqian Fang and others. LFFigure was originally by Daniel Eaton. The LFP reader is based in part on Nirav Patel and Doug Kelley’s LFP readers. Thanks to Michael Tao for help and samples for decoding Illum imagery.

Citing

The appropriate citations for decoding, calibration and rectification and the volumetric focus (hyperfan) filter are (see the README file for bibtex):

- [1] D. G. Dansereau, O. Pizarro, and S. B. Williams, “Decoding, calibration and rectification for lenselet-based plenoptic cameras,” in Computer Vision and Pattern Recognition (CVPR), IEEE Conference on. IEEE, Jun 2013.
- [2] D. G. Dansereau, O. Pizarro, and S. B. Williams, “Linear Volumetric Focus for Light Field Cameras,” in ACM Transactions on Graphics (TOG), vol. 34, no. 2, 2015.

Contents

1	Changes and Future Plans	3
2	Setting up the Toolbox	3
3	A Quick Tour	3
3.1	Decoding the Sample Light Fields	3
3.2	Loading Gantry-style Light Fields	6
3.3	Basic Filters	7
3.4	Running the Small Sample Calibration	9
3.4.1	Calibrating	9
3.4.2	Validating	10
3.4.3	Cleaning Up and Validating	11
3.5	Beyond the Samples: Working with Your Own Light Fields	12
4	Decoding in Detail	13
4.1	Overview	13
4.2	Analyzing White Images	13
4.3	Decoding a Lenslet Image	14
4.4	Structure of the Decoded Light Field	14
5	Calibration in Detail	15
5.1	Calibration Results	16
5.2	Rectification Results	16
5.3	Controlling Rectification	17
	Appendices	17
	Appendix A Working with Lytro Files	20
A.1	Extracting White Images	20
A.2	Locating Picture Files	20
A.2.1	LFP, LFR, lfp or lfr?	21
A.2.2	Thumbnails	21
	Appendix B Function Reference	22

1 Changes and Future Plans

Please see the README file for details.

2 Setting up the Toolbox

After unzipping the toolbox files to an appropriate location, run the convenience function `LFMatlabPathSetup` to set up the Matlab path. This must be re-run every time Matlab restarts, so consider adding a line to `startup.m`, e.g.

```
run('~/MyMatlabCode/LFToolbox0.2/LFMatlabPathSetup.m')
```

3 A Quick Tour

3.1 Decoding the Sample Light Fields

1. **Download** the LF Toolbox and decompress into an appropriate location. Run `LFMatlabPathSetup` to set up Matlab's paths.
2. **Download** the sample light field pack at http://www-personal.acfr.usyd.edu.au/ddan1654/LFToolbox0.3_Samples1.zip and decompress into its own folder. The samples folder structure was chosen for easy addition of your own cameras and calibrations:

<code>Samples</code>	Top level of samples
<code>Images</code>	Sample light field images
<code>F01</code>	F01 images
<code>Illum</code>	Illum images
<code>Cameras</code>	Stores info for one or more cameras
<code>A000424242</code>	Camera used to measure F01 samples
<code>CalZoomedOutFixedFoc</code>	A single calibration result
<code>WhiteImages</code>	White images for the F01 camera
<code>B5143300780</code>	Camera used to measure Illum samples
<code>WhiteImages</code>	White images for the Illum camera

From within Matlab `cd` into the top level of the samples folder. If you are in the correct location, the Matlab command `ls` should list the top-level folders:

```
Images
Cameras
```

3. **Run `LFUtilProcessWhiteImages`** to build a white image database. This searches the `Cameras` folder for white images, generating a lenslet grid model for each – the grid models are saved as `*.grid.json`. The database of white images is saved as `Cameras/WhiteFileDatabase.mat`, and is used in selecting the appropriate white image for decoding each light field.

As of version 0.3 of the toolbox, samples ship with precomputed `.grid.json` files. These files may be removed in order to force their re-generation. When doing so, for each lenslet grid model figures similar to Fig. 1 are presented for visual confirmation that the grid model is a good fit. Each figure shows a small subset of the whole frame to allow close inspection of the lenslets. Five such

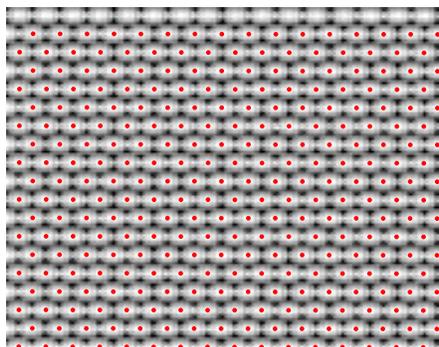


Figure 1: Example of a white image showing estimated lenslet centers as red dots.

images are shown, one for each image corner and one for the central portion of the image. Each red dot should appear near the center of a lenslet – i.e. near the brightest spot on each white bump, as depicted in Fig. 1. It does not matter if there are a few rows of lenslets on the edges with no red markers.

4. **Run `LFUtilDecodeLytroFolder`** to decode the sample light fields. The script searches the `Images` folder and its sub-folders for light fields and decodes each. By default it searches for all compatible Lytro light field formats, including `lfp` and `raw`.

The decoding process selects the appropriate white image for each light field and saves the decoded 4D light fields, `*_Decoded.mat`, and thumbnail, `*.png`, alongside the input images. A thumbnail of each light field is also displayed as it is decoded. Thumbnails are histogram-adjusted, but the saved light field is not. Example thumbnails are shown in Fig. 2.

5. (optional) **Re-run `LFUtilDecodeLytroFolder`** to perform colour correction. Use the commands

```
DecodeOptions.OptionalTasks = 'ColourCorrect';
LFUtilDecodeLytroFolder([], [], DecodeOptions);
```

The `DecodeOptions` argument requests the optional task colour correction be performed. The first and second arguments are omitted by passing empty arrays `[]`.

Colour correction applies the information found in the light field metadata, including basic RGB colour and Gamma correction. The script keeps track of which operations have been applied to each light field, and so it will not repeat the decoding process, but will instead load each already-decoded light field, operate on it, and *overwrite* it with the colour-corrected light field. Similarly, subsequent requests will not repeat the already-completed colour correction operation.

Decoding and colour-correction can be performed in one step by including the `ColourCorrect` task in the first call to `LFUtilDecodeLytroFolder`.

As of version 0.3 of the toolbox, histogram adjustment is no longer automatically applied – you may apply histogram equalization using `LFHistEqualize`.



Figure 2: Decoded (top) and colour-corrected output (bottom) – the white speckles in the bird image are due to a pane of grubby glass between the camera and the bird. Running `LFDispVidCirc` or `LFDispMousePan` creates a shifting-perspective view in which this is more clear.

Illum imagery is not gamma-corrected. Example colour-corrected output is shown in the bottom row of Fig. 2, and in Fig. 3.

- (optional) Use `LFDispVidCirc` or `LFDispMousePan` to visualize the light field with a shifting perspective. First load a light field using a command of the form

```
load('Images/F01/IMG_0001__Decoded.mat');
```

to load the light field variable `LF`, then run either `LFDispVidCirc(LF)` or `LFDispMousePan(LF)`. The former automatically animates a circular motion, while the latter allows mouse-controlled motion: click and drag in the window to change the perspective. Try a larger display with `LFDispMousePan(LF, 2)` or `LFDispVidCirc(LF, [], [], 2)`, which doubles the displayed size. Close any open display windows before changing display sizes.

- (optional) Run `LFUtilProcessCalibrations` then re-run `LFUtilDecodeLytroFolder` to perform rectification. To rectify a specific light field, use the commands

```
DecodeOptions.OptionalTasks = 'Rectify';
LFUtilDecodeLytroFolder( ...
    'Images/F01/IMG_0002__frame.raw', [], DecodeOptions);
```

The `LFUtilProcessCalibrations` script builds a calibration database which it saves in `Cameras/CalibrationDatabase.mat`. This file allows selection of the



Figure 3: Decoded and colour-corrected Illum images, manually Gamma-corrected by raising to the power 0.7.

calibration appropriate for each light field. Only one calibration is provided in the Sample Pack, and it is appropriate only for the F01 samples 2 and 5.

As in the colour-correction example, we pass an `OptionalTasks` argument, this time requesting rectification. The rectified light field *overwrites* the decoded light field file, and the decoding script will not repeat already-completed rectifications. The result of rectifying Sample 2 is shown in Fig. 4.

3.2 Loading Gantry-style Light Fields

New in version 0.3 of the toolbox is `LFReadGantryArray`, which will read gantry-style light fields such as those available at the Stanford Light Field Archive at <http://lightfield.stanford.edu>. Gantry-style light fields are generally collected by a single camera mounted on a robotic gantry, though any light field stored as an ordered collection of individual images can be read using this function.

1. **Download** an image archive from the Stanford light field archive <http://lightfield.stanford.edu>, e.g. the `LegoKnights` light field. Download the “Rectified and cropped” version.
2. **Unzip** the archive into a dedicated folder for Stanford samples, following a structure like this:

```
Stanford      Top of Stanford light fields
  JellyBeans
    rectified  Rectified and cropped image files
  LegoKnights
    rectified  Rectified and cropped image files
```

3. **Run `LFReadGantryArray`** from the top-level of the Stanford samples:

```
LF = LFReadGantryArray('LegoKnights/rectified', struct('UVLimit', 256));
```

This loads the `LegoKnights` light field into a $17 \times 17 \times 256 \times 256 \times 3$ array.

4. **Run `LFDispMousePan`** to display the loaded light field.

It’s possible to read the Stanford light fields at full resolution, e.g. the following yields a $17 \times 17 \times 1024 \times 1024 \times 3$ array, occupying 909 MBytes of RAM:

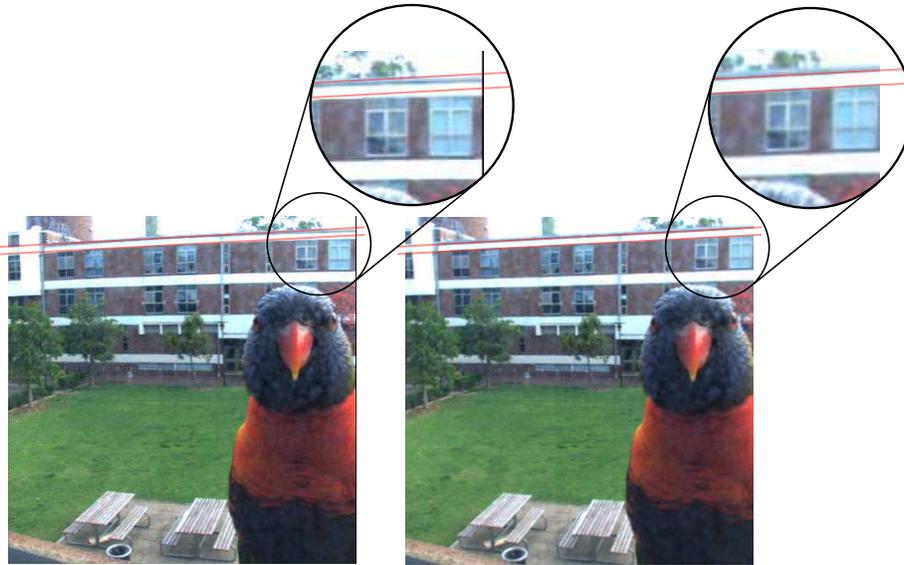


Figure 4: F01 Sample 2 before and after rectification, with insets showing the reversal of lens distortion.

```
LF = LFReadGantryArray('LegoKnights/rectified');
```

A few of the Stanford light fields follow a lawnmower pattern. These can be read and adjusted as follows:

```
LF = LFReadGantryArray('humvee-tree', struct('STSize', [16,16]));
LF(1:2:end, :, :, :) = LF(1:2:end, end:-1:1, :, :, :);
```

3.3 Basic Filters

New in LFToolbox 0.4 are some basic filters, including a shift-and-sum filter for planar focus, and a set of linear 2D and 4D filters for planar and volumetric focus.

Run one of `LFDemoBasicFiltLytroF01` or `LFDemoBasicFiltIllum` for a demo of some of the filters operating on Lytro imagery. This should be done from the top level of the Samples folder, after decoding the light fields as described in Sect. 3.1. The best performance is obtained with rectified light fields.

Run `LFDemoBasicFiltGantry` for a demo filtering the Stanford light fields. This should be done from the top of the Stanford light fields folder, after downloading and unzipping the samples following the instructions in Sect. 3.2.

Uncomment the appropriate line near the top of `LFDemoBasicFiltGantry` to select from the 12 input light fields.

Examples of filtering output are shown in Figs. 5, 6, and 7.



Figure 5: Three examples of filtering Lytro imagery: the shift-and-sum filter performing planar focus on the foreground window (left) and on the Lorikeet (center), and a hyperfan filter performing volumetric focus to pass the Lorikeet and background building while rejecting the foreground window (right, compare with Fig. 3).



Figure 6: Three examples of filtering gantry imagery: the shift-and-sum filter performing planar focus (left), the hyperfan filter performing volumetric focus (center), and the max between two hyperfan filters, focusing simultaneously on two planes (right).



Figure 7: Example of filtering Lytro Illum imagery, showing the input (left) and shift-and-sum filter performing planar focus (right).

3.4 Running the Small Sample Calibration

3.4.1 Calibrating

The example below assumes you've completed the Decoding tour above, including generating the white image database. The small calibration dataset employed here is intended only to quickly demonstrate operation of the toolbox, and has several shortcomings in terms of effectively calibrating the camera:

- The checkerboard is too large for sufficiently short-range poses – a lenslet-based camera has a small spatial baseline, and calibrating this baseline benefits from close-up poses
- The checkerboard is not very dense – more corners would be appropriate
- There is insufficient diversity in the checkerboard poses – ten or more diverse images would be appropriate

More realistic (and larger) datasets are available at <http://marine.acfr.usyd.edu.au/plenoptic-imaging>. Good results have been obtained using a 19×19 grid with a 3.6 mm spacing, with at least ten diverse poses.

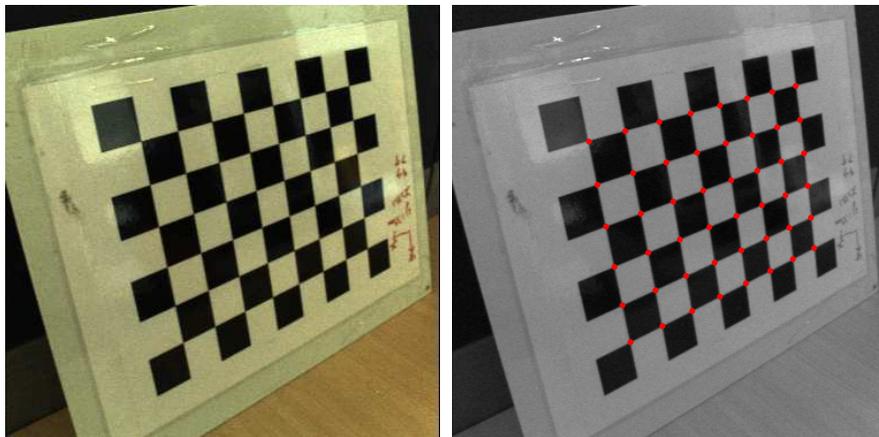


Figure 8: Left: Example of a decoded checkerboard image – no colour correction is necessary and rectification should not be applied; Right: Example of checkerboard corners automatically fit to the checkerboard in the first step of a calibration.

1. **Download** the small sample calibration from <http://www-personal.acfr.usyd.edu.au/ddan1654/PlenCalSmallExample.zip>. **Decompress to your Samples/Cameras/A000424242/ folder:**

Samples	Top level of samples
Cameras	Stores info for one or more cameras
A000424242	The camera used to measure the samples
CalZoomedOutFixedFoc	A single calibration result
PlenCalSmallExample	The newly-added calibration
WhiteImages	White images for the sample camera
Images	Sample light field images

2. **Run LFUtilDecodeLytroFolder** to decode the calibration light fields. From within Matlab `cd` into the top level of the samples folder, then use the command

```
LFUtilDecodeLytroFolder( ...
    'Cameras/A000424242/PlenCalSmallExample/');
```

This should find and decode the calibration checkerboard images. Note that colour-correction is omitted as it is not required, and rectification would invalidate the results. A thumbnail of one of the decoded checkerboard images is shown in Fig. 8.

3. **Run LFUtilCalLensletCam** to run the calibration. This function automatically progresses through all the stages of calibration. Use the commands

```
CalOptions.ExpectedCheckerSize = [8,6];
CalOptions.ExpectedCheckerSpacing_m = 1e-3*[35.1, 35.0];
LFUtilCalLensletCam( ...
    'Cameras/A000424242/PlenCalSmallExample', CalOptions);
```

These options tell the calibration function that the checkerboard spacing is 35.1×35.0 mm, and that there are 8×6 corners. Note that edge corners are not included in this count, so a standard 8×8 square chess board yields 7×7 corners. These values are available in the README file that came with the calibration sample. Calibration automatically proceeds through corner identification, parameter initialization, parameter optimization without lens distortion, then with lens distortion, and a final stage of parameter refinement. These are described in more detail in Sect. 5.

During the parameter initialization step, a pose estimate display is drawn resembling that shown in Fig. 9. This display is updated throughout the remaining stages, reflecting the refinement of the pose and camera model estimates. Reprojection errors are also shown in the textual output. Typical final root mean squared error (RMSE) values for the small calibration example are in the vicinity of 0.2 mm.

The ultimate product of the calibration process is the calibration information file, `CalInfo.json`, which contains pose, intrinsic and distortion parameters, as well as the lenslet grid model used to decode the checkerboard light fields.

3.4.2 Validating

One way to validate a calibration is to rectify the checkerboard images. The process closely resembles the rectification step described in the Decoding tour:

1. **Run LFUtilProcessCalibrations** to add the newly-completed calibration to the calibration database. Note from the output of that function that the small calibration example is very close to the sample calibration provided with the sample pack, differing only by a few focus steps.

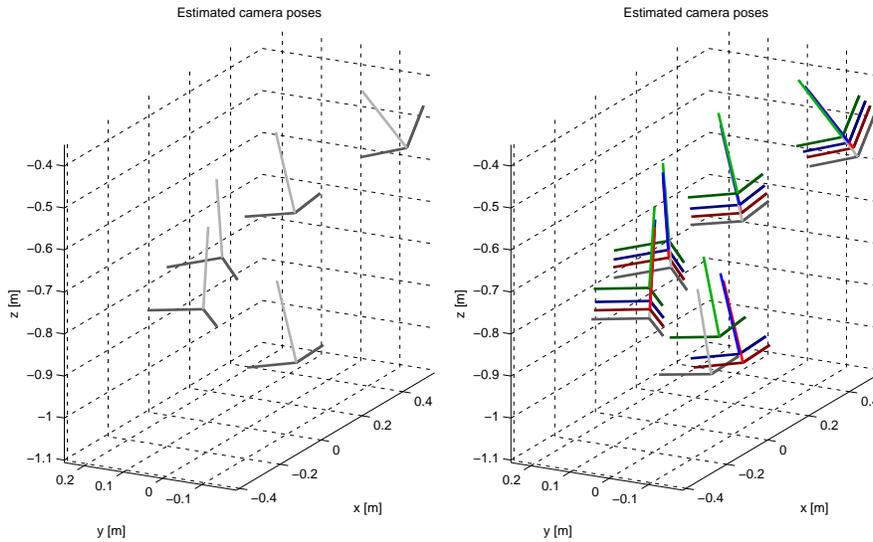


Figure 9: The estimated camera pose display; Left: After parameter initialization, and Right: After completion of a calibration; Gray: initial estimate, Green: optimized without distortion, Blue: optimized with distortion, and Red: after refinement.

2. **Copy** all the files from `Cameras/A000424242/PlenCalSmallExample/01` into a new folder, `Samples/Images/PlenCalSmallExample`. This will allow rectification of the images while maintaining the unrectified versions for comparison.
3. **Run `LFUtilDecodeLytroFolder`** to rectify the images. Use the command

```
DecodeOptions.OptionalTasks = 'Rectify';
LFUtilDecodeLytroFolder('Images/PlenCalSmallExample', ...
    [], DecodeOptions);
```

Examining the textual output, notice that the rectification has automatically selected the small sample calibration for these images, based on their zoom and focus settings.

A visual inspection of the rectified images probably shows poor results, due to the limitations of the small calibration dataset.

3.4.3 Cleaning Up and Validating

Remove the calibration file generated by deleting `CalInfo.json` from `Cameras/A000424242/PlenCalSmallExample` and re-run `LFUtilProcessCalibrations`. Repeating the above validation procedure with the default sample calibration in place yields more reasonable validation results, such as those shown in Fig. 10, despite a slight mismatch in camera parameters.

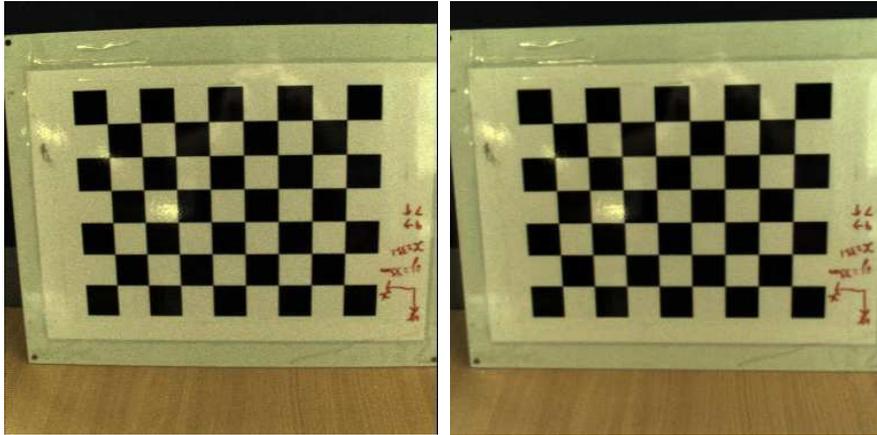


Figure 10: A rectified checkerboard; See also Fig. 4. More complete datasets are explored in [1].

3.5 Beyond the Samples: Working with Your Own Light Fields

Processing images from your own camera closely mirrors the examples covered so far. First, create a new folder parallel to the `Cameras/A000424242` folder, to contain your camera's white images and any calibrations you perform. A good naming convention is to name this folder to match your camera's serial number. Next, create a sub-folder for your white images. Your tree structure should now look like:

<code>Samples</code>	Top level of samples
<code>Cameras</code>	Stores info for one or more cameras
<code>A123412123</code>	Your camera's top level folder
<code>WhiteImages</code>	Your camera's white images
<code>A000424242</code>	The camera used to measure the samples
<code>CalZoomedOutFixedFoc</code>	A single calibration result
<code>WhiteImages</code>	White images for the sample camera
<code>Images</code>	Sample light field images

Following the procedure described in Appendix A, extract your camera's white images and place them in the newly created `WhiteImages` folder. Any calibrations you perform should sit in their own folders alongside the `WhiteImages` folder.

From the top level of the samples folder, run `LFUtilProcessWhiteImages` to process your camera's white images. The resulting grid models will be added to the white image database, and automatically applied to pictures taken with your camera.

You may end up with a complex tree structure with many sub-folders under `Images`. `LFUtilProcessWhiteImages` will search this structure recursively, decoding anything it identifies as a light field.

To rectify your own images, you will need to calibrate your camera. Follow the procedure described in the tour above, except using your own images stored

within your own camera's folder. Your first calibrations might, for example, go in `Samples/A123412123/CalZoomedOut`.

4 Decoding in Detail

4.1 Overview

This toolbox decodes lenslet-based light field images into a 4D light field structure following the decode process described in [1]. At its core, the inputs to this process are a white image and a lenslet image. The white image is an image taken through a diffuser, and is used to correct for vignetting (darkening near the edges of images), and to build a grid representing the locations of lenslet centers.

Each Lytro camera comes preloaded with a unique set of white images corresponding to a variety of zoom and focus settings. When decoding a light field picture, the white image is selected which most closely matches the zoom and focus settings of the camera when it took the picture. The white images can be extracted from the Lytro files following the instructions in Sect. A.1.

Before decoding light field pictures, the white images must be analyzed. This builds a series of grid models, one per white image, and a database listing available images. This only needs to be done once per camera, and the utility function `LFUtilProcessWhiteImages` is provided to automate the process.

For each picture to decode, a white image appropriate to that picture is selected based on the camera serial number, and zoom and focus settings, and the white image and raw lenslet image are passed to a decoding function which builds the 4D light field. The function `LFSelectFromDatabase` is provided to aid in selecting the appropriate white image for a light field, as demonstrated by `LFLytroDecodeImage`.

The following sections describe this workflow in more detail, and assumes that you have extracted the white images and copied light fields into a folder structure similar to that used in the quick tour, above. See Appendix A for details on dealing with the Lytro files.

4.2 Analyzing White Images

Each white image needs to be analyzed once in order to match a grid model to the lenslet locations. The utility `LFUtilProcessWhiteImages` automatically builds a grid model for each white image in your white image folder. If you wish to store your white images in a structure other than the default, simply change the `WhiteImageDatabasePath` variable in `LFUtilProcessWhiteImages` to point to your white images folder, or pass an argument `FileOptions` with the `WhiteImageDatabasePath` field set.

In the F01 camera, the white images come in two exposure levels. Both exposures are not needed, and only the brighter of the two is used by this toolbox. An examination of the white images also generally reveals some extra, very dark images. These are not useful, and are automatically ignored.

As `LFUtilProcessWhiteImages` steps through the white images, it saves the grid models as `.grid.json` files in the white images folder. It simultaneously builds a database keeping track of the serial number, zoom and focus settings associated with each white image. It saves this as `WhiteFileDatabase.mat`. This is utilized by the function `LFSelectFromDatabase` to select the white image appropriate for decoding a given light field picture.

4.3 Decoding a Lenslet Image

The decode procedure is demonstrated in `LFLytroDecodeImage`. This script first loads a lenslet image and associated metadata, selects the appropriate white image using `LFSelectFromDatabase`, then passes the lenslet image, metadata and white image to `LFDecodeLensletImageSimple`, which handles the bulk of the work.

`LFSelectFromDatabase` selects the appropriate white image based on serial number, zoom and focus settings. Presently zoom is prioritized over focus, though whether this is the optimal approach is an open question.

`LFDecodeLensletImageSimple` proceeds as described in [1] to decode the light field. This involves demosaicing, devignetting, transforming and slicing the input lenslet image to yield a 4D structure. More sophisticated approaches exist which combine steps into joint solutions, and they generally yield superior results, particularly near the edges of lenslets. The approach taken here was chosen for its simplicity and flexibility.

Some of the specifics of the decode process can be controlled, see the help text for the above functions.

4.4 Structure of the Decoded Light Field

A light field is fundamentally a four-dimensional structure. Roughly speaking, each pixel corresponds to a ray, and two dimensions define that ray’s position, while the other two define its direction. In the case of the the images measured by a lenslet-based camera such as the Lytro, two dimensions select a lenslet image, and two select a pixel within that lenslet’s image. By the convention followed in [1], the lenslet is indexed by the pair k, l (k is horizontal), and the pixel within the lenslet is indexed by i, j (i is horizontal).

The Lytro’s lenslets each yield approximately 9×9 useful pixels, and so the output of `LFUtilDecodeLytroFolder` has a size approximately 9 in i and j . Similarly, after removing the hexagonal sampling associated with the hexagonal lenslet array, the Lytro imagery yields approximately 380 pixels in both k and l . The actual number of samples depends on how the lenslet grid is aligned with the sensor, and will vary by camera.

Examining the output of `LFDecodeLensletImageSimple`, we see that it yields a light field `LF` which is a 5D array of size around $9 \times 9 \times 380 \times 380 \times 3$. Importantly, **the indexing order for LF is j, i, l, k, c** , where c is the RGB colour channel.

To examine a slice through the k and l dimensions, you might use the command `imshow(squeeze(LF(5,5, :, :, :)))`, yielding a view from the center of the i and j dimensions. This looks very much like the examples shown in Fig. 2.

To examine a slice through the i and j dimensions, you might use the command `imshow(squeeze(LF(:, :, 380/2, 380/2, :)))`, yielding an output similar to that shown in Fig. 11. Notice that this shows the shape of the image under a lenslet, and features darkened corner pixels that contain little or no information. Methods exist for “filling in” these regions of the light field, and may be explored in future releases of the toolbox.

`LFDecodeLensletImageSimple` provides a weight channel `LFWeight`, which represents the confidence associated with each pixel. A slice in i and j of such a channel is shown in Fig. 11. The weight channel is useful in filtering applications which accept a weighting term.

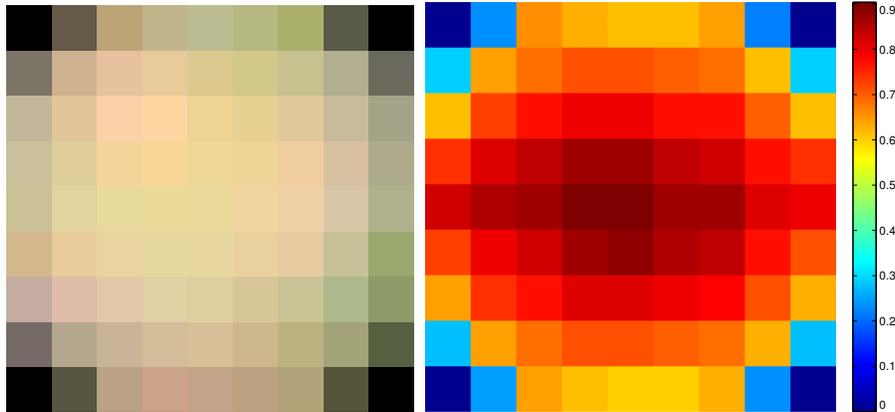


Figure 11: Light field for Sample 1 and its associated weight channel viewed in the i and j dimensions.

Note that `LFLytroDecodeImage` tacks the weight channel onto the variable `LF` to yield a four-channel structure, and it is in this four-channel format that `LFUtilDecodeLytroFolder` saves light fields. This is a convenient format for the light field, as the weight channel is often useful in processing light fields. `LFHistEqualize`, for example, uses this channel to ignore zero-weight pixels. It is likely that the weight channel will be useful in filtering operations in future versions of the toolbox.

To work with a light field without the weight channel – for example to visualize slices – simply index the first three channels, as in

```
imshow(squeeze(LF(5,5,:,: ,1:3)))
```

5 Calibration in Detail

The `LFUtilCalLensletCam` automatically progresses through the following calibration stages:

Checkerboard corner identification. Corner finding is the most time-consuming step, especially for dense checkerboards. First the zoom and focus settings of all the input images are compared, and a warning message is displayed if any of them differ. Next corners are automatically located in 2D slices of the checkerboard light fields. Output resembling that shown on the right in Fig. 8 allows visual confirmation that the extracted corners are sensible. It is normal that not all sub-images will have all corners successfully identified, due to vignetting and bleedthrough between lenslet images. Checkerboard corners for each image are stored in `*.CheckerCorners.mat` files alongside each input file.

Initialization of pose and intrinsic parameters. This begins by summarizing the checkerboard corner information into a single file at the top level of the calibration, `Cameras/A000424242/PlenCalSmallExample/CheckerboardCorners.mat`. Initial pose and intrinsic estimates are then computed and stored at the same level, in the calibration info file `CalInfo.json`.

Optimization without distortion. Intrinsic and poses are optimized, and the results are saved to `CalInfo.json`. The pose estimate display is updated with the new

pose estimates. The textual display shows the progress of the optimization, including the RMSE before and after each stage of the optimization. Each optimization stage also shows a Matlab-generated optimization display, showing first-order optimality – see Matlab’s documentation for more on this.

Optimization with distortion. This completes the camera model by including lens distortion. Again the pose estimate display and textual output are updated.

Refinement. This simply repeats optimization with distortion to further refine the camera model and pose estimates.

5.1 Calibration Results

The calibration results are stored in the calibration information file, `CalInfo.json`. The calibrated estimates are described in detail in [1], and include:

- **Lenslet grid model:** describes the rotation, spacing and offset of the lenslet images on the sensor.
- **Plenoptic intrinsic model:** a 5×5 matrix \mathbf{H} relating a pixel index $\mathbf{n} = [i, j, k, l, 1]^T$ to an undistorted ray $\phi^u = [s, t, u, v, 1]^T$, following $\phi^u = \mathbf{H}\mathbf{n}$.
- **Distortion parameters:** describe radial distortion in ray *direction*, employing the small angle assumption such that $\theta = [\theta_1, \theta_2] \approx [dx/dz, dy/dz]$ for each ray. The five distortion parameters are $\mathbf{b} = [b_s, b_t]$ and $\mathbf{k} = [k_{1..3}]$, where \mathbf{b} captures decentering and \mathbf{k} are radial distortion coefficients. The complete distortion vector is in the order $\mathbf{d} = [\mathbf{b}, \mathbf{k}]$. If θ^u and θ^d are the undistorted and distorted 2D ray directions, respectively, then $\theta^d = (1 + k_1 r^2 + k_2 r^4 + \dots) (\theta^u - \mathbf{b}) + \mathbf{b}, r = \sqrt{\theta_s^2 + \theta_t^2}$.

Because the lenslet grid model forms part of the calibration, it is crucial that light fields to which a calibration is applied be decoded with the same grid parameters used during the calibration process. The software performs a rudimentary check and raises a warning if the lenslet grid model used to rectify a light field differs significantly from that used to decode it.

5.2 Rectification Results

Finding the ray to which a light field sample corresponds in an *unrectified* light field is relatively complex, requiring application of both the intrinsic matrix and distortion model. Once a light field is rectified, however, the *rectified* light field’s intrinsic matrix directly relates samples to rays, as in $\phi = \mathbf{H}\mathbf{n}$. The rectified intrinsic matrix is saved in each rectified light field as `RectOptions.RectCamIntrinsicsH`.

As a simple example, for the small calibration example dataset,

```
n = [1, 1, 1, 1, 1]';
p = RectOptions.RectCamIntrinsicsH * n;
```

Results in the ray $p = [0.0015, 0.0015, -0.34, -0.34, 1]^T$. Similarly, $n = [5, 5, 190.5, 190.5, 1]^T$ yields the ray $p = [0, 0, 0, 0, 1]^T$, because this n corresponds to the center of the sampled light field (recall the light field size is $9 \times 9 \times 380 \times 380$), and so corresponds to the central ray.

5.3 Controlling Rectification

Rectification accepts as an optional parameter the desired intrinsics of the rectified light field – i.e. you can specify the value you want in `RectOptions.RectCamIntrinsicsH`. By default the calibrated intrinsic matrix takes on a conservative value yielding square pixels in s, t and in u, v . You may wish to change this if, for example, non-square pixels are desired.

New in Version 0.3 is a helper function for building this matrix, `LFCalDispRectIntrinsics`. The recommended usage pattern is to load a light field, call `LFCalDispRectIntrinsics` once to set up the default intrinsic matrix, manipulate the matrix, then visualize the manipulated sampling pattern prior to employing it in one or more rectification calls. Assuming `IMG_001` has been decoded but not rectified, a typical process might look like this:

```
load('Images/IMG_0001__Decoded.mat');
RectOptions = ...
    LFCalDispRectIntrinsics( LF, LFMetadata, RectOptions );
```

this loads the light field then sets up the default intrinsic matrix, generating a display showing the sampling pattern, as in Fig. 12.

If we wanted to sample closer to the horizontal u edges of this light field, at the cost on working with non-square pixels, we could increase $\mathbf{H}(3,3)$, as in:

```
RectOptions.RectCamIntrinsicsH(3,3) = ...
    1.1 * RectOptions.RectCamIntrinsicsH(3,3);
RectOptions.RectCamIntrinsicsH = LFRecenterIntrinsics( ...
    RectOptions.RectCamIntrinsicsH, size(LF) );
LFCalDispRectIntrinsics( LF, LFMetadata, RectOptions );
```

This increases the extent of the samples along u , then re-centers the sampling via `LFRecenterIntrinsics`, then displays the resulting sampling pattern, as shown in Fig. 12.

Finally, the appropriate call to `LFUtilDecodeLytroFolder` will rectify multiple light fields with the requested intrinsic matrix:

```
DecodeOptions.OptionalTasks = 'Rectify';
LFUtilDecodeLytroFolder([], [], DecodeOptions, RectOptions);
```

Note that the same matrix can be applied to any light field, but that the resulting sampling pattern will differ for different cameras and focus / zoom settings. Fig. 13 shows the result of applying the example rectifications from Fig. 12 – note that more of the recorded imagery is visible in the second image, but its non-square pixels must be accounted for in subsequent processing steps.

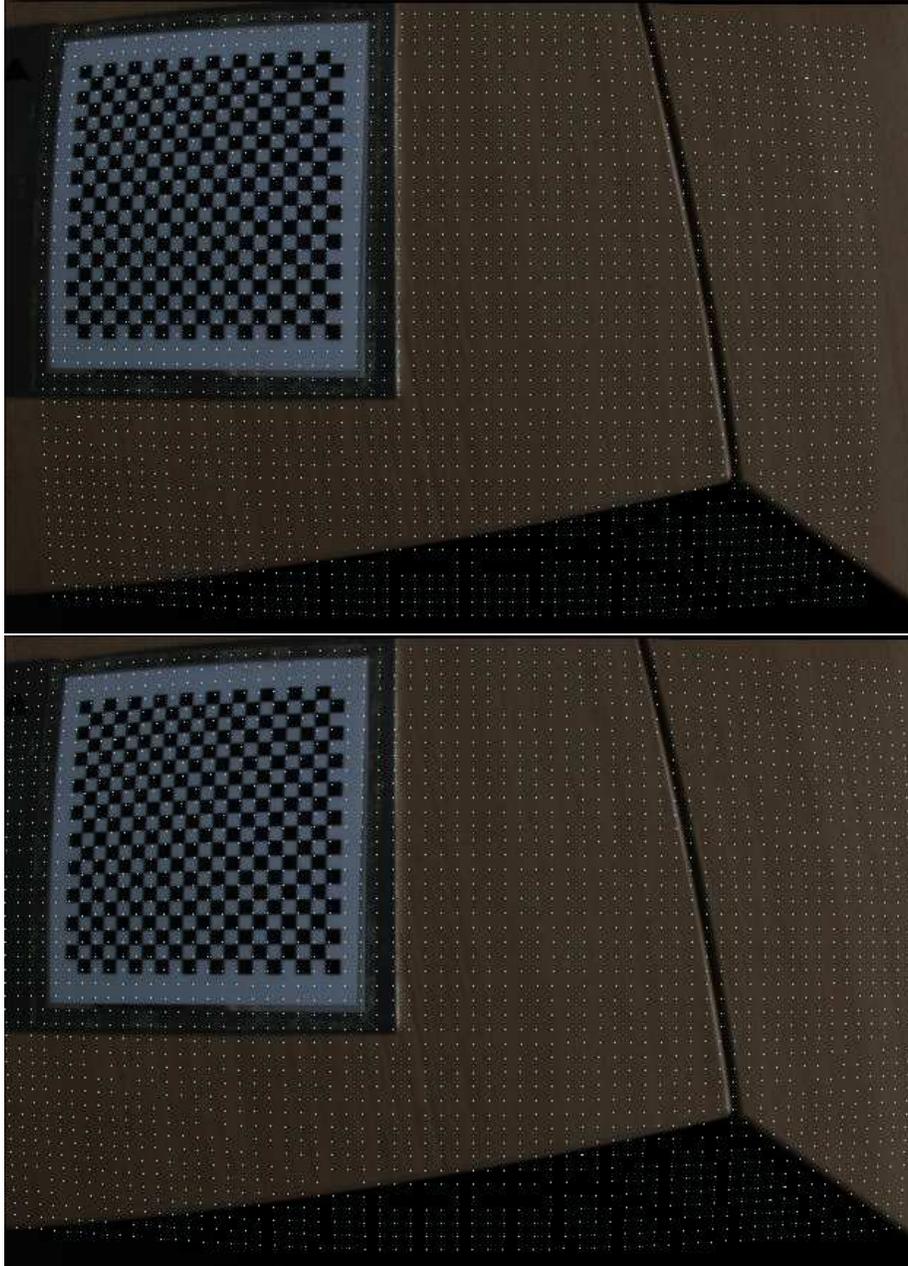


Figure 12: The default and adjusted sampling patterns. Here the sampling pattern has been stretched horizontally, incorporating more of the measured image, but yielding rectangular pixels. The following figure shows the result of applying each of these.

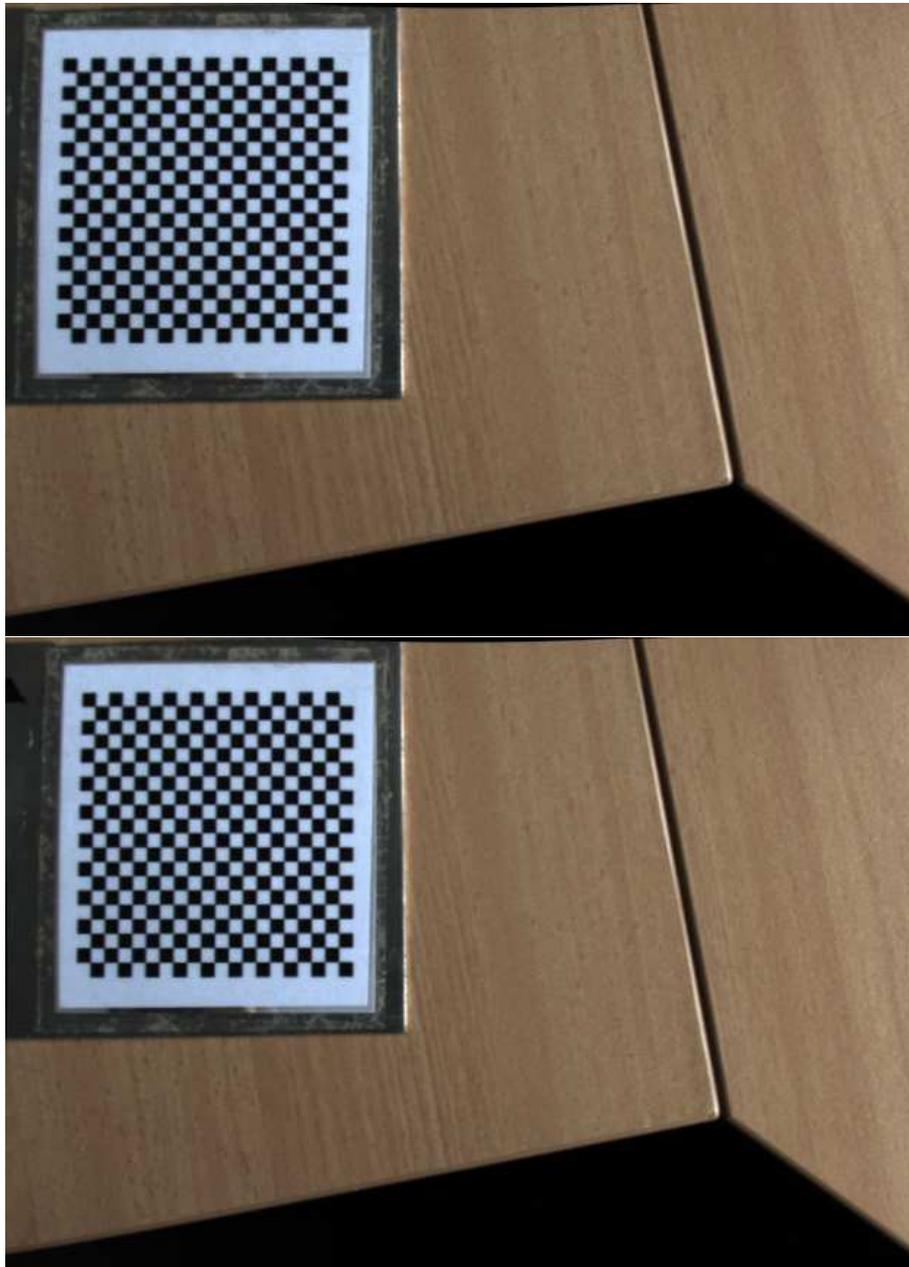


Figure 13: Rectification applied with the default and adjusted sampling patterns described in the previous figure.

Appendix A Working with Lytro Files

As of version 0.3 of the toolbox, an external tool is no longer required to extract white images, pictures or metadata from the files generated by the Lytro software.

A.1 Extracting White Images

Every camera has a unique database of white images needed in decoding. On Windows machines, the white image data is found in a folder of the form

```
<drive_letter>:\Users\<username>\AppData\Local\Lytro\cameras\  
    sn-<serial_number>
```

while on a MAC, the relevant location is

```
/Users/<username>/Lytro.lytrolib/cameras/sn-<serial_number>
```

A concrete example on a Windows machine is

```
C:\Users\Bob\AppData\Local\Lytro\cameras\sn-A000424242
```

The Lytro Illum can save its white images in a compressed file on its SDCard as part of the “pairing process” – see the Lytro literature on creating this “Pairing Data”. The data folder or pairing data file contain files named `data.C.0`, `data.C.1` and so on. These are in a Lytro-specific storage format, and can be unpacked using `LFUtilUnpackLytroArchive`.

For example, after uncompressing the pairing data or copying the contents of the data folder into `Cameras/<YourSerial>/WhiteImages`, run `LFUtilUnpackLytroArchive` from the top of the Samples folder. The function will by default search the `Cameras` folder for all archives and unpack them.

Of the resulting extracted files, those we are interested in have names like

```
MOD_0000.RAW  
MOD_0000.TXT  
MOD_0001.RAW  
MOD_0001.TXT  
...
```

The `raw` files are white images corresponding to a variety of zoom and focus settings, while the `txt` files contain the metadata we require to sort out which is which. The other files contain a wealth of information about your Lytro, but are not utilized in this revision of the toolbox. Once unpacked, you may safely remove the copied `data.C.*` files.

A.2 Locating Picture Files

This version of the toolbox can read Lytro LFP files directly, using the function `LFReadLFP`. The toolbox is also compatible with `.raw` files extracted using an external LFP tool.

Lytro Desktop version 4 and higher make it easy to find LFP files, as they are stored in your operating system’s default Pictures folder – look for a folder of the form `My Pictures/Lytro Desktop/Libraries/Lytro Library.lytrolibrary/`. The picture library takes on a complex directory structure, with many sub-folders. You may copy this structure directly into your working folder – the toolbox will recursively search sub-folders when decoding light fields.

The desktop software can also export light fields to a location of your choice.

If working with an Illum, you may copy the files straight off the camera, as it directly exposes its file system over USB.

If you're running an older versions of the Lytro Desktop software, Lytro picture files may be found in an **images** folder alongside the **cameras** folder where the white images are stored. i.e. on a Windows machine the default location is

```
<drive_letter>:\Users\<username>\AppData\Local\Lytro\images\*
```

and on a MAC it's

```
/Users/<username>/Lytro.lytrolib/images/*
```

where the '*' at the end takes on numerical values, like 01, 02 and so on.

A.2.1 LFP, LFR, lfp or lfr?

The Lytro LFP is a container format, and may contain one of several types of data. The files containing light fields are generally obvious based on their size – about 16 MBytes for the F01, and 55 MBytes for the Illum. The file extension varies based on the source of the files, with exported files, on-camera files and image library files variously taking on the four variants of extension shown in this section's heading.

By default, `LFUtilDecodeLytroFolder` recursively searches for files with any of these extensions, as well as the **raw** files employed by previous toolbox versions, and decodes anything it can make sense of. Focal stacks and other files are also stored as .lfp files, and `LFUtilDecodeLytroFolder` will ignore these files.

A.2.2 Thumbnails

Thumbnails are built into some LFP files. The function `LFUtilExtractLFPThumbs` will find thumbnails and save them to disk.

Appendix B Function Reference

This is a quick list of top-level functions organized by task. Please refer to the documentation included in each function for further information, and the `SupportFunctions` folder for the inner workings of the toolbox.

Several of the functions begin by defining default values for all of their arguments, as in `LFUtilDecodeLytroFolder`. As a result, these can either be called as functions or run directly without any arguments. When calling without arguments, edit the values in the code to match your desired settings. When calling as a function, it is only necessary to pass those arguments and fields which differ from the defaults. Pass an empty array `[]` to omit a parameter, and omit fields that are to take on default values.

See the help for `LFUtilDecodeLytroFolder` for examples of this parameter-passing scheme.

Decoding / Input

LFLytroDecodeImage

Decode a Lytro image from an `LFP` or `raw` file. Can be called directly to decode a single image into memory, or called indirectly through `LFUtilDecodeLytroFolder`.

LFReadGantryArray

Loads gantry-style light fields, e.g. the Stanford light fields found at <http://lightfield.stanford.edu>.

LFUtilDecodeLytroFolder

Utility for decoding, colour correcting and rectifying Lytro imagery. Can process multiple light fields; recursively searches folder structures; accepts filename specifications including wildcards. Automatically selects appropriate white images and calibration files from multiple cameras across multiple zoom and focus settings. Will incrementally apply operations to files so that, for example, previously-decoded light fields can be incrementally colour-corrected, rectified or both without needing to repeat operations. Results are saved to disk. See Figs. 2, 4 and 10 for example output.

Demonstrates `LFlytroDecodeImage`, `LFColourCorrect`, `LFHistEqualize`, and `LFCalRectifyLF`.

Decoding relies on a white image database having been constructed by `LFUtilProcessWhiteImages`, and rectification similarly relies on a calibration database having been created by `LFUtilProcessCalibrations`.

LFUtilProcessWhiteImages

Processes a folder populated with white images, generating a grid model (`.grid.json`) for each, and a white image database (`WhiteFileDatabase.mat`) used to select the white image appropriate to a light field. Dark images are automatically detected and skipped.

Filtering

LFBUILD2DFREQFAN NEW

Construct a 2D fan filter in the frequency domain. Apply this filter with [LFFILT2DFFT](#).

LFBUILD2DFREQLINE NEW

Construct a 2D line filter in the frequency domain. The cascade of two line filters, applied in s,u and in t,v, is identical to a 4D planar filter, e.g. that constructed by [LFBUILD4DFREQPLANE](#). Apply this filter with [LFFILT2DFFT](#).

LFBUILD4DFREQDUALFAN NEW

Construct a 4D dual-fan filter in the frequency domain. Apply this filter with [LFFILT4DFFT](#).

LFBUILD4DFREQHYPERCONE NEW

Construct a 4D hypercone filter in the frequency domain. Apply this filter with [LFFILT4DFFT](#).

LFBUILD4DFREQHYPERFAN NEW

Construct a 4D hyperfan filter in the frequency domain. This is useful for selecting objects over a range of depths from a lightfield, i.e. volumetric focus. Apply this filter with [LFFILT4DFFT](#).

LFBUILD4DFREQPLANE NEW

Construct a 4D plane filter in the frequency domain. This is useful for selecting objects at a single depth from a lightfield, and is similar in effect to refocus using, for example, the shift sum filter [LFFILTSHIFTSUM](#). Apply this filter with [LFFILT4DFFT](#).

LFDemoBasicFiltLytroF01 NEW

Demonstrates some of the basic filters on Lytro F01-captured imagery.

LFDemoBasicFiltIllum NEW

Demonstrates some of the basic filters on Lytro Illum-captured imagery.

LFDemoBasicFiltGantry NEW

Demonstrates some of the basic filters on Stanford light field archive light fields.

LFFILT2DFFT NEW

Applies a 2D frequency-domain filter to a 4D light field using the FFT.

LFFILT4DFFT NEW

Applies a 4D frequency-domain filter using the FFT.

LFFILTSHIFTSUM NEW

The shift sum filter is a spatial-domain depth-selective filter, with an effect similar to planar focus.

Image Adjustment

LFColourCorrect

Applies a colour balance vector, an RGB colour correction matrix, and gamma correction. Usage is demonstrated in [LFUtilDecodeLytroFolder](#).

LFHistEqualize

Adjusts the brightness of a light field based on histogram equalization. Capable of handling colour and monochrome images – colour images are converted to HSV, and the value channel is equalized. Capable of handling different input dimensionalities including 2D images and 4D light fields. If a weight channel is present as a fourth colour channel, it is used to ignore zero-weight pixels. Usage is demonstrated in [LFUtilDecodeLytroFolder](#).

Visualization

LFDisp NEW

Convenience function to display a static, 2D slice of a light field. The centermost image is taken in s and t . Also works with 3D arrays of images.

LFDispMousePan

Display 2D slices of the light field with a rudimentary parallax effect. Click and drag in the image to change the point of view. An optional parameter controls the display size. Note that darkening at the edges of unfiltered light fields mean that the effect is best near the center of the spatial range. For an automatically animated display, see [LFDispVidCirc](#). The function re-uses previously-opened light field display windows. Note that changing display size requires that the display window be closed prior to calling this function.

LFDispVidCirc

Animated display showing 2D slices of the light field, similar to [LFDispMousePan](#) except the motion is preset to a circular path. Optional parameters include the radius of the circular path, animation speed, and display size.

LFFigure

Replacement for Matlab's "figure" which doesn't steal focus, originally sfigure by Daniel Eaton.

Calibration

LFCalDispEstPoses

Visualize camera pose estimates. Called by [LFUtilCalLensletCam](#).

LFCalDispRectIntrinsics

Helper for setting up and visualizing intrinsics requested in rectification, see also [LFRecenterIntrinsics](#).

LFCalRectifyLF

Applies a calibration to rectify a light field. The desired intrinsic matrix can be provided, or computed automatically from the calibrated intrinsics. Demonstrated by [LFUtilDecodeLytroFolder](#).

LFRecenterIntrinsics

Recenters a light field intrinsic matrix, useful for modifying intrinsics requested in [LFCalRectifyLF](#), see [LFCalDispRectIntrinsics](#).

LFUtilCalLensletCam

Runs through all the steps of a lenslet-based camera calibration.

LFUtilProcessCalibrations

Builds a database of calibrations to allow selection of the appropriate calibration for a given light field.

File I/O

LFFindFilesRecursive

Recursively searches a folder for files matching one or more patterns. Refer to this to understand the path parameters to [LFUtilDecodeLytroFolder](#), [LFUtilExtractLFPThumbs](#) and [LFUtilUnpackLytroArchive](#).

LFReadLFP

Reads Lytro `lfp`/`lfr` light field files.

LFReadMetadata

Reads `json` files.

LFReadRaw

Reads 10, 12 and 16-bit `raw` image files.

LFWriteMetadata

Writes `json` files.

Utility / Convenience

LFMatlabPathSetup

Sets up the Matlab path to include the LF Toolbox. This must be re-run every time Matlab restarts, so consider adding a line to `startup.m` as shown in Sect. 2.

LFUtilUnpackLytroArchive

Extracts white images and other files from a multi-volume Lytro archive.

LFUtilExtractLFPThumbs

Extracts thumbnails from LFP files and writes them to disk.