Light Field Toolbox for Matlab

v0.5.2

This is a toolbox for working with light field imagery in MATLAB. Features include loading, visualizing, and filtering light fields, and decoding, calibration, and rectification of lenslet-based imagery.

The most recent release and development versions are here: `https://github.com/doda42/LFToolbox`.

The complementary LiFF light field feature toolbox is here: `http://dgd.vision/Tools/LiFF`.

What's New / Development Plan

For a complete list, see CHANGELOG.txt. v0.5 highlights:

- Linear refocus super-resolution using LFFiltShiftSum, see LFDemoRefocusSuperres
- New display functions LFDispLawnmower, LFDispTiles, LFDispTilesSubfigs, LFDispProj, LFDispProjSubfigs
- LFReadESLF, LFWriteESLF
- Improved decode performance and speed
- Improved calibration accuracy
- LFDisp* functions are better behaved, now display in the active figure window

Future plans include more significant changes to lenslet-based decode and calibration, and support for a broader range of cameras.

Compatibility

Reverse-compatibility: Changes to interfaces have been minimised, LFDispVidCirc is the main exception, with a new parameter structure.

Previously generated calibration files should be re-generated, and to benefit from performance improvements to decoding, white images should be regenerated. See Appendix B for details.

Matlab: LFToolbox 0.5 was written in MATLAB 2020a, but should be compatible with earlier versions.

File Formats: The toolbox can load gantry / array-style folders of images, ESLF files, and raw lenslet-based images.

Plenoptic 1.0 cameras are supported through decoding, calibration, and rectification of imagery. Functions are most easily applied to Lytro imagery. The toolbox can also be applied to other lenslet-based Plenoptic 1.0 cameras, but this is not yet well documented. Calibration of Lytro Illum cameras is experimental.

Plenoptic 2.0 cameras are not well supported. Use with some cameras is possible but not well documented. Multi-focal lenslet-based cameras are not well supported.

Lytro Software: The toolbox is compatible with files generated using Lytro Desktop 4 and 3.

Contributing / Feedback

Suggestions, bug reports, code improvements and new functionality are welcome – email `Donald.Dansereau+LFToolbox` at `gmail` dot `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 Installation

1.1 Installing the Toolbox

There are three main options for installation:

1. Download a `.zip` file through the Mathworks site https://au.mathworks.com/matlabcentral/fileexchange/75250-light-field-toolbox
2. Download a `.zip` file from Github https://github.com/doda42/LFToolbox
3. Clone the repo from GitHub

Options 2 and 3 have the advantage that you may work with the most recent release (the master branch), or a development branch with experimental functionality.

Unzip or clone the files into an appropriate location with a meaningful top-level folder name, e.g. `LFToolbox0.5`. Run the convenience function LFMatlabPathSetup to set up the Matlab path. This must be run every time Matlab starts, so consider adding a line to `startup.m`, e.g.

```
run('~/MyMatlabCode/LFToolbox0.5/LFMatlabPathSetup.m')
```

Be sure to remove any similar calls for previous toolbox versions. To check your installation:

- Run `LFToolboxVersion` to check the toolbox version
- Type `which LFToolboxVersion` to confirm the path to the toolbox folder
- Type `help <toolbox folder name>`, e.g. `help LFToolbox0.5` to display a list of toolbox functions

1.2 Downloading Samples

- LFToolbox v0.5 Sample Pack, including example ESLF and Lytro F01 and Illum images: http://www-personal.acfr.usyd.edu.au/donald/LFToolbox0.5_Samples.zip
- Small Sample Calibration: http://www-personal.acfr.usyd.edu.au/ddan1654/PlenCalSmallExample.zip
- Gantry-style and large collections of ESLF files at Stanford: http://lightfields.stanford.edu

1.3 Additional Resources

- Questions about the toolbox should go to the Light Field Vision mailing list: https://groups.google.com/forum/#!forum/lightfieldvision
- Additional datasets and community links: http://dgd.vision/Tools/LFToolbox

2 Conventions

2.1 Optional Function Arguments

Many toolbox functions accept optional function arguments, taking on default values if no value is provided. Simply omit arguments at the end of the list, or pass an empty array `[]` within the list. For example, for the function

```
LFDispMousePan( LF, ScaleFactor, InitialViewIdx, Verbose )
```

all arguments except the first are optional. Any of the following are valid:

```
LFDispMousePan( LF )
LFDispMousePan( LF, 4 )               % set a scale factor of 4
LFDispMousePan( LF, 4, [3,3] )        % .. also set initial view index 3,3
LFDispMousePan( LF, [], [3,3] )       % initial view 3,3, default scale
LFDispMousePan( LF, [], [3,3], true ) % initial view 3,3, verbose on
LFDispMousePan( LF, 4, [], true )     % scale factor 4, verbose on
```

2.2 Struct Arguments

Some functions accept structs of arguments. A helpful shorthand for calling these is using MATLAB's `struct` function, that allows inline struct construction, e.g.

```
LFUtilDecodeLytroFolder('Images', [], struct('OptionalTasks', 'ColourCorrect'))
```

See the help for LFUtilDecodeLytroFolder for more examples of these argument-passing conventions.

2.2.1 Continuous vs Discrete Indices

In formal presentation, discrete indices i, j, k, l are distinct from continuous spatial coordinates s, t, u, v. In the toolbox, when it is clear that the light field is sampled, the latter is generally understood to refer to sample indices.

2.3 Light Field Indexing

In MATLAB, to index a 2D colour image I at the coordinates x, y, c corresponding to horizontal position, vertical position, and colour, we use the indexing order $I(y, x, c)$.

Generalizing this to a 4D light field L which we wish to index at the horizontal, vertical ray position s, t, horizontal, vertical ray direction u, v, and colour channel c, we use the indexing order $L(t, s, v, u, c)$.

2.3.1 Colour Channels

Colour light fields have at least three colour channels, one for each of red, green, and blue. The toolbox can also handle monochrome light fields, with a single colour channel.

For light fields that also have per-pixel weight information, there is an additional colour channel. Weight indicates confidence, so for example a pixel that is significantly amplified to compensate for vignetting will have a proportionally lower weight because we are less confident of its true value. A weight of zero means we do not know the value of the pixel.

2.4 Two-Plane Parameterization

The intrinsic matrices generated through calibration use a *relative* two-plane parameterization, with a plane separation $D = 1$m. This maps each pixel index $[i, j, k, l]$ to a corresponding ray in space $[s, t, u, v]$, where each of s, t, u and v is measured in meters. This combination unambiguously defines each ray in metric space including scale.

3 A Quick Tour

3.1 Working with Lytro Light Fields

3.1.1 Decoding? Calibration? Rectification?

The toolbox uses the following terminology:

1. **Decoding**: The lenslet array encodes the 4D light field onto a 2D sensor, and decoding undoes this process.
2. **Calibration**: This is characterizing the camera's optics to map each measured pixel to a ray in space.
3. **Rectification**: Removing distortions in the decoded light field to simplify the pixel-to-ray mapping to a linear relationship. In the simplest case, the resulting light field looks like what an array of parallel pinhole cameras would measure.

See Sect. 4 and [1] for further details.

3.1.2 Decoding the Samples

1. **Install** the LF Toolbox and following the instructions in Sect. 1.
2. **Download** the sample light field pack at `http://www-personal.acfr.usyd.edu.au/ddan1654/LFToolbox0.5_Samples.zip` and decompress into its own folder. The samples folder structure was chosen for easy addition of your own cameras and calibrations:

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

3. `cd <sample folder>` inside MATLAB to change to the top level of the samples folder. If you are in the correct location, the Matlab command `ls` should list the top-level folders and README file:

```
Images   Cameras   README
```

4. **Run LFUtilProcessWhiteImages** to build a white image database. This searches the `Cameras` folder for white (flat-field) 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.

   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 a set of figures similar to Fig. 1 will be presented for visual confirmation that

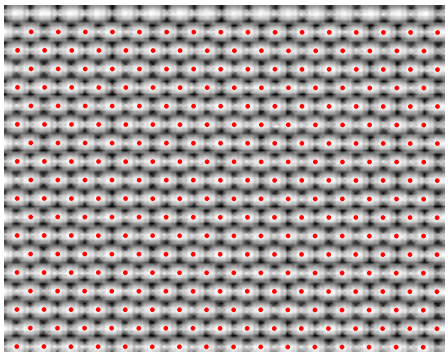

Figure 1: Example of a white (flat-field) image showing estimated lenslet centers as red dots.

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 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, as depicted in Fig. 1.

5. **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.

6. (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.

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.

You may wish to apply histogram stretching using LFHistEqualize. Illum imagery is not gamma-corrected. Example colour-corrected output is shown in the bottom row of Fig. 2, and in Fig. 3.

3.1.3 Rectifying the Samples

Still operating from the top level of the samples folder:

1. **Run LFUtilProcessCalibrations** to locate and catalogue camera calibrations. The result is stored in `Cameras/CalibrationDatabase.mat`.
2. **Run LFUtilDecodeLytroFolder** to rectify a specific light field:

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

The `CalibrationDatabase` file allows selection of the 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.

To rectify your own light fields, you must calibrate your camera. See Sect. 5.

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

3.1.4 Using a Different Folder Structure

You may wish to organize files differently than the toolbox default. A common use case is storing the `Camera` folder in a single global location outside the folder structure of the images. The toolbox supports this via function parameters specifying file locations. For example, to decode images using a global camera folder:

```
cd <top level path of your light field images>
DecodeOptions.WhiteImageDatabasePath = '<path to your cameras folder>';
LFUtilDecodeLytroFolder('.',[],DecodeOptions);
```

Note that you must still run LFUtilProcessWhiteImages at the top level of your cameras folder. Use `RectOptions.CalibrationDatabaseFname` to specify the path to calibration files.

3.2 Displaying Light Fields

1. **Load** a light field, e.g.

   ```
   load('Images/Illum/Lorikeet__Decoded.mat','LF');
   ```

2. **Run LFDisp** to display the central view of the light field.

   ```
   LFDisp( LF )
   axis image    % for correct aspect ratio
   ```

   To brighten the display through simple gamma correction, we can raise the light field to a power, but this first requires us to convert the light field to a floating point format first:

   ```
   LF = LFConvertToFloat(LF);
   LFDisp( LF.^0.5 )
   axis image
   ```

   This is not very efficient, as it raises the entire light field to the power 0.5, then displays only the central view. You can use the following trick to nest LFDisp commands:

   ```
   LFDisp( LFDisp(LF).^0.5 )
   axis image
   ```

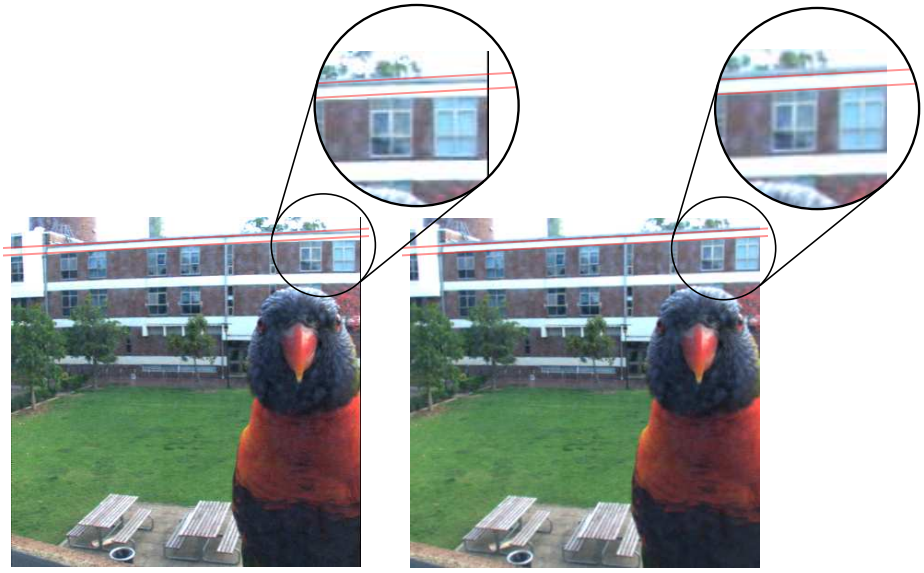

Figure 4: F01 Sample 2 before and after rectification, with insets showing the reversal of lens distortion.

This works because LFDisp returns the image as an output argument. The innermost LFDisp command extracts the central view of the light field, and only this view is then raised to 0.5. The result is passed to the outermost LFDisp for display. This approach can be applied with other display functions including LFHistEqualize:

```
LFDisp( LFHistEqualize(LFDisp(LF)).^0.5 )
axis image
```

3. **Run LFDispMousePan, LFDispVidCirc, or LFDispLawnmower** to visualize the light field with a shifting perspective, e.g.

   ```
   LFDispMousePan( LF )
   ```

   LFDispMousePan requires mouse input, click and drag in the window to change the perspective. The other functions animate through the images in the light field automatically. Try a larger display with `LFDispMousePan(LF, 2)` or `LFDispVidCirc(LF, [], 2)`, which doubles the displayed size.

4. **Run LFDisp** to display other slices of the light field:

   ```
   load('Images/Illum/Lorikeet__Decoded.mat','LF');
   LFDisp( LF(9,:,226,:,:) );
   ```

   displays a slice in s, u at $t = 9$, $v = 226$. This is best viewed with a non-square aspect ratio by expanding the window horizontally.

   ```
   LFDisp( LFHistEqualize( LFDisp( LF(9,:,226,:,:) ) ).^0.7 )
   ```

   displays the same contrast-stretched and gamma-corrected.

5. **Run LFDispTiles** to visualize the light field as a 2D tiling of 2D images:

```
load('Images/F01/IMG_0005__Decoded.mat','LF');
LFDispTiles(LF, 'stuv')
```

to display a tiling of u, v slices in s, t. This is a large image, you can display a subset of the views using

```
LFDispTiles( LF, 'stuv', struct('SubsampRate',2))
```

which will display every other sample in each of the four dimensions, or

```
LFDispTiles( LF(:,:, 85:100, 230:245, :) )
```

to display a crop in u, v. As with LFDisp, LFDispTiles returns the displayed image, allowing nesting:

```
LFDisp( LFHistEqualize( LFDispTiles( LF(:,:,85:100, 230:245,:) ) ) )
```

stretches the contrast of the previous example.

```
LFDispTiles( LF, 'uvst' )
axis image
```

will display a tiling of s, t slices in u, v. Zoom in on the display to see individual lenslet images, or crop the light field when calling the display function, as in

```
LFDispTiles( LF(:, :, 85:100, 230:245, :), 'uvst' )
```

6. **Run LFDispProj** to visualize the light field as a projection onto a 2D plane

```
load('Images/Illum/Lorikeet__Decoded.mat','LF');
LFDispProj( LF, 3,4 )
axis image
```

projects onto dimensions 3 and 4 (v and u) by adding along the remaining dimensions, s and t. The result is equivalent to refocus at the zero-slope depth.

```
LFDispProj( LF, 3,4, 'max' )
axis image
```

does the same but finds the maximum value along s, t for each u, v sample, yielding an interesting effect.

```
LFDispProj( LF, 1,2 )
axis image
```

projects onto the s, t plane, revealing the lenslet vignetting pattern. It is mostly flat because it has been corrected in decoding. The weight channel shows us a better picture of this:

```
LFDispProj( LF(:,:,:,:,4), 1,2 )
colormap gray
axis image
```

We can also project onto s, u or t, v slices

```
LFDispProj( LF, 2,4 )
axis normal
```

this is best viewed with a non-square aspect ratio by resizing the window. The contrast on this image is low since all the differently sloped parts of the scene are adding together. To get more of the structure apparent in the projection onto s, u, find the max along s, v rather than the mean:

```
LFDispProj( LF, 2,4, 'max' )
```

3.3 Loading Gantry-style Light Fields

Camera gantries and camera arrays yield ordered collections of individual image files, with each image corresponding to a different aperture position. LFReadGantryArray will read such an array of images as a light field.

1. **Download** an image archive from the Stanford gantry-based 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 the following will allow LFDemoBasicFiltGantry to run unmodified.

`StanfordGantry`	Top of Stanford gantry 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 gantry samples:

```
cd <path to top of Stanford gantry samples>
LF = LFReadGantryArray('LegoKnights/rectified', struct('UVLimit', 256));
```

   The `UVLimit` option scales the images as they're read to a size of 256x256. By default LFReadGantryArray looks for `.png` files, and assumes an array of 17 x 17 images in row major order. See the function help for specifying other filenames, array sizes, and file orderings.

4. **Run `size LF`** to check the light field size: `17 x 17 x 256 x 256 x 3`.
5. **Run LFDispMousePan** or other display functions to display the loaded light field.

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

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

Some gantry imagery follows a lawnmower pattern rather than direct raster scan order. 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,:,:,:);  % correct lawnmower ordering
```

Some gantry imagery follows a different handedness in horizontal and vertical directions, effectively flipping the order of images in s or t. The demo function LFDemoBasicFiltGantry contains a list of Stanford gantry light fields and demonstrates how to flip the appropriate ones along s.

3.4 Working with ESLF Files

Many of the large online datasets are stored as ESLF files. These collapse the light field into a 2D image by tiling s, t slices in u, v. Because they are 2D images, they can

be saved using any 2D image format. Popular choices are 8-bit and 16-bit `png` files for their lossless compression, and `jpg` files, for their compact size.

`png` files support storage of an 'alpha' channel, and some tools use this as a binary mask to indicate which pixels are valid. The toolbox can also save weight values in the alpha channel, indicating per-pixel confidence. By default, LFReadESLF will load the alpha channel as a weight channel if it is present, but LFWriteESLF will only write to the alpha channel if this is requested via the command arguments.

3.4.1 Reading ESLF Files

1. `cd <sample folder>` inside MATLAB to change to the top level of the samples folder.
2. **Run LFReadESLF** to read the sample ESLF file:

   ```
   LF = LFReadESLF('Images/ESLF/Plant.eslf.jpg');
   ```

3. **Run LFDispMousePan** to display the light field:

   ```
   LFDispMousePan(LF);
   ```

4. To load an ESLF file with a different number of pixels per lenslet, **Run LFReadESLF** with the appropriate option, e.g. after writing `Jacaranda.eslf.png` in the next section, read it using

   ```
   LF = LFReadESLF('Jacaranda.eslf.png', [15,15]);
   ```

3.4.2 Writing ESLF Files

1. Load a light field, and **Run LFWriteESLF** to save it a PNG-compressed ESLF:

   ```
   load('Images/Illum/Jacaranda__Decoded.mat','LF');
   WriteAlpha = true;
   LFWriteESLF( LF, 'Jacaranda.eslf.png', WriteAlpha );
   ```

   Because the loaded light field is 16-bit and has a weight channel, and we requested the alpha channel be written, the output `png` is 16-bit and has four channels, occupying 418 MBytes. If we do not request the alpha channel be written,

   ```
   LFWriteESLF( LF, 'Jacaranda.eslf.png' );
   ```

   the file occupies 300 MBytes. If we first convert to 8-bit pixels using LFConvertToInt and omit the alpha channel,

   ```
   LF = LFConvertToInt( LF, 'uint8' );
   LFWriteESLF( LF, 'Jacaranda.eslf.png' );
   ```

   the file occupies 102 MBytes.

2. After converting the light field to 8-bit format as above, **Run LFWriteESLF** to save a ligt field as a JPEG-compressed ESLF:

   ```
   LFWriteESLF( LF, 'Jacaranda.eslf.jpg' );
   ```

   The resulting 8-bit lossily compressed light field without a weight channel occupies only 16 MBytes, but upon loading and displaying it you may notice strong compression artefacts. Adjust the JPEG compression rate to increase quality

   ```
   LFWriteESLF( LF, 'Jacaranda.eslf.jpg', [], 'Quality', 95 );
   ```

   looks significantly better and occupies 38 MBytes.

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. 2).

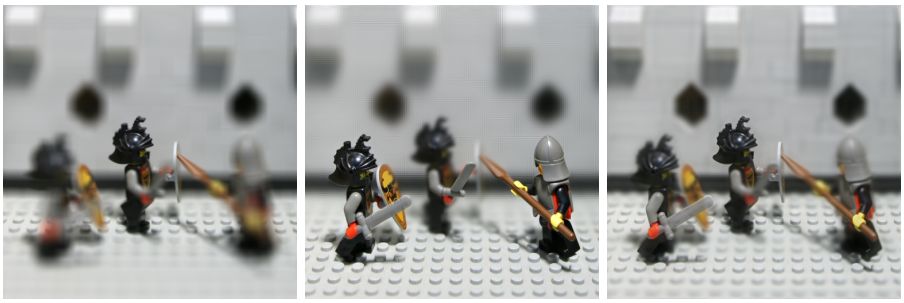

Figure 6: Three examples of filtering gantry imagery: the shift-and-sum filter performing planar focus (left), the hyperfan filter performing volumetric focus (center), and the max between two hyperfan filters, focusing simultaneously on two planes (right).

3.5 Basic Filters

The toolbox comes with a spatial shift-and-sum filter for planar focus and refocus super-resolution, 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 gantry light fields. This should be done from the top of the Stanford gantry light fields folder, after downloading and unzipping the samples following the instructions in Sect. 3.3. 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 7: Examples of filtering Lytro Illum imagery, showing the input (left) and shift-and-sum filter performing planar focus (right).

3.5.1 Linear Refocus Super-Resolution

New in v0.5 is linear refocus super-resolution. This is a way to simultaneously focus at a single depth and boost the resolution of the result. The method modifies LFFiltShiftSum to upsample each image prior to shifting, effectively allowing fractional shifts, and boosting resolution in the merged image.

Note that this approach only adds information for fractional slopes, images focused at integer slopes will see no benefit. Note also that the upsampling ratio does not reflect the effective resolution increase: upsampling by $10\times$ might only increase effective resolution by $1.5 - 2\times$, for example.

Typical results are shown in Fig. 8, for an upsampling rate of $10\times$. These are tight crops on the Flowers sample ESLF and `IMG_6201.eslf.png` from the Flowers category of the Stanford Multi-View Light Field dataset at `http://lightfields.stanford.edu`.

From the figure, we clearly see a dramatic improvement in the visual appearance of the 'Super-res' result compared with the 'Focus' result. Comparing with '2D Interp' cubic interpolation, we see a less dramatic but still noticeable improvement more representative of the effective resolution increase offered by this approach. Better camera calibration and light field rectification will increase the effectiveness of this form of super-resolution.

For a demonstration of linear refocus super-resolution, see LFDemoRefocusSuperres.

3.6 Running the Small Sample Calibration

3.6.1 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

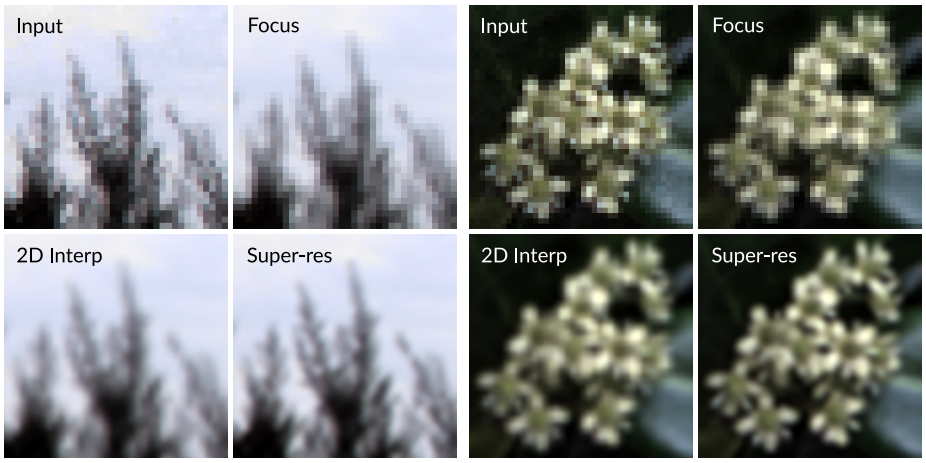

Figure 8: Linear super-resolution using LFFiltShiftSum. The inputs are small crops from larger light fields; 'Focus' shows focus alone, mostly removing noise and limiting depth of field; '2D Interp' shows 2D cubic interpolation applied to the focus image, for comparison; and 'Super-res' shows linear super-resolution with noticeably finer detail than focus alone or 2D interpolation. See LFDemoRefocusSuperres for usage.

- 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://dgd.vision/Tools/LFToolbox`. Good results have been obtained using a 19×19 grid with a 3.6 mm spacing, with at least ten diverse poses.

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/');
```

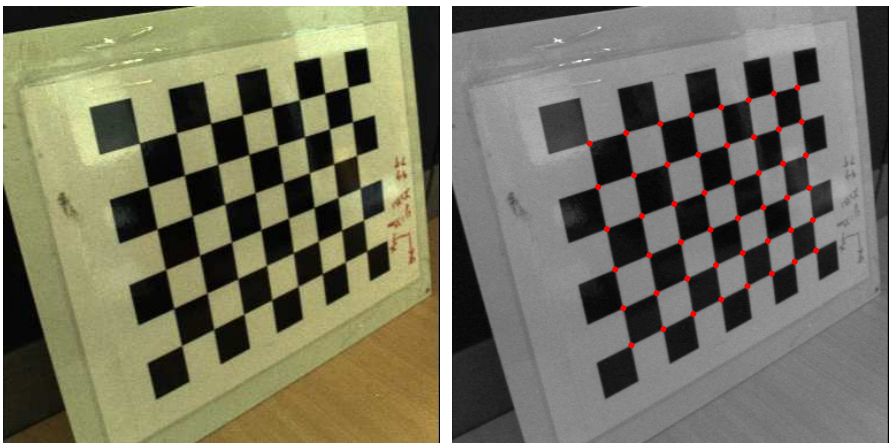

Figure 9: Left: Example of a decoded checkerboard image – no colour correction is necessary and rectification should not be applied; Right: Example of checkerboard corners automatically fit to the checkerboard in the first step of a calibration.

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. 9.

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. 10. 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 text 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.

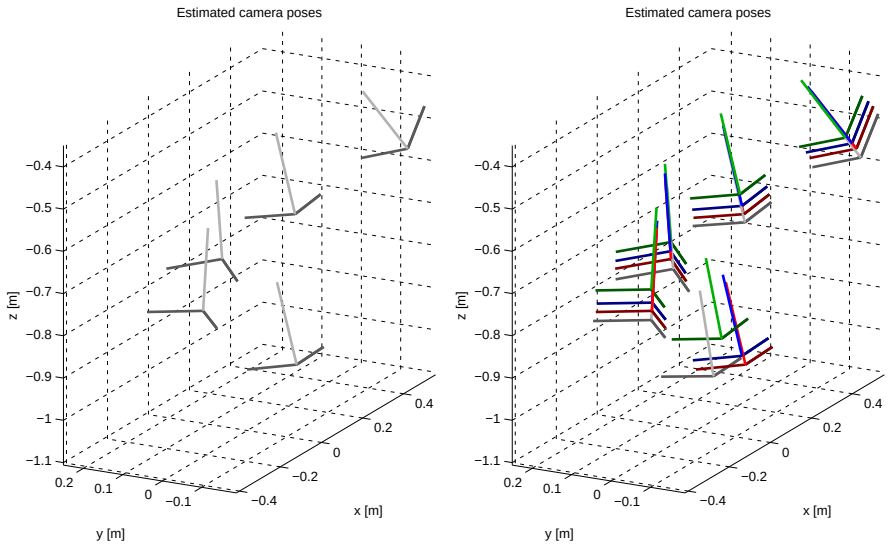

Figure 10: The estimated camera pose display; Left: After parameter initialization, and Right: After completion of a calibration; Gray: initial estimate, Green: optimized without distortion, Blue: optimized with distortion, and Red: after refinement.

3.6.2 Validating

One way of 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.
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 text 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. When calibrating with your own camera, rectification should ideally show good results.

3.6.3 Cleaning Up and Validating

As discussed earlier in this section, the `CalInfo.json` generated from the small sample calibration is not very good. When finished with the sample calibration, you should remove this file from `Cameras/A000424242/PlenCalSmallExample` and re-run LFUtilProcessCalibrations. Repeating the above validation procedure with the better, default sample calibration in place yields more reasonable validation results, such as those shown in Fig. 11, even despite a slight mismatch in camera parameters.

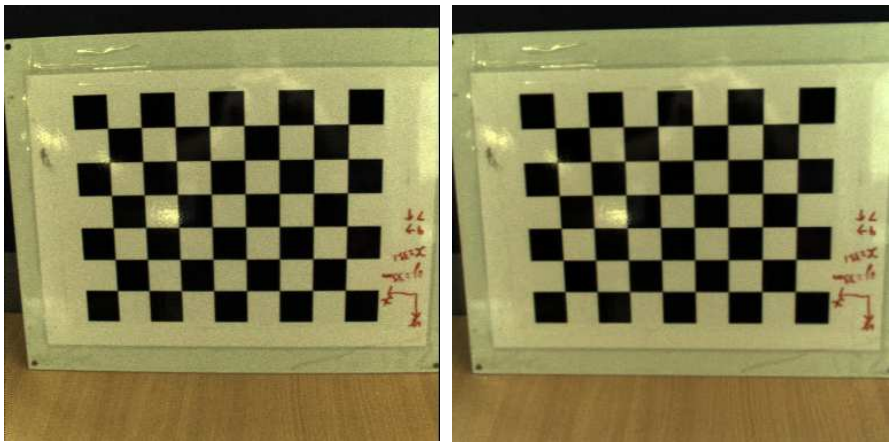

Figure 11: A rectified checkerboard; Run Fig. 4. More complete datasets are explored in [1].

3.7 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 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:

`Samples`	Top level of samples
`Cameras`	Stores info for one or more cameras
`A123412123`	**Your camera's top level folder**
`WhiteImages`	**Your camera's white images**
`A000424242`	The camera used to measure the samples
`CalZoomedOutFixedFoc`	A single calibration result
`WhiteImages`	White images for the sample camera
`Images`	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`.

3.8 Calibrating the Illum

Calibration for the Lytro Illum remains experimental, and the toolbox can struggle for certain focal lengths. Samples near the edges of lenslets are especially challenging, and short focal lengths (with the camera zoomed out) are also more challenging. To make things easier for calibration, try calibrating with the lens zoomed in somewhat, and increasing `CalOptions.LensletBorderSize` to 2 or 3 pixels. Future toolbox versions will address this shortcoming.

4 Decoding in Detail

The toolbox decodes lenslet-based light field images into a 4D light field structure following the process described in [1]. At its core, the inputs to this process are a white image and a lenslet image.

The white image is a flat-field image that reveals the vignetting pattern of the camera, the darkening near the edges of images and lenslets. White images are measured by taking an image of an evenly-illuminated white surface, by taking an image through a diffuser, or by using a specialized tool like an internally illuminated integrating sphere. The toolbox uses these images to correct for vignetting 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 automates 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. The white image and raw lenslet image are passed to a decoding function which builds the 4D light field. The function LFSelectFromDatabase selects the appropriate white image for a light field, and is used 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.

4.1 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 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, change the `WhiteImageDatabasePath` variable in LFUtilProcessWhiteImages to point to your white images folder, or use the function argument `FileOptions.WhiteImageDatabasePath`.

In the F01 camera, the white images come in two exposure levels. Only the brighter of the two is used by this toolbox. The set of white images also generally includes some very dark images. These are not ignored by the toolbox.

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.

4.2 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.

4.3 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). When it is clear that discrete indices are being discussed, these

are often referred to using their continuous-domain equivalents, s, t for i, j, and u, v for k, l.

The Lytro F01'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 Illum has more lenslets and more pixels per lenslet. The actual number of samples in a light field depends on how the lenslet grid is aligned with the sensor, and can vary by a few samples between cameras.

Examining the output of LFDecodeLensletImageSimple, we see that it yields a light field `LF` which is a 5D array of size around $9 \times 9 \times 380 \times 380 \times 3$. As discussed in Sect. 2, **the indexing order for `LF` is** j, i, l, k, c, where c is the RGB colour channel.

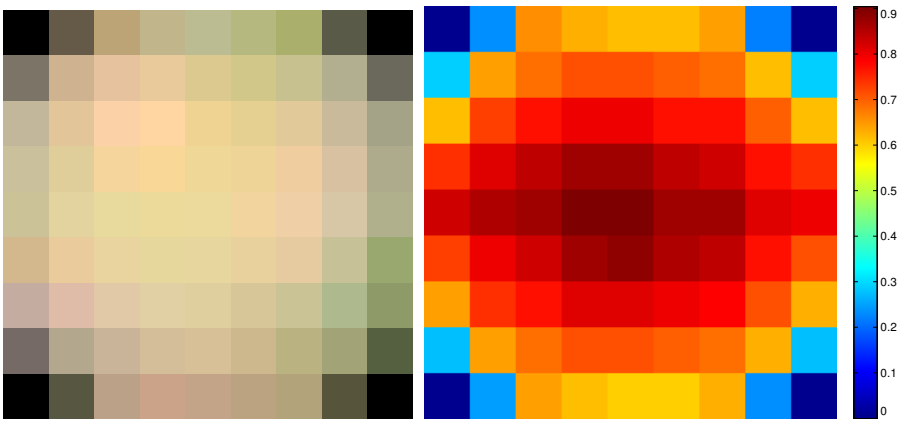

Figure 12: Light field for Sample 1 and its associated weight channel viewed in the i and j dimensions.

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 similar to the output of `LFDisp(LF)`. To examine a slice through the i and j dimensions, you might use the command `imshow(squeeze(LF(:,:,380/2,380/2,:)))`, with an output similar to that shown in Fig. 12. Notice this shows the shape of the image under a lenslet, with darkened corner pixels that contain little or no information. See Sect. 3.2 for more ways of displaying light fields.

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. 12. The weight channel is useful in filtering applications which accept a weighting term.

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.

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

LFUtilCalLensletCam 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. 9 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. Intrinsics and poses are optimized, and the results are saved to `CalInfo.json`. The pose estimate display is updated with the new pose estimates. The text 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 text 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 $\boldsymbol{H}$ relating a pixel index $\boldsymbol{n} = [i, j, k, l, 1]^\mathsf{T}$ to an undistorted ray $\boldsymbol{\phi}^u = [s, t, u, v, 1]^\mathsf{T}$, following $\boldsymbol{\phi}^u = \boldsymbol{H}\boldsymbol{n}$. As described in Sect. 2, the intrinsic model follows a relative two-plane parameterization with plane separation $D = 1$m.
- **Distortion parameters**: describe radial distortion in ray *direction*, employing the small angle assumption such that $\boldsymbol{\theta} = [\theta_1, \theta_2] \approx [dx/dz, dy/dz]$ for each ray. The five distortion parameters are $\boldsymbol{b} = [b_s, b_t]$ and $\boldsymbol{k} = [k_{1..3}]$, where $\boldsymbol{b}$ captures decentering and $\boldsymbol{k}$ are radial distortion coefficients. The complete distortion vector is in the order $\boldsymbol{d} = [\boldsymbol{b}, \boldsymbol{k}]$. If $\boldsymbol{\theta}^u$ and $\boldsymbol{\theta}^d$ are the undistorted and distorted

2D ray directions, respectively, then
$\boldsymbol{\theta}^d = (1 + k_1 r^2 + k_2 r^4 + \cdots)\left(\boldsymbol{\theta}^u - \boldsymbol{b}\right) + \boldsymbol{b}, \quad r = \sqrt{\theta_s^2 + \theta_t^2}$.

Toolbox v0.4 and earlier incorrectly employed as the third polynomial term $k_3 r^8$. As of v0.5 this has been corrected to $k_3 r^6$.

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 $\boldsymbol{\phi} = \boldsymbol{H}\boldsymbol{n}$. 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 $p = [0.0015, 0.0015, -0.34, -0.34, 1]^\mathsf{T}$. Similarly, $n = [5, 5, 190.5, 190.5, 1]'$ yields the ray $p = [0, 0, 0, 0, 1]^\mathsf{T}$, 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.

LFCalDispRectIntrinsics is a helper function for building this matrix. 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. 13.

If we wanted to sample closer to the horizontal u edges of this light field and work with non-square pixels, we could increase $\boldsymbol{H}(3,3)$, 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. 13.

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. 14 shows the result of applying the example rectifications from Fig. 13 – 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.

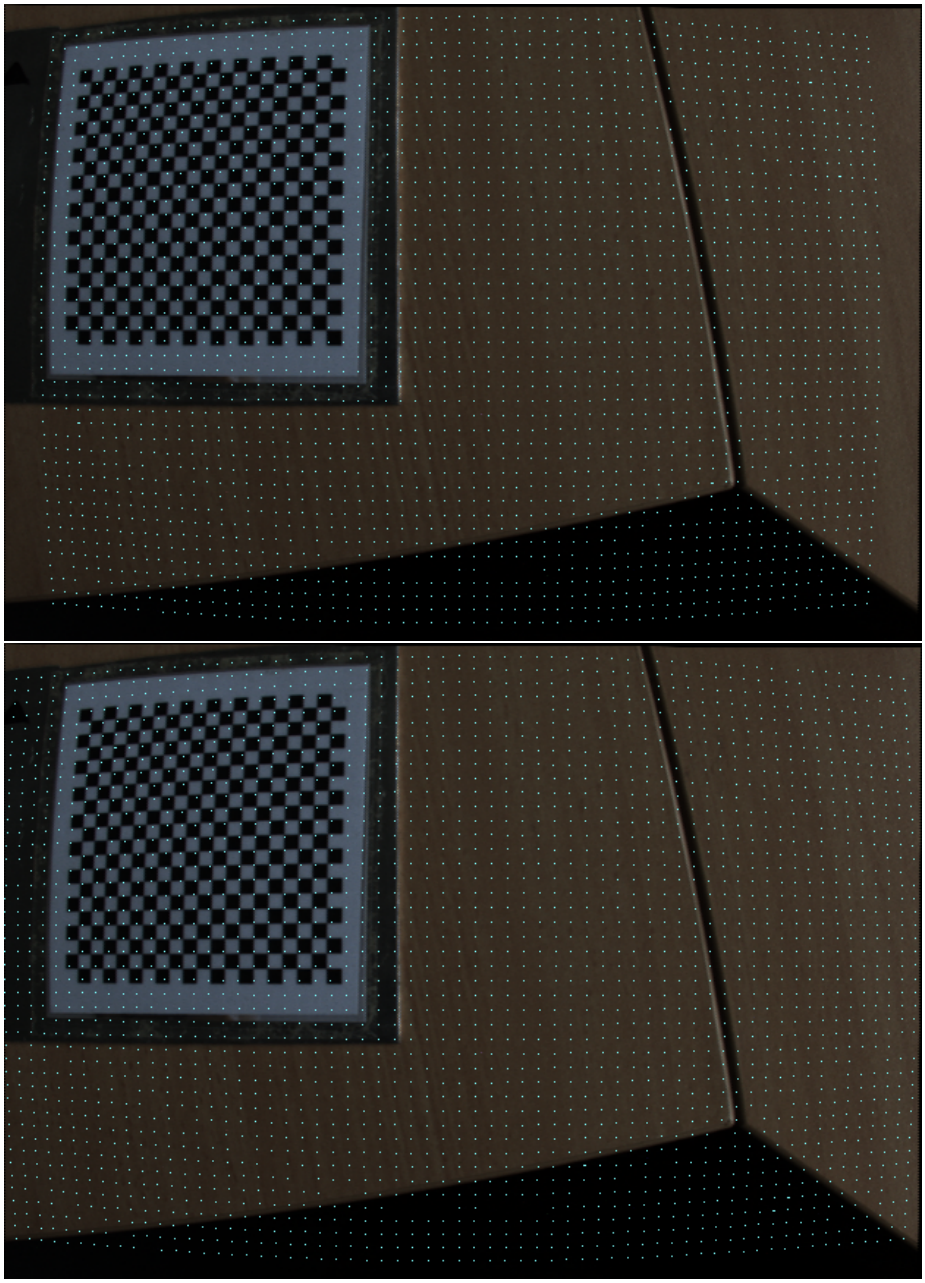

Figure 13: The default and adjusted sampling patterns. Here the sampling pattern has been stretched horizontally, incorporating more of the measured image, but yielding rectangular pixels. The following figure shows the result of applying each of these.

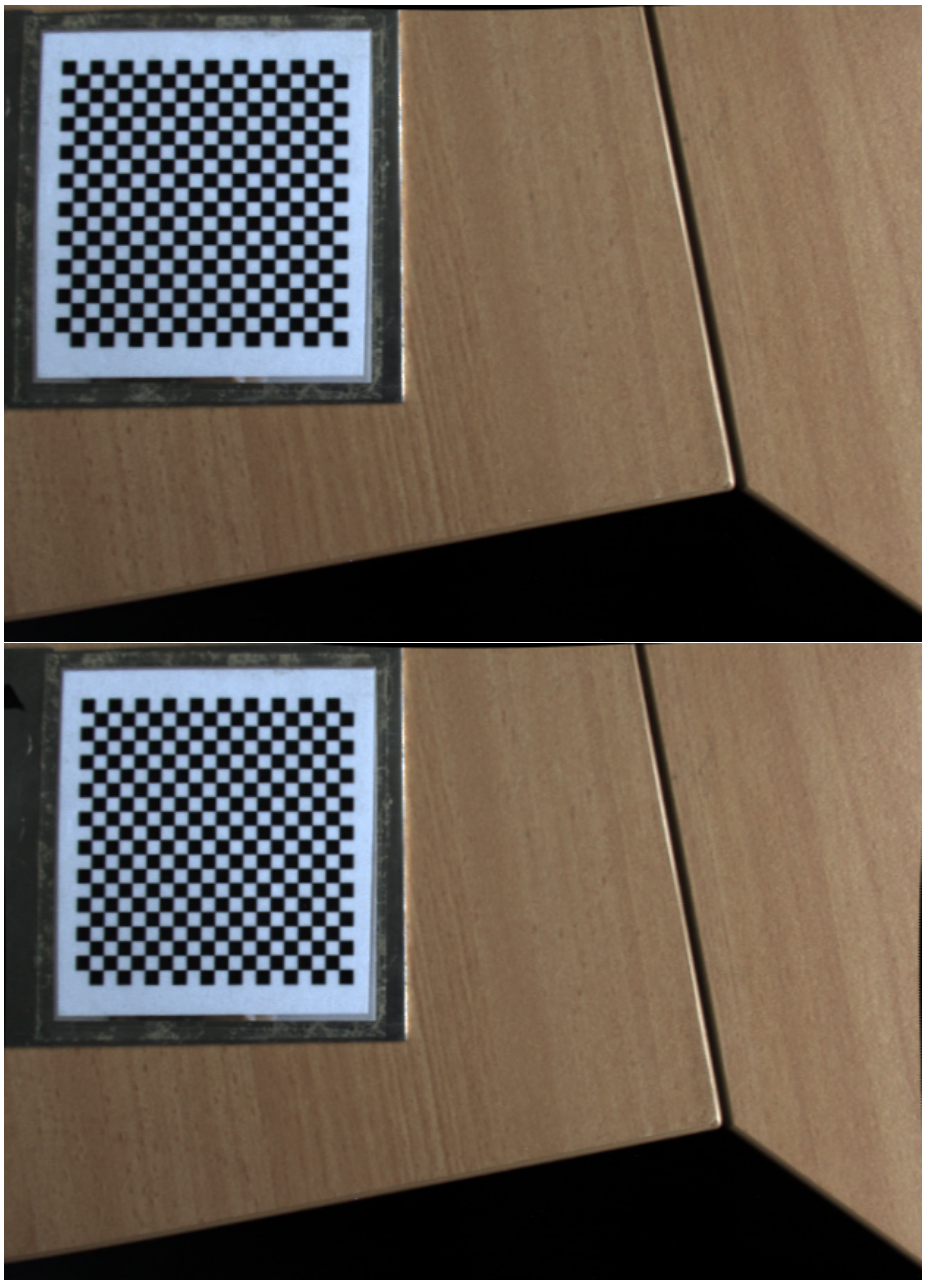

Figure 14: Rectification applied with the default and adjusted sampling patterns described in the previous figure.

Appendix A Working with Lytro Files

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 (on Mac OS 10.10.2 running Lytro Desktop 4.1.2)

```
/Users/<username>/Library/Application Support/Lytro/cameras/sn-<serial_number>
```

note that for previous versions of the MAC software these files were stored at

```
/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

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 extracts thumbnails and saves them to disk.

Appendix B Upgrading from 0.4

There have been minor performance improvements to how white images are analyzed to build grid files, and to how calibration distortion parameters get applied.

For those not using calibration or rectification, no action is required. Re-doing white image analysis will result in a small improvement in decoded light field quality.

Users employing calibration / rectification should not use previously generated calibration files with the new toolbox, and should reprocess their calibrations.

B.1 Reprocessing White Images

LFUtilProcessWhiteImages fits a grid to each of the white images in the `Cameras` folder. To force it to redo this process, pass in a `FileOptions` parameter with the `ForceRedo` field set to true. Alternatively, delete all the `.grid.mat` files and the `WhiteImageDatabase.mat` file in the Cameras folder, then run white image analysis.

After new grid models are generated, every future image decode will benefit from the improved grid fit.

B.2 Reprocessing Calibrations

It is not recommended to rectify files using calibrations built using earlier toolbox versions, as interpretation of distortion parameters has changed.

First, reprocess white images as above, and re-decode calibration files, to benefit from the improved grid fit. Then, remove `CalibrationDatabase.mat`, and either use function parameters to force the calibration to redo previously complete steps, or delete generated files and re-run calibration. The relevant parameters to LFUtilCalLensletCam are `CalOptions.ForceRedoCornerFinding` and `CalOptions.ForceRedoInit`. Or, if removing files, the relevant files to remove are `CheckerboardCorners.mat`, `CalInfo.json`, and `*__CheckerCorners.mat`.

Once complete, run LFUtilProcessCalibrations.

Appendix C Function Reference

This is a partial list of top-level functions organized by task. See also

```
help <LFToolbox top folder name>
```

for a more complete list. Refer to the documentation in each function for further information. The SupportFunctions folder contains additional functions used internally by the toolbox.

Decoding

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.

LFUtilDecodeLytroFolder

Utility for decoding, colour correcting and rectifying Lytro imagery. Can process multiple light fields; recursively searches folder structures; accepts filename specifications including wildcards. 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 11 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 detected and skipped.

Filtering

LFBuild2DFreqFan

Construct a 2D fan filter in the frequency domain. Apply this filter with LFFilt2DFFT.

LFBuild2DFreqLine

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

Construct a 4D dual-fan filter in the frequency domain. Apply this filter with LFFilt4DFFT.

LFBuild4DFreqHypercone

Construct a 4D hypercone filter in the frequency domain. Apply this filter with LFFilt4DFFT.

LFBuild4DFreqHyperfan

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

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

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

LFDemoBasicFiltIllum

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

LFDemoBasicFiltGantry

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

LFDemoRefocusSuperres NEW

Demonstrates linear refocus-based super-resolution, see Sect. 3.5.1.

LFFilt2DFFT

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

LFFilt4DFFT

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

LFFiltShiftSum

The shift sum filter is a spatial-domain depth-selective filter, with an effect similar to planar focus. NEW: Use the UpsampRate parameter for linear refocus super-resolution.

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 stretching. 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

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 and 2D images. Can be nested as demonstrated in Sect. 3.2.

LFDispLawnmower NEW

Similar to LFDispVidCirc, but visits every light field view in sequence following horizontal and vertical lawnmower patterns.

LFDispMousePan

Interactively display 2D slices of the light field with a 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 lenslet-based light fields mean that the effect is best near the center of the spatial range.

LFDispProj NEW

Displays the light field by projecting it onto a plane.

LFDispProjSubfigs NEW

Uses LFDispProj to display six different projections of the light field.

LFDispTiles NEW

Displays the light field as a 2D tiling of 2D images.

LFDispTilesSubfigs NEW

Uses LFDispTiles to display six different tilings of the light field.

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.

LFReadESLF NEW

Reads ESLF light field images, as generated by the Lytro Power Tools.

LFReadGantryArray

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

LFReadLFP

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

LFReadMetadata

Reads `json` files.

LFReadRaw

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

LFWriteESLF NEW

Writes ESLF light field images.

LFWriteMetadata

Writes `json` files.

Utility / Convenience

LFConvertToInt NEW

Converts to `uint8` or `uint16`, with automatic scaling.

LFConvertToFloat

Converts to `single` or `double`, with automatic scaling.

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. 1.

LFUtilUnpackLytroArchive

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

LFUtilExtractLFPThumbs

Extracts thumbnails from LFP files and writes them to disk.