TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

# oLaF: A flexible 3D reconstruction framework for light field microscopy

Anca Stefanoiu, Josue Page, Tobias Lasser

TUMI-1978

# **oLaF**: A flexible 3D reconstruction framework for light field microscopy

v3.0 released 19-May-2020

## Overview

oLaF is a flexible Matlab framework for 3D reconstruction of light field microscopy data. It is designed to cope with various LFM configurations in terms of MLA type (regular vs. hexagonal grid, single-focus vs. mixed multi-focus lenslets) and MLA placement in the optical path (original 1.0 vs. defocused 2.0 LFM vs. Fourier LFM designs).

oLaF can be found at https://gitlab.lrz.de/IP/olaf.

## Acknowledgments

oLaF evolved gradually improving and expanding on the functionality in [2]. Where possible, the naming conventions were kept for the sake of relatability and convenience of the shared users. The pre-processing of the raw light field images relies on the image rectification functionality [1] in the Light Field Toolbox for Matlab by Donald G. Dansereau.

## Citing

When using **oLaF**, please reference the following citations:

- A. Stefanoiu, J. Page, P. Symvoulidis, G. G. Westmeyer, and T. Lasser. Artifact-free deconvolution in light field microscopy. *Optics Express*, 27(22):31644, 2019.
- A. Stefanoiu, G. Scrofani, G. Saavedra, M. Martínez-Corral, and T. Lasser. What about computational super-resolution in fluorescence Fourier light field microscopy? *Optics Express*, 28(11):16554, 2020.
- A. Stefanoiu, G. Scrofani, G. Saavedra, M. Martínez-Corral, and T. Lasser. Deconvolution in Fourier integral microscopy. *Proc. SPIE 11396, Computational Imaging V*, 2020.

## Feedback

Feedback and suggestions are always welcome.

anca.stefanoiu(at)tum(dot)de / josue.page(at)tum(dot)de / lasser(at)in(dot)tum(dot).de

## References

[1] D. G. Dansereau, O. Pizarro, and S. B. Williams. Decoding, Calibration and Rectification for Lenselet-Based Plenoptic Cameras. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1027–1034. IEEE, jun 2013.

[2] R. Prevedel, Y. G. Yoon, M. Hoffmann, N. Pak, G. Wetzstein, S. Kato, T. Schrödel, R. Raskar, M. Zimmer, E. S. Boyden, and A. Vaziri. Simultaneous whole-animal 3D imaging of neuronal activity using light-field microscopy. *Nature Methods*, 11(7):727–730, jul 2014.

[3] A. Stefanoiu, J. Page, P. Symvoulidis, G. G. Westmeyer, and T. Lasser. Artifact-free deconvolution in light field microscopy. *Optics Express*, 27(22):31644–31666, 2019.

[4] A. Stefanoiu, G. Scrofani, G. Saavedra, M. Martínez-Corral, and T. Lasser. Deconvolution in Fourier integral microscopy. In *Computational Imaging V*, volume 11396, page 18. SPIE, may 2020.

[5] A. Stefanoiu, G. Scrofani, G. Saavedra, M. Martínez-Corral, and T. Lasser. What about computational super-resolution in fluorescence Fourier light field microscopy? *Optics Express*, 28(11):16554, 2020.

## Contents

# 1 Getting Started

Clone or download *oLaF* at https://gitlab.lrz.de/IP/olaf into a location of choice and run the script `olaf/Code/import2ws.m` to set up the Matlab path. Every time Matlab restarts, this script needs to be re-run. Alternatively, add it to `startup.m`.

## 1.1 LFM vs. FLFM

*oLaF* deals with 3D reconstruction of data from both conventional light field microscopy (LFM) and Fourier light field microscopy (FLFM). While both modalities make use of a micro-lens arrays to record the spatio-angular light field information, they are in many ways conceptually different. In this sense, *oLaF* distinguishes between LFM and FLFM functionality and in the following we will demonstrate each of them separately. To make the distinction clear, we employ the *LFM* prefix for the LFM related function names and *FLFM* for the FLFM related ones; see section 5.

## 1.2 Sample Datasets

`olaf/SampleData/` contains several example light field datasets, acquired with various conventional LFM and FLFM setups. Every dataset contains a raw light field image, a white image (in case of LFM) or a calibration image (in case of FLFM) used for detecting the micro-lens centers and a YAML file describing the microscope configuration parameters. Fig. 1 displays three such datasets.

## 1.3 Step-by-step reconstruction of conventional LFM data

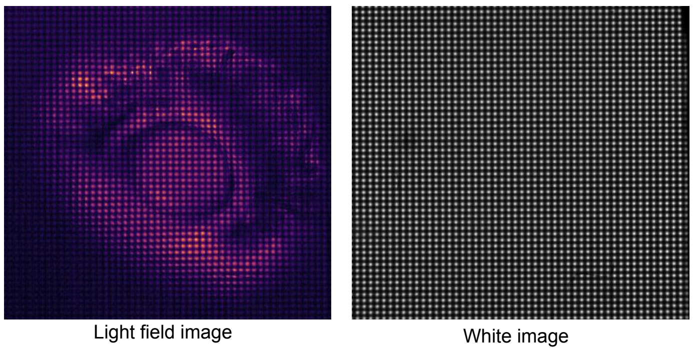
`olaf/Code/mainLFM.m` serves as a step-by-step demo script of the LFM related functionality in oLaF v3.0.

You can switch between datasets by un-commenting the associated lines at the beginning of the script. Different `depthRange` and `depthStep` are suggested for different datasets in order to keep a low runtime for demonstration purposes. The impact of `depthRange` on the runtime is discussed in Sec. 4.1.

### 1.3.1 Loading datasets

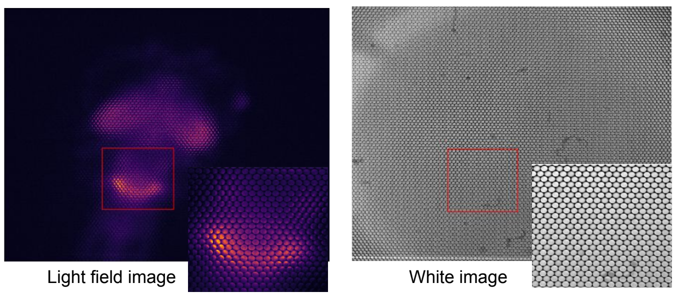
`[LensletImage, WhiteImage, configFile] = LFM_selectLFImages(dataset)` is a convenience function for loading the light field image together with the corresponding white image and a configuration file containing the acquisition specific parameters, for a given dataset. Note, the raw light field images are cropped to a region of interest (ROI), as depicted in Fig. 1 (b), in order to speed up the computations. For the provided datasets the ROIs are predefined. However, you can choose custom regions using the built-in Matlab interactive function `getrect()`:

```
figure; imagesc(LensletImage);
rect = round(getrect);
```



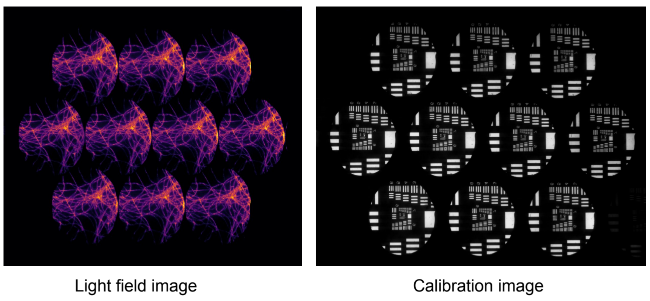
LFConfig.yaml
```
gridType: 'reg'
focus: 'single'
plenoptic: 1
uLensMask: 1
M: 20
NA: 0.5000
ftl: 165000
fm: 3000
tube2mla: 165000
mla2sensor: 3000
lensPitch: 150
pixelPitch: 6.5000
WaveLength: 0.5200
n: 1
```

(a) Zebrafish eye: Single focal length, regular grid micro-lens array; original LFM setup.



LFConfig.yaml
```
gridType: 'hex'
focus: 'multi'
plenoptic: 2
uLensMask: 0
M: 28.0
NA: 0.5000
ftl: 200000.0
fm: [4250.0, 4950.0, 5900.0]
tube2mla: 190000.0
mla2sensor: 3400.0
lensPitch: 127.0
pixelPitch: 5.5000
WaveLength: 0.5200
n: 1
```

(b) Zebrafish brain: Multi focus, hexagonal grid micro-lens array; defocused LFM setup.



LFConfig.yaml
```
gridType: 'hex'
NA: 0.4
fobj: 9000
f1: 50000
f2: 40000
fm: 6500
mla2sensor: 6500
lensPitch: 1000
pixelPitch: 2.2
WaveLength: 0.5200
n: 1
noLensHoriz: 5
noLensVert: 4
shiftRow: 1
spacingPixels: 450
horizOffset: 325
vertOffset: 340
gridRot: 0
```

(c) Cotton fibers: Hexagonal grid micro-lens array; FLFM setup.

Figure 1: Example light field data sets.

### 1.3.2 User Inputs.

When reconstructing a light field image, the user decides the resolution related parameters:

- `depthRange` is the axial range (in $\mu m$) of the reconstructed volume, relative to the focal plane of the objective lens, e.g., `depthRange = [-20,20];`
- `depthStep` represents the axial resolution (in $\mu m$) of the reconstructed object.
- `newSpacingPx` is the desired spacing (in number of pixels) between neighboring lenslet centers. This parameter was introduced to allow for up-/down-sampling of the raw lenslet image. `newSpacingPx = 'default'` means no up-/down-sampling i.e., `newSpacingPx = spacingPx`. `spacingPx` is the real lenslet spacing given by the micro-lens pitch and

the camera sensor pixel pitch. In other words, `newSpacingPx` controls the sensor resolution and the lenslet image is interpolated to this user-specified pixel spacing, prior to reconstruction.

- `superResFactor` controls the lateral resolution of the reconstructed object. It is interpreted as a multiple of the lenslet resolution (1 voxel/lenslet). `superResFactor = 'default'` means the object is reconstructed at sensor resolution, while `superResFactor = 1` means lenslet resolution.

### 1.3.3 Light field microscope (LFM) setup descriptor.

`Camera = LFM_setCameraParams(configFile, newSpacingPx)` builds the `Camera` structure based on the fields in the `configFile`. A configuration file (see Fig. 1 (a) and (b)) typically contains the following LFM setup specific parameters:

- `gridType` - micro-lens array grid type. `gridType = 'reg'` for regular grid array; `gridType = 'hex'` for hexagonal grid array.
- `focus` - micro-lens array focus. `focus = 'single'` when all the micro-lens in the array have the same focal length; `focus = 'multi'` when the array contains three mixed focal lengths in a hexagonal grid; see Fig. 6.
- `fm` - focal length of the micro-lens; when `focus = 'multi'`, `fm` is an array (see Fig. 1 (b)).
- `plenoptic` - flag for LFM design. `plenoptic = 1` for the original LFM design (`tube2mla = ftl`); `plenoptic = 2` for defocused LFM (`tube2mla` $\neq$ `ftl`) setups.
- `uLensMask` - micro-lens shape mask. `uLensMask = 1` when there is no space between micro-lens (e.g., regular grid array with square aperture micro-lens). `uLensMask = 0` when there is space between micro-lens e.g., circular aperture.
- `M` - the objective magnification.
- `NA` - objective numerical aperture.
- `ftl` - focal length of the tube lens (in $\mu m$).
- `lensPitch` - micro-lens pitch (in $\mu m$).
- `pixelPitch` - sensor pixel pitch (in $\mu m$).
- `tube2MLA` - distance between the tube lens and MLA ( `tube2MLA = ftl` in original LFM design, i.e. when `plenoptic = 1`). If `tube2MLA` is not known from the acquisition (when measuring was not possible), set the `tube2MLA` field in the YAML configuration file to '0'. Then `tube2mla = computeTube2MLA(lensPitch, mla2sensor, deltaOT, objRad, ftl)` retrieves `tube2MLA` distance such that the effective image-side NA of the tube lens matches the effective NA of the micro-lenses i.e., the micro-images optimally fill the sensor without overlapping. Here, `deltaOT` is the distance between the objective and the tube lens, usually `deltaOT = ftl + fobj` for commercial microscopes.

- `mla2sensor` - distance between the MLA and sensor. Again, set `mla2sensor` to `'0'` in the configuration file if `tube2MLA` is known and `mla2sensor` has to be retrieved such that the F-number matching condition is satisfied.
- `wavelength` - wavelength of the emission light.
- `n` - refraction index (1 for air).

Based on the input parameters and the ones in the configuration file, the `Camera = LFM_setCameraParams (configFile, newSpacingPx)` function computes extra parameters relevant for the reconstruction; these are described in Sec. 2.1.

### 1.3.4 Retrieve lenslet centers and related data structures.

For every image/dataset to be reconstructed, an associated white image needs to be provided in order to detect the lenslet centers; Fig. 1 (a) and (b) show example light field (lenslet) and white images.

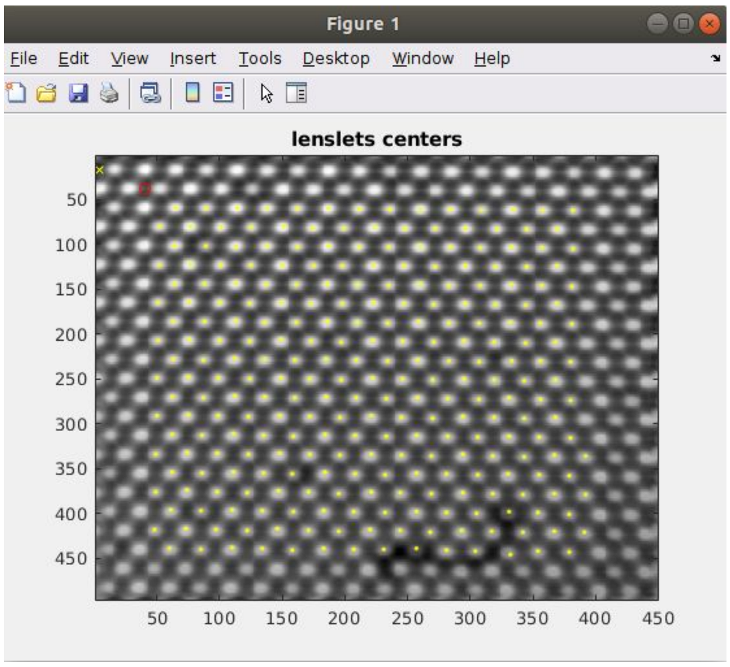


Figure 2: Detected micro-image centers in a white image.

Function `[LensletCenters, Resolution, LensletGridModel, NewLensletGridModel] = LFM_computeGeometryParameters(Camera, WhiteImage, depthRange, depthStep, superResFactor, DebugBuildGridModel, imgSize)` uses such a white image, `WhiteImage`, together with the `Camera` structure and the user inputs to retrieve the lenslet centers and builds several data structures relevant for the 3D reconstruction process. Details on the implementation of this function and the returned data structures are given in Sec. 2.1.

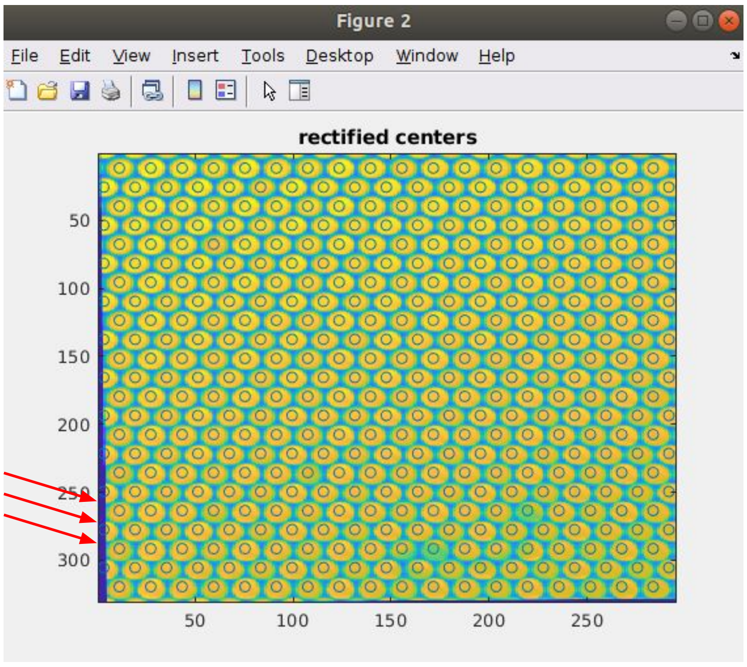


Figure 3: Rectified micro-lens centers after the `NewLensletGridModel` matching transformation.

### 1.3.5 Compute light field point spread function.

Function `[H, Ht] = LFM_computeLFMatrixOperators(Camera, Resolution, LensletCenters)` uses the data structures introduced in the previous section to pre-compute the forward (`H`) and backward (`Ht`) light transport patterns. `H` describes the discrete light field point spread function (LFPSF), and `Ht` describes the inverse light propagation.

Sec. 3.1 describes in detail the functionality implemented in `LFM_computeLFMatrixOperators`.

### 1.3.6 Correct/rectify light field image.

Prior to the reconstruction, the input light field image is transformed to match the `NewLensletGridModel`. For this purpose, we retrieve the 2D affine transformation between the original and new (user defined) grids, `LensletGridModel` and `NewLensletGridModel`:

- `FixAll = LFM_retrieveTransformation(LensletGridModel, NewLensletGridModel)`

and apply this transformation to the light field and white images:

- `[CorrectedLensletImage, CorrectedWhiteImage] = LFM_applyTransformation(LensletImage, WhiteImage, FixAll, LensletCenters, debug)`
  % When `debug = '1'`, the transformed white image (with overlaid rectified centers) is displayed for a visual check like in Fig. 3.

While transforming both images, due to the rotation without cropping, the rectified images show some zero borders, as pointed out by the red arrows in Fig. 3. In order to even out these parts, for smooth visualization of the subsequent reconstruction result, we apply a pre-processing step:

- `correctedLensletImage(correctedLensletImage < mean(correctedLensletImage(:))) = mean(correctedLensletImage(:))`

however, this is optional, and other strategies can be used.

### 1.3.7 Set forward/backward projection operators.

The forward projection operators are functions which use the forward projection patterns (`H`) to simulate a light field image from a 3D volume (object), and the backward projection operators are meant to do the inverse mapping, i.e. generate the 3D object from a lenslet image by applying the pre-computed back-projection patterns (`Ht`).

Based on the `Camera.focus`, function pointers are set up to be passed to the deconvolution routine.

When `Camera.focus = 'single'`:

- `forwardFUN = @(object) LFM_forwardProject( H, object, LensletCenters, Resolution, imgSize, Camera.range);`
- `backwardFUN = @(projection) LFM_backwardProject(Ht, projection, LensletCenters, Resolution, texSize, Camera.range).`

In case `Camera.focus = 'multi'`, we use projection operators which adapt the functionality to multi-focus MLA setups:

- `forwardFUN = @(object) LFM_forwardProjectMultiFocus( H, object, LensletCenters, Resolution, imgSize, Camera.range);`
- `backwardFUN = @(projection) LFM_backwardProjectMultiFocus(Ht, projection, LensletCenters, Resolution, texSize, Camera.range),`

here `H, Ht` pre-computed contain multi-focus patterns, as discussed in Sec. 3.1.

`imgSize` and `texSize` are pre-computed light field image and volume container sizes; they are different when we reconstruct at a different resolution than sensor resolution.

### 1.3.8 3D reconstruction.

Once we have all the necessary ingredients we proceed to 3D reconstruct the light field image, `correctedLensletImage`.

Function `reconVolume = deconvEMS(forwardFUN, backwardFUN, LFimage, it, initVolume, filterFlag, lanczos2FFT, onesForward, onesBack)` implements the `Estimate-Maximize-Smooth` deconvolution algorithm we presented in [3]. The arguments are described below:

- `forwardFUN` and `backwardFUN` are the function pointers set previously,
- `LFimage = correctedLensletImage`,
- `initVolume = ones([texSize, length(Resolution.depths)]);` % is the initial guess,
- `it` is the number of iterations,
- When `filterFlag = 0`, the `deconvEMS` implements the Richardson-Lucy deconvolution, otherwise, when `filterFlag = 1`, `deconvEMS` implements the depth-dependent aliasing-free deconvolution with smoothing step as described in [3].
- `lanczos2FFT = LFM_buildAntiAliasingFilter([texSize, length(Resolution.depths)], widths, lanczosWindowSize);` % contains the depth-dependent anti-aliasing filter kernels, used in the smoothing step of the EMS algorithm. When `filterFlag = 0`, `lanczos2FFT` argument can be `[]`.
  - `widths = LFM_computeDepthAdaptiveWidth(Camera, Resolution);` % are the ideal filter radii, computed based on the LFM depth-dependent sampling. There is one computed width per object depth.
  - `lanczosWindowSize` % is the size of the Lanczos window used to implement the ideal filters. Typically, in our experiments, `lanczosWindowSize = 2,3`.
- `onesForward` and `onesBackward` are used for background correction during deconvolution, to cope with the illumination and noise effects imbalances in real images:
  - `onesvol = ones(size(initVolume));`
  - `onesForward = forwardFUN(onesvol);`
  - `onesBack = backwardFUN(onesForward);`

  When using the EMS deconvolution (`filterFlag = 1`), the background normalization is not necessary, as the smoothing step keeps the process stable over the iterations.

  For more details on the implementation of the deconvolution, see the corresponding Matlab files. For the theory behind the EMS deconvolution algorithm, check out [3].

## 1.4 Step-by-step reconstruction of Fourier LFM data

`olaf/Code/mainFLFM.m` serves as a step-by-step demo script of the FLFM related functionality in oLaF v3.0 using the example dataset shown in Fig.1 (c).

### 1.4.1 User Inputs.

Once the `LensletImage`, `CalibrationImage` and `configFile` are loaded, the user decides the resolution related parameters:

- `depthRange` is the axial range (in $\mu m$) of the reconstructed volume, relative to the front focal plane of the microscope objective lens, e.g., `depthRange = [-80,80];`

- `depthStep` represents the axial resolution (in $\mu m$) of the reconstructed object. `superResFactor` controls the lateral sampling rate of the reconstructed object. When `superResFactor == 1` the volume is reconstructed at the resolution of the sub-aperture images. Conversely, `superResFactor > 1`, the object is discretized at a sampling rate `superResFactor` times the sensor sampling rate. The `LensletImage` (as well as the `CalibrationImage`) is up-sampled accordingly using the *nearest neighbor* method.

### 1.4.2 Fourier light field microscope (FLFM) setup descriptor.

`[Camera, LensletGridModel] = FLFM_setCameraParams(configFile, superResFactor)` builds the `Camera` and LensletGridModel structures based on the fields in the `configFile`. A configuration file (see Fig. 1 (c)) typically contains the following FLFM setup specific parameters:

- `gridType` - micro-lens array grid type. `gridType = 'reg'` for regular grid array; `gridType = 'hex'` for hexagonal grid array.
- `NA` - numerical aperture of the microscope objective.
- `fobj` - focal length of the microscope objective lens.
- `f1` and `f2` - focal lengths of the two relay lenses.
- `fm` - focal length of the micro-lens.
- `mla2sensor` - distance between the MLA plane and camera sensor plane.
- `lensPitch` - micro-lens pitch (in $\mu m$).
- `pixelPitch` - sensor pixel pitch (in $\mu m$).
- `Wavelength` - wavelength of the emission light.
- `n` - refraction index (1 for air).

The `configFile` also contains parameters describing the micro-lens array. These are usually rough estimates (by analyzing the `LensleImage` of the `CalibrationImage`) and they will be refined later, as discussed in section 2.2.

- `noLensHoriz` and `noLensVert` - the number of micro-lens in the array to match the sensor extent.
- `spacingPixels` - the number of pixels between two horizontally neighboring lenslets.
- `horizOffset` and `vertOffset` - the coordinates of the center of the first whole elemental image in the `LensleImage` of the`CalibrationImage`.
- `shiftRow` - whether odd (`shiftRow = 1`) or even (`shiftRow = 2`) rows are half diameter shifted in case of hexagonal grids.

  `gridRot` - the rotation of the grid with respect to the optical axis.

Based on the input parameters and the ones in the configuration file, the function `[Camera, LensletGridModel] = FLFM_setCameraParams(configFile, superResFactor)` computes extra parameters relevant for the reconstruction; these are described in Sec. 2.2.

### 1.4.3 Retrieve lenslet centers and resolution related parameters.

For every image to be reconstructed, an associated calibration image needs to be provided in order to detect the lenslet centers; Fig. 1 (c) shows an example light field image and a corresponding calibration image. A calibration image is an image of an object (usually, but not necessarily, a resolution target) placed at the front focal plane of the microscope objective, so that the centers of the micro-lens coincide with the centers of the elemental images.

Function `[LensletCenters, Resolution] = FLFM_computeGeometryParameters(CalibrationImage, Camera, LensletGridModel, depthRange, depthStep)` uses such a `CalibrationImage`, together with the `Camera` and `LensletGridModel` structures and the user inputs to retrieve the lenslet centers and compute several resolution related parameters that are relevant for the 3D reconstruction process.

Details on the implementation of this function are given in Sec. 2.2.

### 1.4.4 Compute the light field point spread function.

Function `[H, Ht] = FLFM_computeLFMatrixOperators(Camera, Resolution)` uses the data structures introduced in the previous sections to pre-compute the forward (`H`) and backward (`Ht`) light transport patterns. `H` describes the discretized 3D light field point spread function (LFPSF), and `Ht` is its transpose.

Sec. 3.2 describes the functionality implemented in `FLFM_computeLFMatrixOperators`.

### 1.4.5 3D reconstruction.

The forward projection operator is a function which uses the pre-computed LFPSF (`H`) to simulate a light field image from a 3D volume (object), and the backward projection operator is meant to do the inverse mapping, i.e. generate the 3D object from a lenslet image by applying the pre-computed back-projection model (`Ht`).

We set up function pointers for the projectors to be passed to the deconvolution routine:

- `forwardFUN = @(volume) FLFM_forwardProject(H, volume);`
- `backwardFUN = @(projection) FLFM_backwardProject(Ht, projection);`

Once we have all the necessary ingredients we proceed to deconvolved the light field image, `LensletImage`.

Function `recon = deconvRL(forwardFUN, backwardFUN, LensletImage, iter, init)` implements the `Richardson-Lucy` deconvolution algorithm as presented in [5]. The arguments are described below:

- `forwardFUN` and `backwardFUN` are the function pointers set previously;
- `LensletImage` is the raw light field image we want to 3D reconstruct;
- `iter` is the number of iterations;

- `init` is the initial solution guess; usually a uniform white object (`init = ones(volumeSize)`).

Alternatively, the `recon = deconvOSL(forwardFUN, backwardFUN, LensletImage, iter, init, lambda)` function implements the `One-Step-Late` algorithm which combines the `Richardson-Lucy` deconvolution with a total variation (TV) prior on the data to be reconstructed as presented in [4] The additional argument, `lambda` represents the regularization parameter.

The axial slices of the 3D reconstructed object have the same size as the light field image (in terms of number of pixel), for the ease of use of the convolution operation (see section 4.2). However, the actual field of view of the microscope is smaller (it matches an elemental image size in terms of number of pixels; see section 2.2). We compute the field of view (`fovRangeX` and `fovRangeY`) and crop the `recon` accordingly:

- `reconCropped = recon(fovRangeY, fovRangeX, :)`

## 2 Microscope geometry

### 2.1 LFM setup

Based on the setup description (from user inputs and configuration file), function `Camera = LFM_setCameraParams(configFile, newSpacingPx)` computes extra parameters relevant for the reconstruction:

- `Camera.range` - when `gridType = 'reg'` we exploit the symmetry in the alignment of the discrete volume to the system's optical axis. Then we can pre-compute the light field point spread function (LFPSF) for a reduced (one quarter) set of source point positions. In this case we set `Camera.range = 'quarter'`. When `gridType = 'hex'`, unfortunately, the discretization does not allow for such optimization and we set `Camera.range = 'full'`. For a detailed explanation see Sec. 3.1.
- `Camera.fobj` - focal length of the objective lens. `Camera.fobj = Camera.ftl/Camera.M`.
- `Camera.DeltaOT` - objective to tube lens distance. `Camera.DeltaOT = Camera.ftl + Camera.fobj` for 4-f systems.
- `Camera.spacingPx` - number of pixels behind a micro-lens. `Camera.spacingPx` is computed as `Camera.lensPitch/Camera.pixelPitch`.
- `Camera.newSpacingPx` - as specified by the user. See paragraph 1.3.2.
- `Camera.newPixelPitch` - sensor pixel pitch corresponding to the new micro-lens spacing. It is computed as `Camera.lensPitch/newSpacingPx`.
- `Camera.k` - wave number. `Camera.k` $= 2 * \pi *$ `Camera.n/Camera.WaveLength`.
- `Camera.dof` - an object at a distance `Camera.dof` in front of the objective is focused on the MLA by the tube lens. Fig. 4 depicts `dof` together with all the relevant LFM quantities.

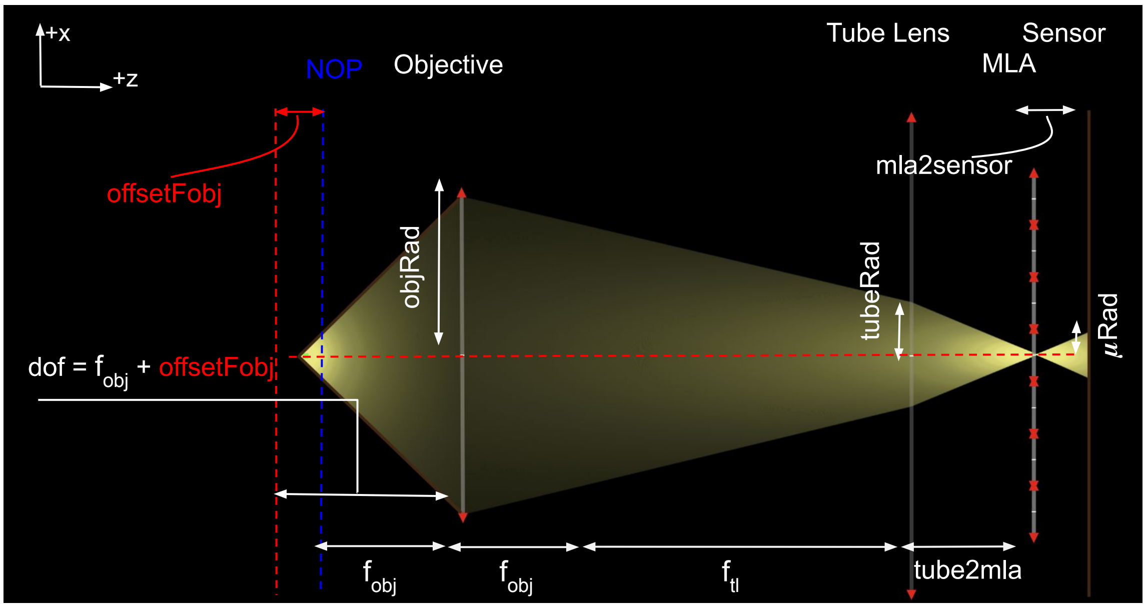


Figure 4: LFM setup specific quantities.

- `Camera.offsetFobj` - the offset from `fobj` to `dof`. `Camera.offsetFobj = Camera.dof - Camera.fobj`.
- `Camera.objRad` - radius of the objective lens. `Camera.objRad = Camera.fobj * Camera.NA`.
- `Camera.uRad` - radius of the micro-image formed on the sensor for an object depth focused on the MLA by the tube lens (`Camera.dof`). $\mu$`Rad` is depicted in Fig. 4.
- `Camera.tubeRad` - effective tube lens radius. It represents the radius of the wave-front distribution incident on the tube lens, for a source point at `dof` - see Fig. 4.

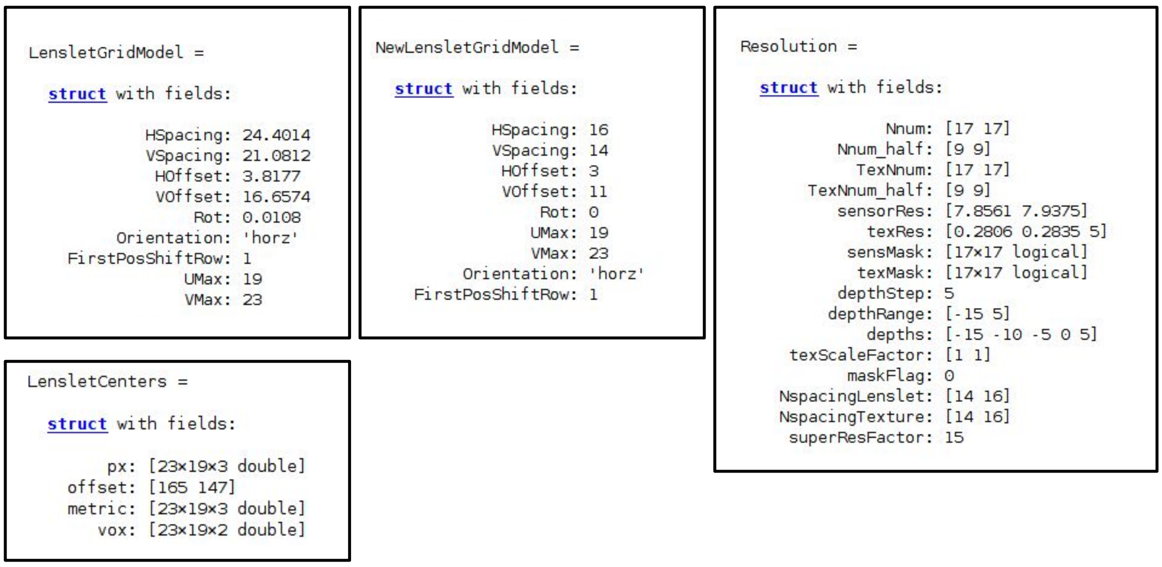
Function `[LensletCenters, Resolution, LensletGridModel, NewLensletGridModel] = LFM_computeGeometryParameters(Camera, WhiteImage, depthRange, depthStep, superResFactor, DebugBuildGridModel, imgSize)` uses a white image, together with the `Camera` structure and the user inputs to retrieve the lenslet centers and to build several data structures relevant for the reconstruction.

`DebugBuildGridModel` is a binary flag. When it is set `true`, the white image with overlaid micro-image centers is displayed for a visual check as in Fig. 2.

`imgSize` is only needed when a white image is not available (`WhiteImage = []`). This is the case in simulations, when a lenslet grid model is built based on the `Camera` specs and the desired sensor image size, `imgSize`. Conversely, when a white image is provided, the `imgSize` argument can be omitted as the sensor image size is just the size of the `WhiteImage`.

Finally, the `LFM_computeGeometryParameters` function returns the following data structures:

- `LensletGridModel` which is derived when analyzing the raw white image. `LensletGridModel` contains information like the spacing between lenslet centers in pixels (`HSpacing` and



```
LensletGridModel =

  struct with fields:

             HSpacing: 24.4014
             VSpacing: 21.0812
              HOffset: 3.8177
              VOffset: 16.6574
                  Rot: 0.0108
          Orientation: 'horz'
     FirstPosShiftRow: 1
                 UMax: 19
                 VMax: 23
```

```
NewLensletGridModel =

  struct with fields:

             HSpacing: 16
             VSpacing: 14
              HOffset: 3
              VOffset: 11
                  Rot: 0
                 UMax: 19
                 VMax: 23
          Orientation: 'horz'
     FirstPosShiftRow: 1
```

```
Resolution =

  struct with fields:

               Nnum: [17 17]
          Nnum_half: [9 9]
            TexNnum: [17 17]
       TexNnum_half: [9 9]
          sensorRes: [7.8561 7.9375]
             texRes: [0.2806 0.2835 5]
           sensMask: [17×17 logical]
            texMask: [17×17 logical]
          depthStep: 5
         depthRange: [-15 5]
             depths: [-15 -10 -5 0 5]
     texScaleFactor: [1 1]
           maskFlag: 0
   NspacingLenslet: [14 16]
   NspacingTexture: [14 16]
     superResFactor: 15
```

```
LensletCenters =

  struct with fields:

      px: [23×19×3 double]
  offset: [165 147]
  metric: [23×19×3 double]
     vox: [23×19×2 double]
```

Figure 5: Lenslet grid and resolution related parameters.

`Vspacing` fields; which are different for hexagonal grids), the offset to the center of the first whole lenslet image in the white image (`HOffset` and `VOffset`), the rotation of the grid with respect to the optical axis (`Rot`), or whether odd or even rows are half diameter shifted in case of hexagonal grids (`FirstPosShiftRow`), etc. An example `LensletGridModel` for a regular grid MLA, with $lensPitch = 127\mu m$ and $pixelPitch = 5.5\mu m$ is displayed in Fig. 5.

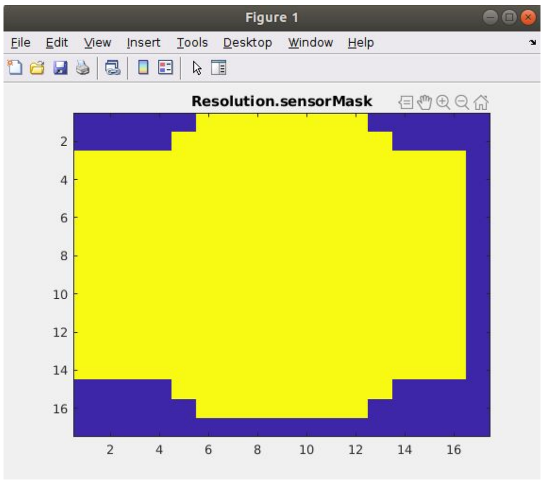
- `NewLensletGridModel` is the ideal (`Rot` $= 0$) grid model created based on the user inputs and the `LensletGridModel`. Fig. 5 (middle) shows a `NewLensletGridModel` for an input `newSpacingPx = 17`. The new pixel pitch is now given by `lensPitch/newSpacingPx`, such that `HSpacing` and `VSpacing` are now integer values in terms of the new pixel size.

- `Resolution` contains all the sensor and object space resolution related quantities. Similarly to the lenslet grid model, we build a `TextureGridModel` which describes the object space. The object space is interpreted as tiled lateral patches (areas) that are imaged exactly behind a micro-lens. In Sec. 3.1 and Sec. 4.1 we describe how such a representation of the object space makes the imaging process computationally efficient. The `TextureGridModel` depends on the `LensletGridModel` and the `superResFactor` introduced earlier.

  Finally, the function `LFM_computeResolution(NewLensletGridModel, TextureGridModel, Camera, depthRange, depthStep)` builds the `Resolution` structure as shown in Fig. 5. `NewLensletGridModel` is the distance in pixels between lenslets centers. `Resolution` contains fields like `Nnum`, `TexNnum` (always odd, to ensure a center pixels exists) which refer to the number of pixels/voxels behind/in front of a micro-lens, together with sensor/texture (object) resolution in $\mu m$, or `texScaleFactor` (which is computed as `TexNnum/Nnum`).

  `Resolution.sensMask` is a `Nnum*Nnum` binary mask computed by the function `LFM_computePatchMask(NspacingLenslet, Camera.gridType, sensorRes, Camera.uRad,` Nnum), such that the sensor patches behind the micro-lens perfectly fill the sensor plane without overlapping; this is particularly important for hexagonal grid MLAs, where the discretization of the patches is not symmetric with respect to the center of a micro-lens (NspacingLenslet vs. Nnum). Fig. 6 (left) shows such a lenslet mask. Resolution.texMask is analogous for the object space patches aligned with the micro-lenses. Ultimately, the struct also keeps depth related (axial resolution) input parameters.

- LensletCenters, as computed in the function LFM_computeLensCenters(NewLensletGridModel, TextureGridModel, Resolution.sensorRes, Camera.focus, Camera.gridType) and illustrated in Fig. 5 (bottom), is a struct containing the lenslet centers coordinates in pixels (LensletCenters.px), in $\mu m$ (LensletCenters.metric), the position of the central micro-lens (LensletCenters.offset), as well as the centers of the object space patches (LensletCenters.vox). LensletCenters.px has a third dimension in case of multi-focus MLAs (Fig. 6 (right)) in order to store the lenslet type along side its center x/y coordinates.



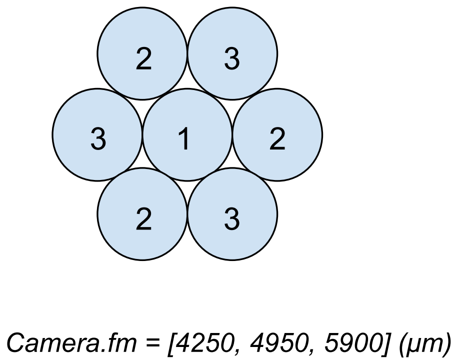


Figure 6: Left: Non-overlapping sensor lenslet mask. Right: Multi-focus MLA with three mixed focal length micro-lenses.

## 2.2 FLFM setup

Based on the setup description (from user inputs and configuration file; see paragraphs 1.4.1 and 1.4.2), function [Camera, LensletGridModel] = FLFM_setCameraParams(configFile, superResFactor) computes extra parameters relevant for the reconstruction:

- Camera.spacingPixels = Camera.spacingPixels * superResFactor. We scale the spacing between micro-lenses according to the super-sampling factor superResFactor. See paragraph 1.4.1.
- Camera.objRad - radius of the microscope objective under paraxial approximation. Camera.objRad = Camera.fobj * Camera.NA.

- `Camera.k` - wave number. `Camera.k` $= 2 * \pi *$ `Camera.n/Camera.WaveLength`.
- `Camera.M` - total system magnification factor. `Camera.M = Camera.fm*Camera.f1/(Camera.f2*Camera.fobj)`.
- `Camera.fsRad` - the radius of the field stop. `Camera.fsRad = Camera.lensPitch/2 * Camera.f2/Camera.fm`; see Fig. 7.
- `Camera.fovRad` - the radius of the object space field of view. `Camera.fovRad = Camera.fsRad * Camera.fobj/Camera.f1`.
- `LensletGridModel.VSpacing` - vertical spacing (in pixels) between neighboring lenslets. In case of `LensletGridModel.gridType == 'hex'`, `LensletGridModel.VSpacing = round(sqrt(3)/2*LensletGridModel.HSpacing)`. Otherwise (for regular grid arrays) `LensletGridModel.VSpacing` and `LensletGridModel.HSpacing` are the same.

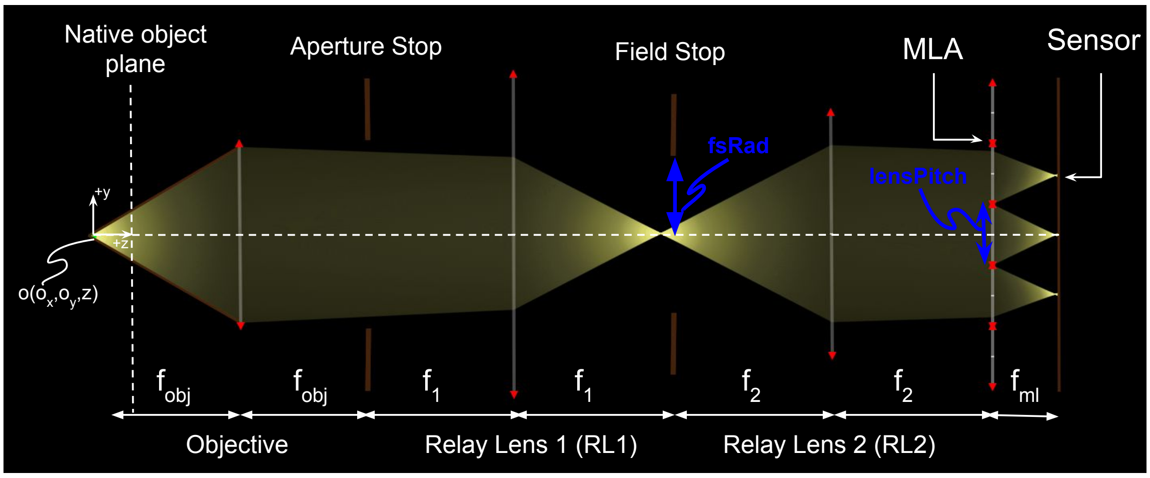
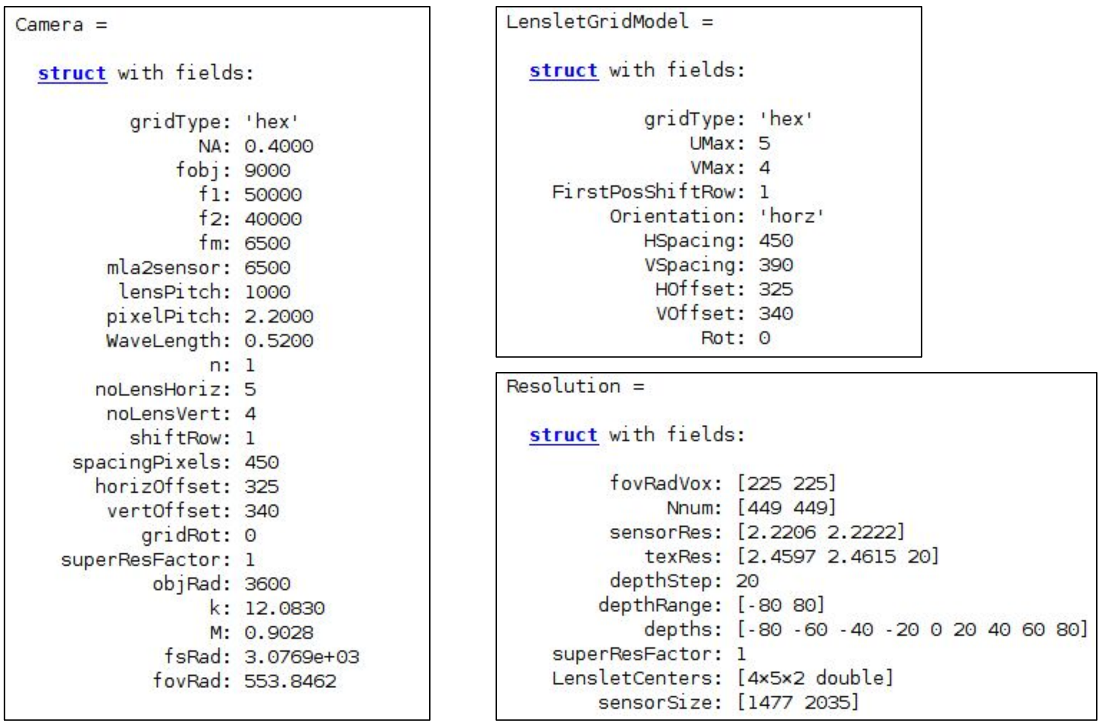


Figure 7: FLFM ray diagram and setup specific quantities.

Function `[LensletCenters, Resolution] = FLFM_computeGeometryParameters (CalibrationImage, Camera, LensletGridModel, depthRange, depthStep)` uses a calibration image, together with the `Camera` and `LensletGridModel` data structures and the user inputs to retrieve the lenslet centers and to compute several resolution quantities relevant for the reconstruction. Fig. 8 illustrates the content of the `Camera` and `LensletGridModel` data structures for the experimental FLFM configuration used to acquire the cotton fibers image in Fig. 1 (c).

The `FLFM_computeGeometryParameters` function returns the following:

- `Resolution` contains all the sensor and object space resolution related quantities, as displayed in Fig. 8. The function `FLFM_computeResolution(LensletGridModel, Camera, depthRange, depthStep)` builds the `Resolution` structure as shown in Fig. 8. It contains fields like `Nnum` (always odd, to ensure a center pixel exists) which refers to the number of pixels behind each micro-lens, together with sensor/texture(object space) resolution in $\mu m$ or the radius of the object side field of view in voxels, `Resolution.fovRadVox`



```
Camera =

  struct with fields:

         gridType: 'hex'
               NA: 0.4000
             fobj: 9000
               f1: 50000
               f2: 40000
               fm: 6500
       mla2sensor: 6500
        lensPitch: 1000
       pixelPitch: 2.2000
       WaveLength: 0.5200
                n: 1
      noLensHoriz: 5
       noLensVert: 4
         shiftRow: 1
    spacingPixels: 450
      horizOffset: 325
       vertOffset: 340
          gridRot: 0
   superResFactor: 1
           objRad: 3600
                k: 12.0830
                M: 0.9028
            fsRad: 3.0769e+03
           fovRad: 553.8462
```

```
LensletGridModel =

  struct with fields:

            gridType: 'hex'
                UMax: 5
                VMax: 4
    FirstPosShiftRow: 1
         Orientation: 'horz'
            HSpacing: 450
            VSpacing: 390
             HOffset: 325
             VOffset: 340
                 Rot: 0
```

```
Resolution =

  struct with fields:

         fovRadVox: [225 225]
              Nnum: [449 449]
         sensorRes: [2.2206 2.2222]
            texRes: [2.4597 2.4615 20]
         depthStep: 20
        depthRange: [-80 80]
            depths: [-80 -60 -40 -20 0 20 40 60 80]
    superResFactor: 1
    LensletCenters: [4x5x2 double]
        sensorSize: [1477 2035]
```

Figure 8: Setup, lenslet grid and resolution related parameters.

= `[round(Camera.fovRad./texRes(1)), round(Camera.fovRad./texRes(2))]`.
`Resolution.sensorSize` is the size of the input light field image, `LensletImage`.

- `LensletCenters`, as computed in the function `FLFM_computeLensCenters(CalibrationImage, Camera, LensletGridModel)` contains the lenslet centers coordinates in pixels.

  In order to detect the exact lenslet centers, in function `transformationsStack = FLFM_retrieveEItransformations(LensletGridModel, CalibrationImage)`, we first extract the elemental images from the `CalibrationImage` using the `LensletGridModel`:

  – `LF = FLFM_extractEI(LensletGridModel, CalibrationImage)`,

  we then pick a reference elemental image (the most central one):

  – `fixed = LF(:,:,ceil(size(LF,3)/2), ceil(size(LF,4)/2))`,

  and register all the other elemental images to this reference one, in order to retrieve the translational transformation between them:

  – `tform = imregtform(LF(:,:,i,j), fixed, 'translation', optimizer, metric)`.

  The `transformationsStack` contains the transformations corresponding to all the elemental images.

  Having computed these translational offsets, we can now find the exact centers of the micro-lenses by correcting the coordinates of the centers in the uniformly spaced grid

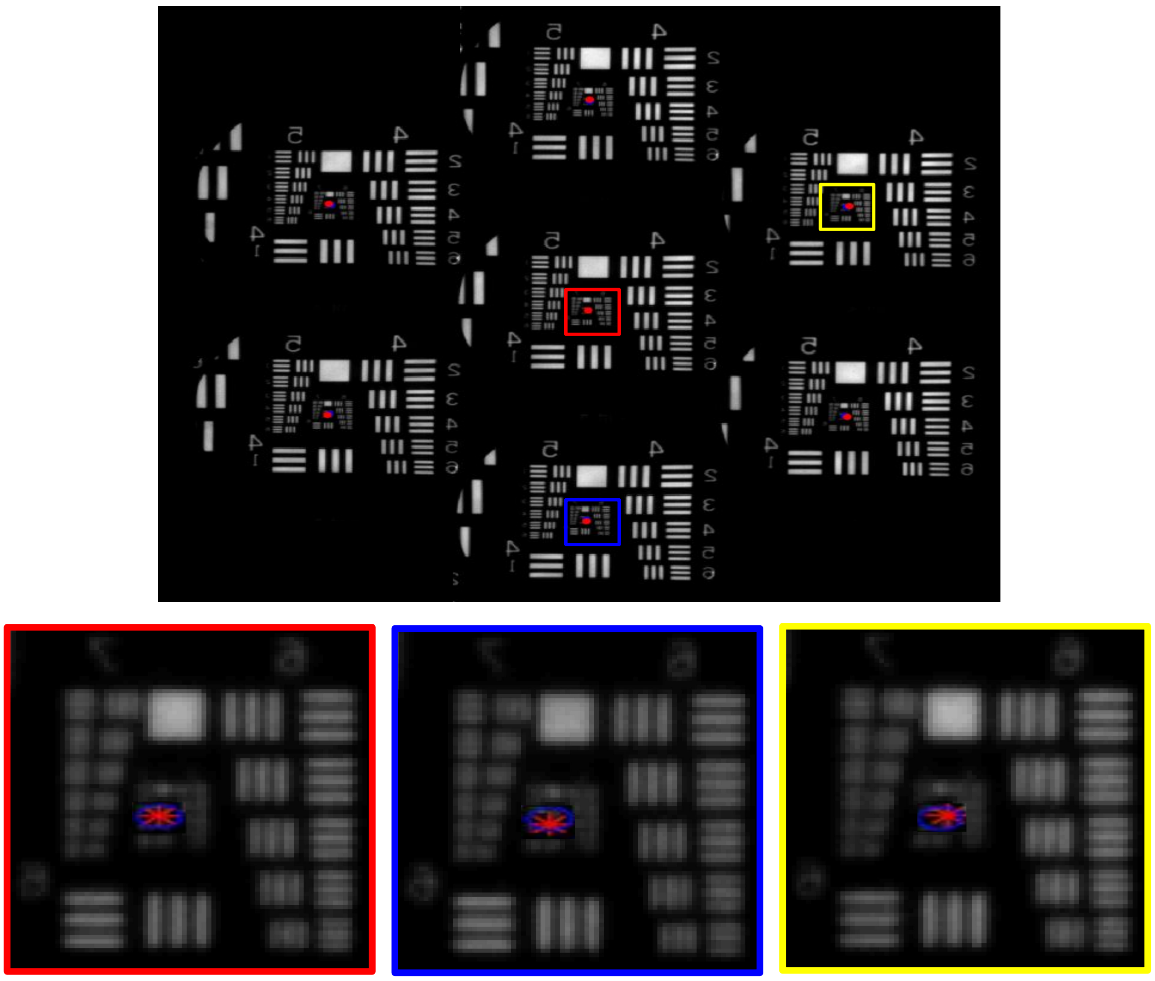


Figure 9: FLFM light field image of the USAF-1951 resolution target and elemental image close-ups: uniformly spaced micro-lens centers according to the `LensletGridModel` specifications (red stars) vs. corrected centers via elemental image registration (blue circles).

estimated from the `LensletGridModel`:

- `centersUniform = LFBuildGrid(LensletGridModel, Camera.gridType)`
- ...
- `LensletCenters(j,k,1) = centersUniform(j,k,1) - transformationsStackj,k.T(3,1);`
- `LensletCenters(j,k,2) = centersUniform(j,k,2) - transformationsStackj,k.T(3,2).`

Fig. 9 shows a light field image of the USAF-1951 resolution target and elemental image close-ups overlaid with the uniformly spaced (according to `LensletGridModel`) micro-lens centers (red stars) and the real (corrected) centers (blue circles).

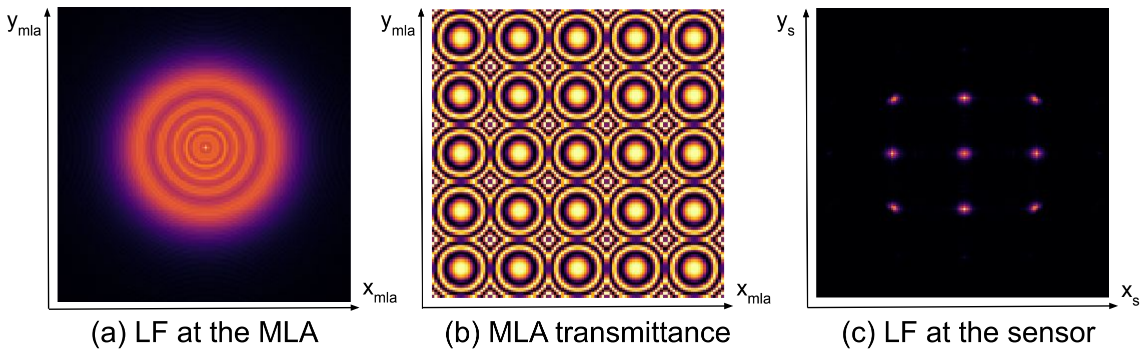


Figure 10: Step-by-step LF point spread function computation.

## 3 Light Field Point Spread Function

### 3.1 LFPSF in conventional LFM

When passing through an optical system, an ideal source point generates a diffraction pattern at the observation plane, known as the system's point spread function (PSF). While in a conventional optical microscope, the PSF is invariant with respect to the position of the point source, in the LF microscope, the light field PSF (LFPSF) is translationally variant i.e., the diffraction pattern generated behind the MLA depends on the 3D position of the point source.

Fortunately, the repeating nature of the MLA grid makes the LFPSF to be periodic such that for source points at $s * \frac{lensPitch}{M}$ apart ($s$ is any integer scalar factor), their response on the sensor (LFPSFs) are identical up to a $s * lensPitch$ translation.

When we represent the discrete forward imaging model as:

$$i = H * t, \tag{1}$$

the operator $H$ represents the discrete LFPSF, $i$ the sensor image (LF measurements) and $t$ the discretized imaged 3D object.

Due to the periodicity of the LFPSF, the computational burden of computing the columns of matrix $H$ reduces dramatically as we only compute and store the LFPSF for a limited number (`Resolution.TexNnum*Resolution.TexNnum`) of discrete source points per axial depth; we call these the forward projection patterns. Consequently, $H$ can be efficiently applied (forward projection) as a series of convolutions (each PSF pattern as kernel over corresponding source points) at every object depth. This procedure is described in Sec. 4.1.

Function `[H, Ht] = LFM_computeLFMatrixOperators(Camera, Resolution, LensletCenters)` uses the data structures introduced in Sec. 2.1 to pre-compute the forward (`H`) and backward (`Ht`) projection patterns in several steps:

- Function `PSFsize = LFM_computePSFsize(maxDepth, Camera)` first computes the maximum blur size in pixels (area of the wave-front distribution) at the MLA plane in order to know the needed size of the `H, Ht` matrix containers. `maxDepth` is that depth among

`Resolution.depthRange` which is further away from the front focal point of the microscope objective, such that it generates the largest blur in extent. Once we know the maximum size of the LFPSF for the chosen axial range, using the `Resolution.sensorRes` we define the sensor plane coordinates (in $\mu m$), `Resolution.xspace, Resolution.yspace` as well as local lenslet coordinates (relative to a lenslet center), `Resolution.xMLspace, Resolution.yMLspace`.

- Function `psfWaveStack = LFM_calcPSFAllDepths(Camera, Resolution)` computes the wave-front distribution at the MLA plane generated by point source on the optical axis at every depth in `Resolution.depths`. This intermediary PSF at the MLA plane is shift invariant and so it is enough to compute the response for a single source point, for simplicity, a point on the optical axis. In order to optimize the implementation and computation time, we exploit the nature of the wave-based PSF:

  - Due to the circular symmetry of the PSF, we only compute one quarter of the 2D distribution and then replicate it for the full range.
  - The PSF for source points at equal distance from the focus point of the objective are complex conjugates. In practice, in general, we are interested in reconstructing axial ranges symmetrically around the native object plane, e.g. $[-50, 50]\mu m$; we then only effectively compute the responses at the MLA for half of the range and use their complex conjugates for the rest.

  The simulated PSF at the MLA plane for a point at depth $\Delta z = -20$ is shown in Fig. 10 (a). For these simulations, we emulated a LFM setup as described in the `LFConfig.yaml` of Fig. 1 (a).

- Once we have the `psfWaveStack` at the MLA plane, `[H, Ht] = LFM_computePatternsSingleWaves (psfWaveStack, Camera, Resolution, tolLFpsf)` computes the forward and back-projection patterns for LFM setups with single focus MLA.

  Similarly, function `LFM_computePatternsMultiWaves(psfWaveStack, Camera, Resolution, tolLFpsf)` adapts the functionality for multi-focus (`Camera.focus = 'multi'`) MLA setups as depicted in Fig. 6 (right). Over the remaining of this section we will point out to the multi-focus specific functionality when relevant. The forward/backward light patterns computation is implemented in several steps:

  - Function `ulensPattern = LFM_ulensTransmittance(Camera, Resolution)` computes the 2D transmittance function for one micro-lens and `MLARRAY = LFM_mlaTransmittance (Camera, Resolution, ulensPattern)` replicates the `ulensPattern` one `lensPitch` apart for the size of the MLA. The simulated 2D MLA transmittance function is depicted in Fig. 10 (b).
  - Function `H = LFM_computeForwardPatternsWaves(psfWaveStack, MLARRAY, Camera, Resolution)` computes the forward projection for every point ( `Resolution.TexNnum * Resolution.TexNnum * length(Resolution.depths)`) inside a patch aligned with the central (w.r.t the optical axis) micro-lens. The size of this patch is chosen such that its image on the sensor is formed exactly behind one micro-lens. It is then sufficient to compute the LFPSFs for these points and then apply them in a shifted matter for any object size in front of the microscope.

In case of regular grid MLAs, it is sufficient to compute the LFPSF (response at the sensor) for only one quarter of the object space patch (`Resolution.TexNnum_half * Resolution.TexNnum_half * length(Resolution.depths)`) due to symmetry, and then use shift rotated versions of these for the rest of the discrete points in the patch. Unfortunately, such optimization is not possible in case of hexagonal grid MLAs, as the object space cannot be split into symmetrically (with respect to the center point of a patch) discretized patches; this is due to the integer rounding of the half `lenspitch` displacement of the micro-lens every other row in the hex MLA. This discrimination gives rise to the `Camera.range` flag introduced in Sec. 2.1:

∗ `Camera.range = 'quarter'`, for regular grid MLA setups;

∗ `Camera.range = 'full'`, for hexagonal grid MLA setups.

Coming back to computing the forward patterns, for each discrete point in the object space range (full or quarter patch) described above, we laterally shift the previously compute 2D PSF (at the MLA plane) corresponding to the axial position of the point (from `psfWaveStack`) to the $xy$ position of the current point, using the function `newImg = imShift2(Img, ShiftX, SHiftY)`. Then the shifted PSF (`psfSHIFT`) is passed through the MLA by applying the MLA transmittance:

∗ `psfMLA = psfSHIFT.*MLARRAY`.

The response is then propagated to the sensor using the function:

∗ `LFpsfAtSensor = prop2Sensor(psfMLA, sensorRes, Camera.mla2sensor, Camera.WaveLength, 0)`,

which implements the Rayleigh-Sommerfeld diffraction.

Finally, the 2D response patterns are shifted back and the results are stored in sparse format:

∗ `H(aa_tex, bb_tex, c) = sparse(abs(double(LFpsf).2̂))`.

Here, `aa_tex, bb_tex` represent the local lateral object patch coordinates and `c` indexes the axial coordinate, `Resolution.depths(c)`:

∗ `aa_tex = 1..Resolution.TexNnum`,

∗ `bb_tex = 1..Resolution.TexNnum`,

∗ `c = 1..length(Resolution.depths)`.

In case of the multi-focus LFM setup, we compute three (as many as the number of different micro-lens focal length available in the MLA) different sets of forward patterns. In order to do so, we simulate three different 2D MLA transmittance functions, each with one of the three micro-lens in the center (w.r.t the optical axis of the system), by shifting around the available focal lengths, e.g.:

∗ `CameraShift.fm = circshift(Camera.fm, -1);`

∗ `ulensPattern = LFM_ulensTransmittance(CameraShift, Resolution);`,

∗ `MLARRAY = LFM_mlaTransmittance(CameraShift, Resolution, ulensPattern).`.

Fig. 10 (c) shows a simulated forward pattern for source a point on the optical axis (`(aa_tex, bb_tex) = (Resolution.TexNnum_half, Resolution.TexNnum_half)` at $-20\mu m$.

– Function `H = ignoreSmallVals(H, tolLFpsf)` is a convenience function which first clamps the values of `H` smaller than `tolLFpsf` in order to speed up the computations (see convolution in Sec. 4.1). Then the individual 2D LFPSFs, `H(aa,bb,c)` are normalized to `[0,1]` range.

– Function `Ht = LFM_computeBackwardPatterns(H, Resolution, range, lensOrder)` computes the backward light transport patterns, for every discrete point on the sensor plane, behind a micro-lens. Analogous to the forward pass, it is sufficient to compute the patterns for only a limited set (behind the central micro-lens) of points (`Resolution.Nnum * Resolution.Nnum`) as the entire sensor plane contains only shifted version of these.

  The `lensOrder` argument is left empty (`[]`) for single-focus MLA setups, and it is only used in case of multi-focus setups when we compute three different sets of back-projection patterns, one for each micro-lens type:

  ∗ `lensOrder = [1,2,3];`

  ∗ `Ht1 = LFM_computeBackwardPatterns(H, Resolution, Camera.range, lensOrder);`

  ∗ `Ht2 = LFM_computeBackwardPatterns(H, Resolution, Camera.range, circshift(lensOrder, -1));`

  ∗ `Ht3 = LFM_computeBackwardPatterns(H, Resolution, Camera.range, circshift(lensOrder, -2));`

  The backward light propagation represents the inverse process of the forward projection, and it is thus a mapping from the 2D sensor space to the 3D object space; every backward pattern is stored in a 3D container:

  ∗ `Ht = cell(coordsRange(1), coordsRange(2), nDepths)`,
  where `coordsRange = Resolution.Nnum` when `Camera.range = 'full'` and `coordsRange = Resolution.Nnum_half` when `Camera.range = 'quarter'`.

  In order to compute the backward projection patterns, we iterate through every pixel behind the central micro-lens and compute its 3D object space response (which part -and to what extent- of the 3D object space affects the current pixel).

  Function `tempback = LFM_backwardProjectSinglePoint(H, Resolution, imgSize, texSize, currentPixel, lensletCenters, range, lensOrder);` returns the back-projection response for one activated pixel, `currentPixel`. The idea to obtain the back-projection pattern for an active pixel is to forward project the whole (an area as wide as it can be captured) 3D space in front of the microscope and acknowledge which object space points were seen by the active pixel and to what extent (intensity of the LFPSF); then this is the object space response to our active sensor pixel (the back-projection pattern).

  In practice, in order to forward project the 3D object space, we convolve the object with the LFPSF, `H`. However, since the LFPSF is different for different source point positions, the object space needs to be split into discrete sets of points which give the same (only translated) sensor response and apply each `H(aa_tex, bb_tex, c)` to the corresponding points and store the sensor response back into the volume at the same locations. This process is equivalent to convolving each rotated forward pattern kernel with the sensor image with a single active pixel and grabbing the result of this convolution only at the relevant (which generate the forward pattern in discussion) coordinates in the volume:

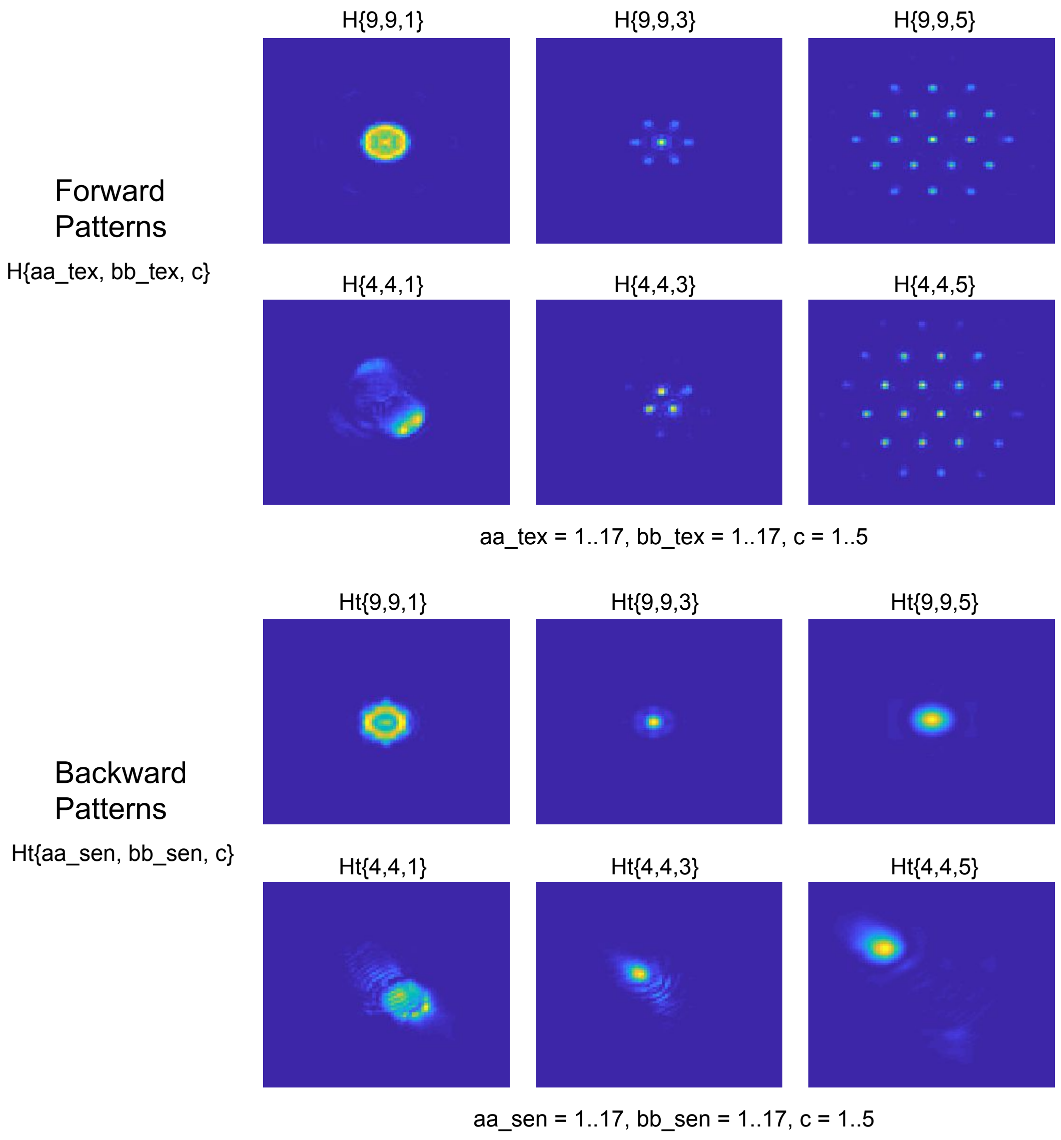


Figure 11: Example forward and backward light transport patterns for a hexagonal grid MLA system.

  * `H_rot = imrotate(H(aa_tex,bb_tex,cc), 180);`
  * `tempSlice = sconv2singlePointFlip(imgSize, currentPixel, H_rot, flipX, flipY, 'same');`
    % Instead of a 2D convolution between `H_rot` and the sensor image, we have implemented a computationally efficient single point convolution, as the sensor image has only one active pixel. `flipX, flipY` are flags used in case of `Camera.range = 'quarter'`, when we only compute forward projection patterns for one quarter of the patch in front of the central micro-lens, and thus for the rest of the points the LFPSFs are flipped versions of the available ones.
  * `sliceCurrentDepth(indicesTex) = sliceCurrentDepth(indicesTex) + tempSlice(indicesSen);`
    % `indicesTex` and `indicesSen` are the relevant (associated with `H(aa_tex,bb_tex,cc)`) volume and sensor coordinates. `indicesTex` and `indicesSen` are different when we reconstruct at a different lateral resolution than the sensor resolution. For a comprehensive understanding, see `backwardProjectSinglePoint.m` file. We repeat this for all the object depths, as one sensor pixel back-projects to the entire axial range.

Finally, we shift the patterns back (needed for convolution, see Sec. 4.1) and store them:

  * `Ht(aa_sensor,bb_sensor,:)  = tempbackShift`, with
  * `aa_sensor = 1..coordsRange(1)`,
  * `bb_sensor = 1..coordsRange(2)`.

Fig. 11 shows example forward and backward light transport patterns for a defocused LFM setup with hexagonal grid MLA and a desired reconstruction with axial resolution `Resolution.depths = [-15, -10, -5, 0, 5]` $\mu m$, and `Resolution.TexNnum = Resolution.Nnum = 17`. We show forward patterns for source points on the optical axis (Fig. 11 (top row)), as well as off-axis ones (Fig. 11 (second row)), for the `-15, -5, 5` $\mu m$ depth planes. The displayed backward patterns correspond to the central pixel behind a micro-lens (Fig. 11 (third row)), as well as an offset pixel (Fig. 11 (bottom row)).

The derivation of the wave-based LFPSF implemented in this framework is described in detail in [3].

## 3.2 LFPSF in Fourier LFM

When passing through an optical system, an ideal source point generates a diffraction pattern at the observation plane, known as the system's point spread function (PSF). Similarly to a conventional wide field microscope, the light field PSF (LFPSF) of the Fourier LFM is translationally invariant for a fixed axial position, due to the strategic placement of the MLA at the back aperture stop of the objective. Conveniently, this allows us to describe the imaging process in FLFM as a 2D convolution (at each axial slice) of the object with the LFPSF. This process is described in Sec. 4.2.

When we represent the discrete forward imaging model as:

$$i = H * t, \tag{2}$$

$H$ represents the discrete LFPSF (a stack of the 2D PSFs for each object space axial position in the object), $i$ the sensor image (LF measurements) and $t$ the discretized imaged object. $*$ is then the slice-by-slice 2D convolution operator.

Function `[H, Ht] = FLFM_computeLFMatrixOperators(Camera, Resolution)` uses the data structures introduced in Sec. 2.2 to pre-compute the LFPSF (`H`) and its transpose (`Ht`) in several steps:

- Function `psfStack = FLFM_calcPSFAllDepths(Camera, Resolution)` computes the wavefront distribution incident on the micro-lens array (MLA) plane generated by a point source on the optical axis at every depth in `Resolution.depths`. We first propagate the field to the native object plane (NOP; see Fig. 7), and then function `[U1, LU1] = FLFM_lensProp(U0, LU0, Camera.WaveLength, Camera.fobj)` implements the Fourier property of a lens (under paraxial context) to generate the field at the back focal plane (see AS plane in Fig. 7) of the objective as a scaled Fourier transform of the field at the front focal plane. The field at the MLA plane is then just a scaled version of the field at the AS plane, by the relay magnification factor, `Mrelay = Camera.f2/Camera.f1`. For implementation details of these functions, see the corresponding Matlab functions (Sec. 5).

- Once we have the `psfStack` at the MLA plane, `[H, Ht] = FLFM_computeLFPSF (psfStack, Camera, Resolution, tolLFpsf)` computes the forward and backward light propagation models of the FLFM setup in several steps:

  – Function `ulensPattern = FLFM_ulensTransmittance(Camera, Resolution)` computes the 2D transmittance function for one micro-lens and `MLARRAY = FLFM_mlaTransmittance (Resolution, ulensPattern)` replicates the `ulensPattern` one `lensPitch` apart for the size of the MLA.

  – Then each of the 2D PSFs in the `psfSTACK` is passed through the MLA by applying the MLA transmittance:

    * `psfREF = psfSTACK(:,:,c);`
    * `psfMLA = psfSHIFT.*MLARRAY`.

    The response is then propagated to the sensor using the function:

    * `LFpsfSensor = prop2Sensor(psfMLA, sensorRes, Camera.mla2sensor, Camera.WaveLength, 0)`,

    which implements the Rayleigh-Sommerfeld diffraction transfer function.

    Finally, the 2D response patterns are stored in the `H` container:

    * `H(1,1,c) = sparse(abs(double(LFpsfSensor).2̂))`.

  – Function `H = ignoreSmallVals(H, tolLFpsf)` is a convenience function which first clamps the values of `H` smaller than `tolLFpsf` in order to speed up the computations (see convolution in Sec. 4.2). Then the individual 2D LFPSFs, `H(1,1,c)` are normalized to the `[0,1]` range.

  – `Ht` stores the backward light transport patterns, which are the rotated LFPSF kernels:

    * `Ht(1,1,c) = imrotate(H(1,1,c), 180)`.

- Finally, `Ht = normalizeHt(Ht)` makes sure the transpose LFPSF adds up to 1 for each source axial position. This step ensures that when applying the inverse mapping ($H_t * i$), the energy in the reconstruction is kept through the Richardson-Lucy iterative deconvolution scheme.

The derivation of the wave-based LFPSF implemented in this framework is described in detail in [5].

## 4 Forward-/Backward projection

### 4.1 Projection operators in conventional LFM

Function `Projection = LFM_forwardProject(H, realSpace, LensletCenters, Resolution, imgSize, Camera.range)` implements the light field forward projection operator. It applies the pre-computed (see Sec. 3.1) forward patterns, `H` to a given 3D volume, `realSpace` and returns a light field image, `Projection` of size `imgSize`.

In order to apply the patterns in `H` to the object, `realSpace`, we pre-store the object voxels with coordinates `(aa_tex,bb_tex)` relative to the '`Resolution.TexNnum*Resolution.TexNnum`' object space repetition patches (imaged behind exactly one micro-lens extent), into `indicesTex(aa_tex,bb_tex)`. The corresponding sensor image coordinates are pre-stored into `indicesImg(aa_tex,bb_tex)`.

Once we sort out which object coordinates generate which pattern, for every reconstruction depth, `cc = 1..length(Resolution.depths)`, we grab, at a time, only those parts of the object corresponding to current `indicesTex(aa_tex,bb_tex)` and apply the associated `H(aa_tex,bb_tex,cc)`:

- `realspaceCurrentDepth = realSpace(:,:,cc);` % Grab object slice (current depth) to be forward projected.
- `tempspace(indicesImgaa_tex,bb_tex) = realspaceCurrentDepth(indicesTex(aa_tex,bb_tex));` % From the current slice, keep only the `indicesTex(aa_tex,bb_tex)` lateral locations.
- `projectedPattern = conv2(tempspace, Hs, 'same');` % Convolve `tempspace` with the corresponding pattern, `Hs = H(aa_tex,bb_tex, cc)`.

  The non-zero values in `H(aa_tex,bb_tex, cc)` (effective sparse kernel size) directly affect the computation time. The further away a source point is from the native object plane (NOP, depicted in blue in Fig. 4) of the LFM, the larger the LFPSF size on the sensor. This means object depths further away from the NOP require are computationally more expensive to forward/backward-project.

The above procedure is executed in parallel on workers using Matlab's `parallel pool` functionality, for all `aa_tex = 1..Resolution.TexNnum, bb_tex = 1..Resolution.TexNnum`.

Finally, the intermediate projection are accumulated:

- `Projection = Projection + projectedPattern.`

Function `BackProjection = LFM_backwardProject(Ht, projection, lensCenters, Resolution, texSize, range)` implements the light field backward projection operator. It applies the pre-computed (see Sec. 3.1) backward patterns, `Ht` to a given light field image, `projection` and returns 3D volume, `BackProjection` of size `texSize`.

In order to tapply he patterns in `Ht` to the light field image, `projection`, the `LFM_backwardProject` function implements selective sparse convolutions in an analogous manner to the `LFM_forwardProject` function described above.

`imgSize` and `texSize` are pre-computed container sizes for the returned light field image in case of the `forwardProjectACC` function and 3D object in case of the `backwardProjectACC` function, respectively. The `imgSize` and `texSize` sizes are different when we reconstruct an object at a lateral resolution different from the sensor resolution. Then `texSize` is retrieved as(see `main.m` script):

- `imgSize = size(correctedLensletImage);`
- `imgSize = imgSize + (1-mod(imgSize,2));` % ensure odd size
- `texSize = ceil(imgSize.*Resolution.texScaleFactor);`
- `texSize = texSize + (1-mod(texSize,2));` % ensure odd size

Both `LFM_forwardProject` and `LFM_backwardProject` functions apply to single focus MLA setups (`Camera.focus = 'single'`). In case of multi-focus setups (`Camera.focus = 'multi'`), functions `LFM_forwardProjectMultiFocus` and `LFM_backwardProjectMultiFocus` expand and adapt the functionality above in a straightforward manner.

For details on the implementation of the projection operators, see the corresponding Matlab files.

## 4.2 Projection operators in FLFM

Function `Projection = FLFM_forwardProject(H, realSpace)` implements the light field forward projection operator. It applies the pre-computed (see Sec. 3.2) forward light model, `H` to a given 3D volume, `realSpace` and returns a light field image, `Projection`.

In order to project the object, `realSpace` using the LFPSF in `H`, we perform 2D slice-by-slice convolutions at every depth and cumulate the responses:

- `Projection = Projection + conv2(realSpace(:,:,j), full(H(1,1,j)),'same').`

Function `BackProjection = FLFM_backwardProject(Ht, projection)` implements the light field backward projection operator. It applies the pre-computed (see Sec. 3.2) backward patterns, `Ht` to a given light field image, `projection` and returns a 3D volume, `BackProjection`.

In order to apply the patterns in `Ht` to the light field image, `projection`, the `LFM_backwardProject` function implements slice-by-slice convolutions in an analogous manner to the `FLFM_forwardProject` function described above:

- `BackProjection(:,:,j) = conv2(projection , full(Ht(1,1,j)),'same').`

For details on the implementation of the projection operators, see the corresponding Matlab files.

## 5 Function References

### 5.1 Microscope geometry

#### 5.1.1 LFM setup

- `Camera = LFM_setCameraParams(configFile, newSpacingPx);`
- `tube2mla = LFM_computeTube2MLA(lensPitch, mla2sensor, deltaOT, objRad, ftl)`
- `[LensletCenters, Resolution, LensletGridModel, NewLensletGridModel] = LFM_computeGeometryParameters(Camera, WhiteImage, depthRange, depthStep, superResFactor, DebugBuildGridModel, imgSize);`
- `[LensletGridModel, gridCoords] = LFM_processWhiteImage(WhiteImage, spacingPx, gridType, DebugBuildGridModel);`
- `[LensletGridModel, GridCoords] = LFM_BuildLensletGridModel(WhiteImg, gridType, GridModelOptions, DebugDisplay)`
- `[GridCoords] = LFBuildGrid( LensletGridModel, gridType);`
- `LensletGridModel = LFM_setGridModel(SpacingPx, FirstPosShiftRow, UMax, VMax, HOffset, VOffset, Rot, Orientation, gridType);`
- `Resolution = LFM_computeResolution(LensletGridModel, TextureGridModel, Camera, depthRange, depthStep);`
- `mask = LFM_computePatchMask(spacing, gridType, res, patchRad, Nnum);`
- `newMask = LFM_fixMask(mask, NewLensletSpacing, gridType);`
- `lensletCenters = LFM_computeLensCenters(LensletGridModel, TextureGridModel, sensorRes, focus, gridType);`
- `centersWithTypes = LFM_addLensTypes(lensCentersPx, matrixCenter);`
- `lensCurrentType = LFM_extractLensType(lensletCenters, type).`

#### 5.1.2 FLFM setup

- `[Camera, LensletGridModel] = FLFM_setCameraParams(configFile, superResFactor);`
- `Resolution = FLFM_computeResolution(LensletGridModel, Camera, depthRange, depthStep);`
- `LensletCenters = FLFM_computeLensCenters(CaibrationImage, Camera, LensletGridModel);`

- [LensletCenters, Resolution] = FLFM_computeGeometryParameters(CalibrationImage, Camera, LensletGridModel, depthRange, depthStep);
- [GridCoords] = LFBuildGrid(LensletGridModel, gridType).

## 5.2 Image rectification

### 5.2.1 LFM

- FixAll = LFM_retrieveTransformation(LensletGridModel, NewLensletGridModel);
- [CorrectedLensletImage, CorrectedWhiteImage] = LFM_applyTransformation(LensletImage, WhiteImage, FixAll, LensletCenters, debug).

### 5.2.2 FLFM

- transformationsStack = FLFM_retrieveEItransformations(LensletGridModel, calibrationImage);
- LF = FLFM_extractEI(LensletGridModel, LensletImage)

## 5.3 LSPSF model

### 5.3.1 Shift Variant LFPSF Patterns in LFM

- [H, Ht] = LFM_computeLFMatrixOperators(Camera, Resolution, LensletCenters));
- PSFsize = LFM_computePSFsize(maxDepth, Camera);
- usedLensletCenters = LFM_getUsedCenters(PSFsize, lensletCenters);
- psfWaveStack = LFM_calcPSFAllDepths(Camera, Resolution);
- psf = LFM_calcPSF(p1, p2, p3, Camera, Resolution);
- [H, Ht] = LFM_computePatternsSingleWaves(psfWaveStack, Camera, Resolution, tolLFpsf);
- [H, Ht] = LFM_computePatternsMultiWaves(psfWaveStack, Camera, Resolution, tolLFpsf);
- ulensPattern = LFM_ulensTransmittance(Camera, Resolution);
- MLARRAY = LFM_mlaTransmittance(Camera, Resolution, ulensPattern);
- H = LFM_computeForwardPatternsWaves(psfWaveStack, MLARRAY, Camera, Resolution);
- f1 = prop2Sensor(f0, sensorRes, z, lambda, idealSampling);
- H = ignoreSmallVals(H, tol);
- Ht = LFM_computeBackwardPatterns(H, Resolution, range, lensOrder);

- Backprojection = LFM_backwardProjectSinglePoint(H, Resolution, imgSize, texSize, currentPixel, lensletCenters, range, lensOrder);
- tempSlice = sconv2singlePointFlip(imgSize, currentPixel, Ht, flipX, flipY, 'same');
- Ht = normalizeHt(Ht).

### 5.3.2 LFPSF in Fourier LFM

- [H, Ht] = FLFM_computeLFMatrixOperators(Camera, Resolution);
- psfSTACK = FLFM_calcPSFAllDepths(Camera, Resolution);
- [psf] = FLFM_calcPSF(p1, p2, p3, Camera, Resolution);
- [u2, L2] = FLFM_lensProp(u1, L1, lambda, z);
- [H, Ht] = FLFM_computeLFPSF(psfSTACK, Camera, Resolution, tolLFpsf);
- ulensPattern = FLFM_ulensTransmittance(Camera, Resolution);
- MLARRAY = FLFM_mlaTransmittance(Resolution, ulensPattern);
- f1 = prop2Sensor(f0, sensorRes, z, lambda, idealSampling);
- H = ignoreSmallVals(H, tol);
- Ht = normalizeHt(Ht).

## 5.4 Projection operators

### 5.4.1 LFM

- Projection = LFM_forwardProject( H, realSpace, lensCenters, Resolution, imgSize, range);
- BackProjection = LFM_backwardProject(Ht, projection, lensCenters, Resolution, texSize, range);
- Projection = LFM_forwardProjectMultiFocus( H, realSpace, lensCenters, Resolution, imgSize, range);
- BackProjection = LFM_backwardProjectMultiFocus( Ht, projection, lensCenters, Resolution, texSize, range).

### 5.4.2 FLFM

- Projection = FLFM_forwardProject(H, realSpace);
- BackProjection = FLFM_backwardProject(Ht, projection).

## 5.5 Deconvolution

### 5.5.1 Estimate-Maximize-Smooth algorithm

- `widths = LFM_computeDepthAdaptiveWidth(Camera, Resolution);`
- `lanczos2FFT = LFM_buildAntiAliasingFilter(filterSize, widths, n);`
- `reconVolume = deconvEMS(forwardFUN, backwardFUN, LFimage, iter, reconVolume, filterFlag, kernelFFT).`

### 5.5.2 Richardson-Lucy algorithm

- `recon = deconvRL(forwardFUN, backwardFUN, LensletImage, iter, init).`

### 5.5.3 One-Step-Late algorithm

- `recon = deconvOSL(forwardFUN, backwardFUN, LensletImage, iter, init, lambda).`

## 5.6 I/O

- `[LensletImage, WhiteImage, configFile] = LFM_selectLFImages(dataset);`
- `Camera = ReadYaml(configFile).`