DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Focal Flow: Supporting material

Citation
Alexander, Emma, Qi Guo, S.J. Koppal, S.J. Gortler, & T. Zickler. 2016. Focal Flow: Supporting material. Harvard Computer Science Group Technical Report TR-01-16.

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:28219150

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Focal Flow: Supporting material

E. Alexander
Qi Guo
S.J. Koppal
S.J. Gortler
and
T. Zickler

TR-01-16

Computer Science Group
Harvard University
Cambridge, Massachusetts

Focal Flow: Supporting material

E. Alexander, Qi Guo, S.J. Koppal, S.J. Gortler, & T. Zickler

TR-01-16

This report accompanies (Alexander et al., ECCV 2016 [1]) and contains additional mathematical details, experimental results, and information for building and calibrating a focal flow sensor.

Contents

1 Alternate Derivations of the Focal Flow Constraint **2**
1.1 From Taylor Expansion 2
1.2 From Sinusoidal Textures 3

2 Supplementary Experimental Results **4**
2.1 Performance versus noise 4
2.2 Working range versus aperture. 4
2.3 Performance for varying apertures and textures 5
2.4 Empirical comparison with single-shot depth from defocus . 6

3 Sensor Implementation **7**
3.1 List of Parts 7
3.2 Calibration Procedure 8
3.3 Matlab implementation of measurement algorithm 9

1 Alternate Derivations of the Focal Flow Constraint

The main paper [1] proves that the Gaussian aperture filter and Laplacian image operator are the only filter-operator pair that can remove texture dependence from the residual (R in equation (11) in [1]) that results from computing differential optical flow using images with defocus blur. This filter-operator pair provides access to depth and scene velocity through the focal flow equations (equations (18-20) in [1]):

$$0 = \begin{bmatrix} I_x & I_y & (xI_x + yI_y) & (I_{xx} + I_{yy}) \end{bmatrix} \vec{v} + I_t,$$
$$\vec{v} = [u_1, u_2, u_3, v]^T = \left[-\frac{\dot{X}\mu_s}{Z}, -\frac{\dot{Y}\mu_s}{Z}, -\frac{\dot{Z}}{Z}, -\frac{\dot{Z}}{Z}\left(1 - \frac{\mu_f}{Z}\right)\left(\frac{\Sigma\mu_s}{\mu_f}\right)^2 \right]^T, \tag{1}$$
$$Z = \frac{(\mu_s^2\Sigma^2\mu_f)u_3}{(\mu_s^2\Sigma^2)u_3 - (\mu_f^2)v}, \tag{2}$$
$$(\dot{X}, \dot{Y}, \dot{Z}) = -\left(Zu_1/\mu_s, Zu_2/\mu_s, Zu_3\right). \tag{3}$$

Putting aside the question of uniqueness, the correctness of these equations is easily verified by setting $I = k * P$ with Gaussian k and simply taking the relevant derivatives. Here we provide two alternative confirmations of equation (1) that may provide additional intuition. One of these derivations is based on a truncated Taylor expansion, mirroring a common derivation for linearized optical flow. The other is based on sinusoidal textures, illustrated in Figure 1 of [1] and analyzed there for inherent sensitivity.

1.1 From Taylor Expansion

Following the well-known Taylor series derivation for differential optical flow, we can consider the difference in intensity at a pixel between a pair of images taken a time step Δt apart. We take advantage of the fact that the brightness of the underlying sharp texture does not change, but we must correct for the change in blur to process the images.

To do so, we assume Gaussian blur kernels k,

$$k(x, y, \sigma) = \frac{e^{-\frac{x^2+y^2}{2\Sigma^2\sigma^2}}}{2\pi\Sigma^2\sigma^2}, \tag{4}$$

and define a reblurring filter b that takes narrow Gaussians to wider Gaussians under spatial convolution:

$$k(x, y, \sigma_2) = b\left(x, y, \sigma_1, \sigma_2\right) * k(x, y, \sigma_1). \tag{5}$$

This reblurring filter takes the form

$$b(x, y, \sigma_1, \sigma_2) = k(x, y, \sqrt{\sigma_2^2 - \sigma_1^2}). \tag{6}$$

The unchanging texture brightness constraint states that for an all-in-focus pinhole image P,

$$P(x + \Delta x, y + \Delta y, t + \Delta t) = P(x, y, t), \tag{7}$$

with features moving from (x, y) to $(x + \Delta x, y + \Delta y)$ on the image. We are free to convolve both sides of this constraint by a Gaussian, for example:

$$k(x, y, \sigma(t + \Delta t)) * P(x + \Delta x, y + \Delta y, t + \Delta t) = k(x, y, \sigma(t + \Delta t)) * P(x, y, t). \tag{8}$$

Then, for images blurred with different Gaussian kernels, where we set the sign of Δt without loss of generality so that $\sigma(t + \Delta t) > \sigma(t)$, we can express this modification of the unchanging texture brightness constraint in terms of blurred images I:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = b\left(x, y, \sigma(t), \frac{Z(t + \Delta t)}{Z(t)}\sigma(t + \Delta t)\right) * I(x, y, t), \tag{9}$$

where the $\frac{Z+\Delta Z}{Z}$ term accounts for the change in magnification between images. Taking the Taylor expansion of either side and dropping terms above first order, we have the approximation

$$I(x,y,t) + I_x\Delta x + I_y\Delta y + I_t\Delta t \;\approx \delta(x,y) * I(x,y,t) + \left(\Delta t \left(\dot{Z}\sigma/Z + \dot{\sigma}\right) b_{\sigma_2}(x,y,\sigma,\sigma) * I(x,y,t)\right). \quad (10)$$

Subtracting the $I(x,y,t)$ term from each side, dividing by Δt, and noting that

$$\left(\dot{Z}\sigma/Z + \dot{\sigma}\right) b_{\sigma_2}(x,y,\sigma,\sigma) = -\,v\,(b_{xx}(x,y,\sigma,\sigma) + b_{yy}(x,y,\sigma,\sigma)) \quad (11)$$

$$= -\,v(\delta_{xx} + \delta_{yy}), \quad (12)$$

our approximate constraint becomes

$$I_x\frac{\Delta x}{\Delta t} + I_y\frac{\Delta y}{\Delta t} + I_t \approx -\,v(I_{xx} + I_{yy}). \quad (13)$$

In the absence of blur, $v = 0$ and this is identical to optical flow. In the limit as Δt approaches zero, and under the separation of $(\dot{x}, \dot{y})$ into translation and magnification terms, this produces the focal flow constraint (1).

1.2 From Sinusoidal Textures

For general sinusoidal texture

$$T(a,b) = \sin(\omega_a a + \omega_b b + \phi_0) \quad (14)$$

a pinhole camera will record the image

$$P(x,y,t) = \sin(\omega_x(t)x + \omega_y(t)y + \phi(t)), \quad (15)$$

$$\omega_x = -\,\frac{Z(t)}{\mu_s}\omega_a, \quad (16)$$

$$\omega_y = -\,\frac{Z(t)}{\mu_s}\omega_b, \quad (17)$$

$$\phi = \omega_a X(t) + \omega_b Y(t) + \phi_0. \quad (18)$$

Under Gaussian blur as in equation (4), frequency and phase will not change but amplitude will:

$$I(x,y,t) = B(t)\sin(\omega_x x + \omega_y y + \phi), \quad (19)$$

$$B(t) = \max_{\phi}\,(k * P) = e^{-\Sigma^2(\omega_x^2+\omega_y^2)\sigma^2/2}. \quad (20)$$

The derivatives of this image are as follows:

$$I_x = \omega_x B\cos(\omega_x x + \omega_y y + \phi), \quad (21)$$

$$I_y = \omega_y B\cos(\omega_x x + \omega_y y + \phi), \quad (22)$$

$$I_{xx} = -\,\omega_x^2 B\sin(\omega_x x + \omega_y y + \phi), \quad (23)$$

$$I_{yy} = -\,\omega_y^2 B\sin(\omega_x x + \omega_y y + \phi), \quad (24)$$

$$I_t = (\dot{\phi} + \dot{\omega}_x x + \dot{\omega}_y y)B\cos(\omega_x x + \omega_y y + \phi) + \dot{B}\sin(\omega_x x + \omega_y y + \phi), \quad (25)$$

so that

$$I_t = \frac{\omega_a\dot{X}}{\omega_x}I_x + \frac{\omega_b\dot{Y}}{\omega_y}I_y + \frac{\dot{\omega}_x x}{\omega_x}I_x + \frac{\dot{\omega}_y y}{\omega_y}I_y + \frac{\dot{B}}{-B(\omega_x^2+\omega_y^2)}(I_{xx} + I_{yy}) \quad (26)$$

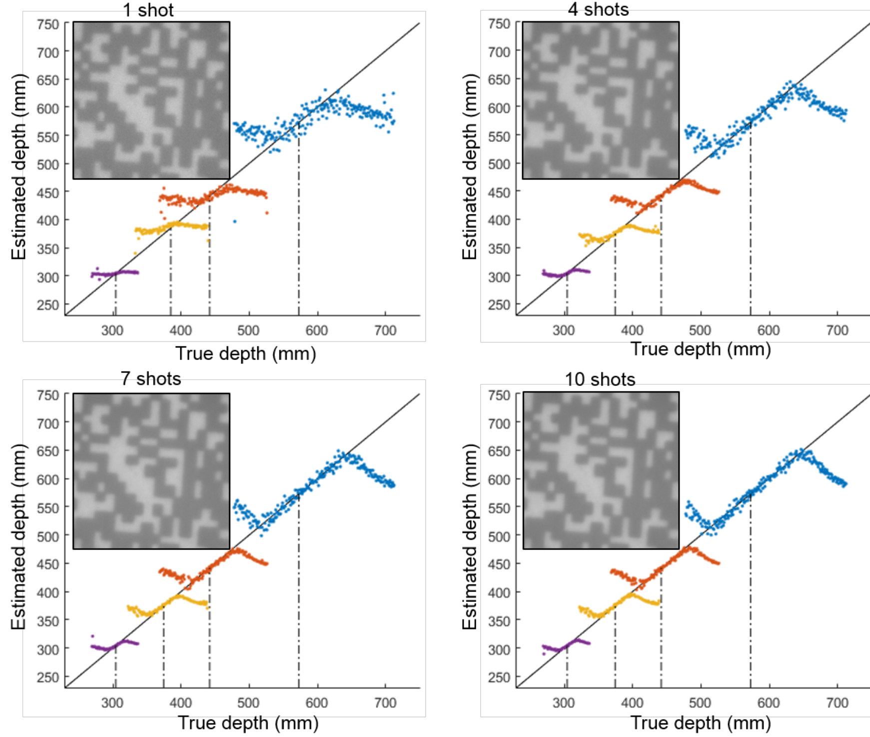
$$= -\,u_1 I_x - u_2 I_y - u_3 x I_x - u_3 y I_y - v I_{xx} - v I_{yy}. \quad (27)$$

By the linearity of convolution and differentiation, equation (27) holds for all sum-of-sinusoid textures, so that the focal flow constraint applies to any texture with a Fourier transform.

2 Supplementary Experimental Results

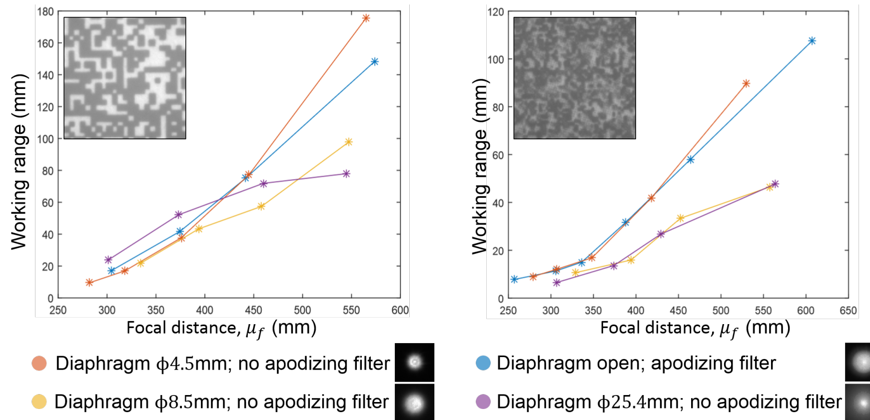
2.1 Performance versus noise

To counteract sensor noise, several shots can be averaged to create an input image (inset) to the measurement algorithm. (zoom in to see difference in noise level). Comparing measured depth to ground truth (solid black line) shows that that, as expected, measurement accuracy improves with shot count. Unless otherwise noted, all results here and in [1] use 10-shot averages.



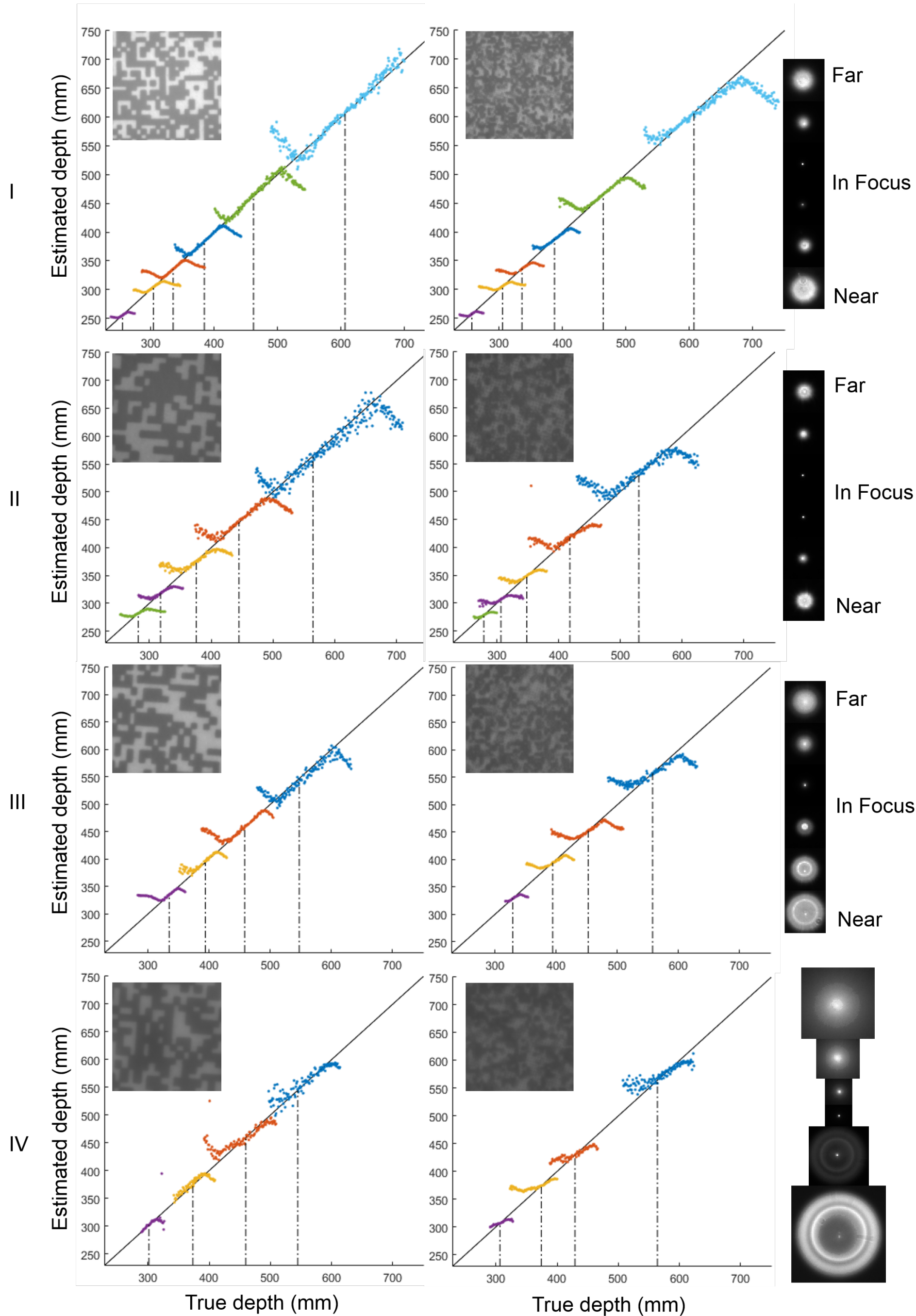
2.2 Working range versus aperture.

We show working range ($\equiv$ range for which depth error $< 1\%\mu_f$) versus focal depth μ_f for four apertures, over two scene textures. We also show a sample point spread function for each aperture, at the same scale as the input image.



2.3 Performance for varying apertures and textures

Distance measurements versus ground truth (black lines) for a variety of focal distances and aperture configurations. Each row is a different aperture configuration, and the left and middle columns show results for both lower-frequency scene textures (left column) and higher-frequency scene textures (middle column). The right-most column shows corresponding sample point spread functions, each for a variety of depths. The measurement algorithm is quite robust to deviations from the idealized Gaussian blur model. From top to bottom, the aperture configurations are: (I) diaphragm open, with apodizing filter; (II) diaphragm ⌀4.5mm, no filter; (III) diaphragm ⌀8.5mm, no filter; (IV) diaphragm ⌀25.4mm, no filter.



2.4 Empirical comparison with single-shot depth from defocus

Traditional (multi-shot) depth from defocus depends on camera actuation to change the camera's blur kernels between exposures, which requires substantially different hardware than the internally-fixed focal flow sensor. However, the sensor hardware that is used for single-shot depth from defocus with a coded aperture (e.g., [2]) is quite similar to that of focal flow. In that approach, a single image is captured through a binary coded aperture, and a non-blind deconvolution algorithm is used to explicitly deconvolve each image patch with a discrete set of depth-indexed blur kernels before selecting the most "natural" result.

We used simulation to compare the depth performance of focal flow to that of single-shot depth from defocus. In particular, we used the aperture pattern and deconvolution algorithms provided by Levin et al. [2]. The simulation used sensor dimensions $f = 100\text{mm}$ and $\mu_s = 130\text{mm}$ for both approaches. For single-shot depth-from-defocus, it used the binary aperture pattern from [2]. For focal flow, it used Gaussian blur kernels (covariance matrix $= [1\text{mm}, 0; 0, 1\text{mm}]$), and all other settings were the same as those used in the experiments presented in [1]. Zero-mean Gaussian noise with variance $\Sigma = 10^{-6}$ was added to the simulated input images for both methods. We used a randomly selected texture from CuRRET database[1], and to capture the best possible performance of the single-shot approach, we used the same texture for the training step (parameters λ_k) of [2]. For focal flow, we simulated input images for depths between 400mm and 500mm at increments of 1mm; and for the single-shot approach, we simulated input images for depths between 320mm and 720mm with increments of 4mm. As in the main paper, the working range (400mm-500mm) of focal flow is determined as the set of depths for which the absolute depth error is less than 1% of the focal distance μ_f. For each approach, we obtained depth estimates for each of the 101 increments, and the RMS depth error was computed over these estimates.

The depth performance of the two approaches is shown in the table below. The table includes separate performance numbers for each of the three deconvolution algorithms proposed in [2], as implemented in Matlab by the authors.[2] The depth performance of focal flow and single-shot depth from defocus is complimentary. The working volume of the single-shot approach is larger, but its precision is more than seven times lower and its computation time is at least hundreds of times greater.

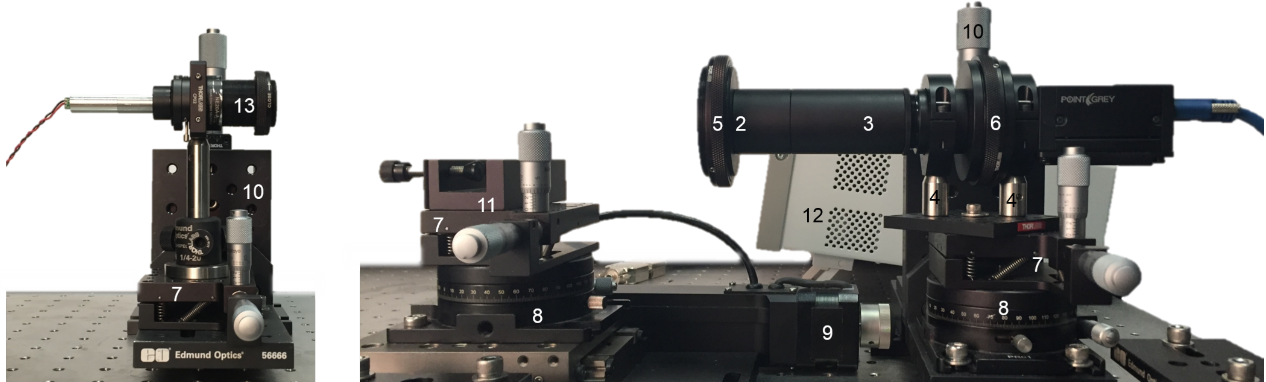
Method	**Running Time (sec/estimate)**	**Working Range (mm)**	**RMS depth error (mm)**
Focal Flow	0.03	400-500	2.94
Coded DfD: L2 deconvolution in frequency domain, we = 0.01	7.90	320-720	20.95
Coded DfD: L2 deconvolution in frequency domain, we = 0.002	7.93	320-720	44.73
Coded DfD: L2 deconvolution we = 0.01, max_it = 80	159.03	320-720	56.25
Coded DfD: sparse deconvolution we = 0.01, max_it = 200	1456.62	320-720	45.24

[1] http://www.cs.columbia.edu/CAVE/software/curet/, [3]

[2] https://groups.csail.mit.edu/graphics/CodedAperture/

3 Sensor Implementation

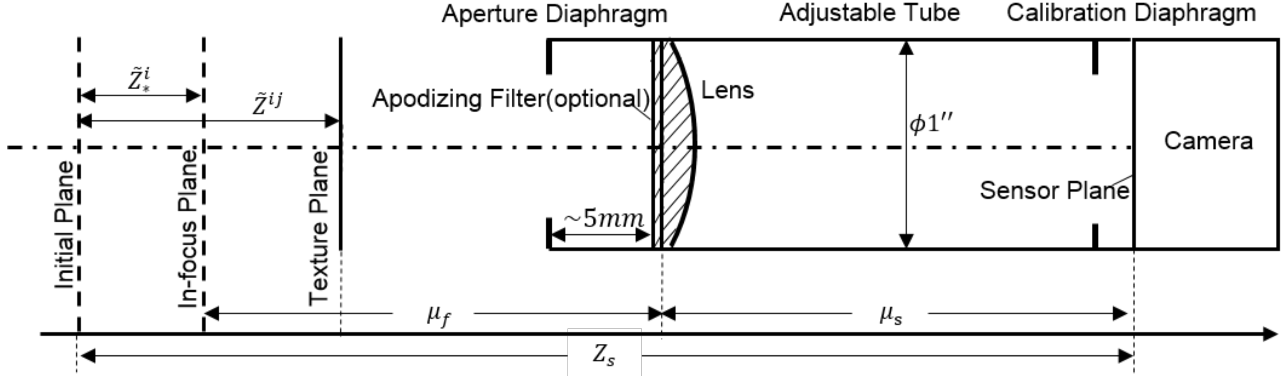
3.1 List of Parts



No.	Component	Source	Part Number	Quantity	Description
1	Camera	Point Grey	GS3-U3-23S6M-C	1	High speed, monochrome, powered by USB
2	Lens	Thorlabs	LA1509-A	1	Planar-convex, $\varnothing 1''$, $f = 100$mm, AR coated(350-700nm)
2	Apodizing Filter (Optional)	Thorlabs	NDYR20B	1	Reflective, $\varnothing$25mm, ND, OD: 0.04 - 2
3	Lens Tube	Thorlabs	SM1 Family	Flexible	SM1 thread, $\varnothing 1''$, recommend SM1V15 for adjustable μ_s
4	Lens Tube Mounts	Thorlabs	SM1TC+TR075	2	
5	Aperture Diaphragm	Thorlabs	SM1D25D or SM1D12D	1	SM1 thread, $\varnothing 2''$ or $\varnothing 1''$, removed when using apodizing filter
6	Calibration Diaphragm	Thorlabs	SM2D25D	1	SM2 thread, $\varnothing 2''$, connected with SM1A2 and SM2A6
7	Pitch & Yaw Platform	Thorlabs	PY003	3	
8	Rotation Platform	Thorlabs	SM2D25D	2	
9	Translation Stage	Thorlabs	LNR50S	1	Controlled and powered by 12
10	X-Y Translation Stage	Thorlabs & EO	2×PT1+PT101+ PT102+EO56666	2	
11	Wide Plate Holder	Thorlabs	FP02	1	
12	Stepper Motor Controllers	Thorlabs	BSC201	1	Powered by 110V, connected with PC via USB
13	Laser	Thorlabs	CPS532	1	Mounted with AD11F, SM1D12SZ, CP02, NE20A-A, SM1D12D

3.2 Calibration Procedure

Calculation of depth and velocity from each scene vector $\boldsymbol{v}$ requires calibrating the camera parameters that appear in equations (2, 3). There are only two degrees of freedom (sensor distance μ_s and aperture width parameter Σ) because the object focal distance μ_f is determined by the lens' known focal length ($f = 100$mm). We visualize all parameters mentioned in the calibration process in the diagram below.



The sensor distance μ_s is straightforward to calibrate, but the aperture width parameter Σ is much less so, especially when the blur kernels that are induced by the lens and aperture configuration deviate substantially from the ideal Gaussian form. Its value could be calibrated by measuring the system's point spread functions $k(x, y, Z_i)$ for depths Z_i and optimizing Σ to fit the measured blur kernels with $\kappa(r) = e^{-r^2/4\Sigma^2}$. A more direct method, which we find to be effective in practice, is to forgo measuring the blur kernels and to instead optimize the effective aperture width Σ using a depth-based objective. This directly improves the measurement that we actually care about (depth), in addition to absorbing the effects of nonidealities described above.

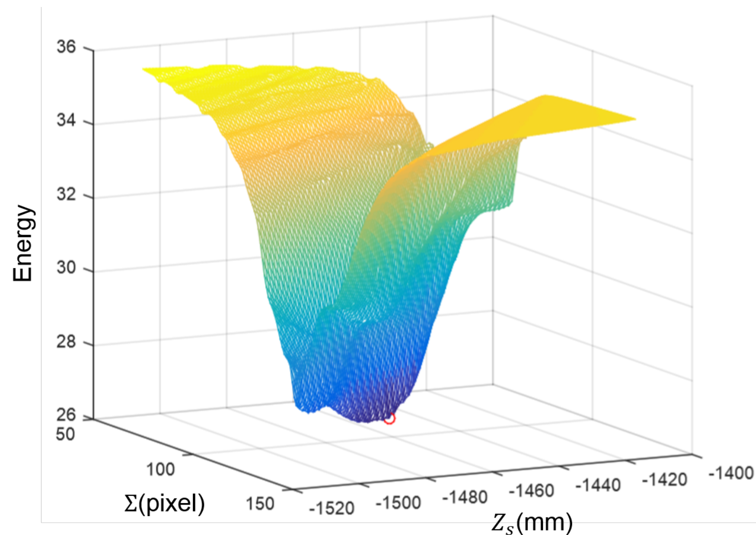
We perform a depth-based calibration as follows. We mount a textured plane on a high-precision translation stage in front of the sensor, and carefully align it to be normal to the system's optical axis with an axis-aligned laser. The initial distance from the sensor plane to the texture Z_s is inferred during calibration. The stage provides precise information about relative texture depths $\tilde{Z}(t)$, and these are related to the (unknown) veridical depths via $Z(t) = Z_s + \tilde{Z}(t)$.

The lens tube is sequentially adjusted to create a small number N of (unknown) sensor distances $\mu_s^i, i \in \{1, 2, ...N\}$, and for each sensor distance, the texture is translated to, and imaged at, a dense set of M depths $\tilde{Z}^{ij}, j \in \{1, 2, ...M\}$. We ensure that this set of depths includes the optical system's (unknown) point of focus, and for each of the tube lengths i, the relative depth of this point $\tilde{Z}^i_*$ is identified by the image that is most sharp. Note that knowledge of these relative focal depths $\{\tilde{Z}^i_*\}$ determines the entire set of sensor distances $\{\mu_s^i\}$ up to a single unknown scalar offset Z_s via the combination of the thin lens equation $1/f = 1/\mu_s^i + 1/\mu_f^i$ and the simple observation that $\mu_s^i + \mu_f^i = Z_s + \tilde{Z}^i_*$.

Next, we consider each triple of images that corresponds to an adjacent triple of texture locations $\tilde{Z}^{i(j-1)}$, $\tilde{Z}^{ij}$, $\tilde{Z}^{i(j+1)}$, and by computing derivatives and aggregating per-pixel linear constraints over a large window around the image center, we determine the corresponding least-squares scene vector $\boldsymbol{v}^{ij}$ as described in the measurement algorithm of the main paper. Given hypothesized values for the effective aperture width Σ and for the texture offset Z_s (and therefore $\{\mu_s^i, \mu_f^i\}$) these scene vectors produce depth estimates $Z^{ij}_{\text{est}}(\boldsymbol{v}^{ij}; \Sigma, Z_s, \tilde{Z}^i_*)$ via equation (2), and we can simultaneously determine the two unknown parameters by solving

$$\Sigma^*, Z_s^* = \arg\min_{\Sigma, Z_s} \sum_{i,j} \rho(Z^{ij}_{est}(\boldsymbol{v}^{ij}; \Sigma, Z_s, \tilde{Z}^i_*) - (Z_s + \tilde{Z}^{ij})), \tag{28}$$

through iterative optimization. We find it useful to include a robust functional, $\rho(x) = \{x^2 \text{ if } |x| \leq 1, \text{and } 1$ otherwise$\}$, to reduce the effect of outliers. A typical cost surface can be seen below. The surface of the energy function (28) has an optimal value (shown as red circle) in a wide range of feasible regions, which ensures the convergence of the optimization in the calibration process.



3.3 Matlab implementation of measurement algorithm

```
function [Z,xdot,ydot,zdot] = focalflow(I1,I2,I3,params)
% This function computes the focal flow given three images and parameters.
% INPUT:
% I1, I2, I3: HxW double images
% params: a structure specifying the following parameters
% Sigma: the filter standard deviation
% mu_s: sensor distance
% mu_f: focal distance
% xp, yp: principal point (default to the center)
% szxW, szyW: half size of window function (default to 50)
% mask: a HxW 0,1 mask to remove some areas like background, shadow, etc. (default to all 1)

% OUTPUT:
% Z: depth
% xdot, ydot, zdot: velocity
%
%% Initialization
Sigma = params.Sigma;mu_s = params.mu_s;mu_f = params.mu_f;
if isfield(params,'szxW')
    szxW = params.szxW;
    szyW = params.szyW;
else
    szxW = 35;
    szyW = 35;
end
diffKernels = [-1/2,0,1/2];
im = cell(3,1);im{1} = I1;im{2} = I2;im{3} = I3;
H = size(I1,1);W = size(I1,2);
if isfield(params,'xp')
    xp = params.xp;
    yp = params.yp;
else
    xp = floor(W/2);
    yp = floor(H/2);
end
% Remove the edge
szx = xp - 20; szy = yp - 20;
if isfield(params,'mask')
    mask = params.mask;
else
    mask = ones(H,W);
end
```

```
%% Temporal difference
imT = (im{3} - im{1})/2;
imT = imT(yp-szy:yp+szy, xp-szx:xp+szx);

%% Spatial difference
[XX,YY] = meshgrid(-szx+xp-xp:szx+xp-xp,-szy+yp-yp:szy+yp-yp);
ratio = 10^4; % normalization factor to avoid numerical instability
XX = XX ./ ratio;
YY = YY ./ ratio;

imX = conv2(im{2},-diffKernels,'same');
imY = conv2(im{2},-diffKernels','same');
imXX = conv2(imX,-diffKernels,'same');
imYY = conv2(imY,-diffKernels','same');
% Crop only the necessary part
imX = imX(yp-szy:yp+szy, xp-szx:xp+szx) .* mask(yp-(szy):yp+(szy), xp-(szx):xp+(szx));
imY = imY(yp-szy:yp+szy, xp-szx:xp+szx) .* mask(yp-(szy):yp+(szy), xp-(szx):xp+(szx));
imXX = imXX(yp-szy:yp+szy, xp-szx:xp+szx) .* mask(yp-(szy):yp+(szy), xp-(szx):xp+(szx));
imYY = imYY(yp-szy:yp+szy, xp-szx:xp+szx) .* mask(yp-(szy):yp+(szy), xp-(szx):xp+(szx));
XimX = imX .* XX;
YimY = imY .* YY;

% Precompute entries of the normal equation A'Ax=A'b (Ix^2, IxIy, Ix(xIx + yIy), etc.)
% at every pixel. These will be windowed momentarily.
A11 = imX .* imX;
A12 = imX .* imY;
A13 = imX .* (XimX + YimY);
A14 = imX .* (imXX + imYY);

A22 = imY .* imY;
A23 = imY .* (XimX + YimY);
A24 = imY .* (imXX + imYY);

A33 = (XimX + YimY) .* (XimX + YimY);
A34 = (XimX + YimY) .* (imXX + imYY);

A44 = (imXX + imYY) .* (imXX + imYY);

b1 = imX .* imT;
b2 = imY .* imT;
b3 = (XimX + YimY) .* imT;
b4 = (imXX + imYY) .* imT;


mask = mask(yp-(szy-szyW):yp+(szy-szyW), xp-(szx - szxW):xp+(szx - szxW));
mask = reshape(mask,(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1);
W = ones(szyW * 2 + 1, szxW * 2 + 1);

% Window the entries by convolution with a windowing function W.
% ATA and ATb are big matrices storing all the windowed entries of A'A and A'b for each pixel

ATA = zeros(4,4,(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1));
ATb = zeros(1,4,(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1));
ATA(1,1,:) = reshape(conv2(A11,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(1,2,:) = reshape(conv2(A12,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(1,3,:) = reshape(conv2(A13,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(1,4,:) = reshape(conv2(A14,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(2,1,:) = ATA(1,2,:);
ATA(2,2,:) = reshape(conv2(A22,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(2,3,:) = reshape(conv2(A23,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(2,4,:) = reshape(conv2(A24,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(3,1,:) = ATA(1,3,:);
ATA(3,2,:) = ATA(2,3,:);
ATA(3,3,:) = reshape(conv2(A33,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(3,4,:) = reshape(conv2(A34,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATA(4,1,:) = ATA(1,4,:);
ATA(4,2,:) = ATA(2,4,:);
ATA(4,3,:) = ATA(3,4,:);
ATA(4,4,:) = reshape(conv2(A44,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
```

```
ATb(1,1,:) = reshape(conv2(b1,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATb(1,2,:) = reshape(conv2(b2,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATb(1,3,:) = reshape(conv2(b3,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;
ATb(1,4,:) = reshape(conv2(b4,W,'valid'),(2*(szx - szxW) + 1) * (2*(szy - szyW) + 1),1) .* mask;

%% Computing the u vector, the u = [u_1,u_2,u_3,v]'
% Invert all perpixel 4x4 matrices in parallel using cofactors
detATA = ATA(1,1,:) .* ATA(2,2,:) .* ATA(3,3,:) .* ATA(4,4,:) + ...
    ATA(1,1,:) .* ATA(2,3,:) .* ATA(3,4,:) .* ATA(4,2,:) + ...
    ATA(1,1,:) .* ATA(2,4,:) .* ATA(3,2,:) .* ATA(4,3,:) + ...
    ...
    ATA(1,2,:) .* ATA(2,1,:) .* ATA(3,4,:) .* ATA(4,3,:) + ...
    ATA(1,2,:) .* ATA(2,3,:) .* ATA(3,1,:) .* ATA(4,4,:) + ...
    ATA(1,2,:) .* ATA(2,4,:) .* ATA(3,3,:) .* ATA(4,1,:) + ...
    ...
    ATA(1,3,:) .* ATA(2,1,:) .* ATA(3,2,:) .* ATA(4,4,:) + ...
    ATA(1,3,:) .* ATA(2,2,:) .* ATA(3,4,:) .* ATA(4,1,:) + ...
    ATA(1,3,:) .* ATA(2,4,:) .* ATA(3,1,:) .* ATA(4,2,:) + ...
    ...
    ATA(1,4,:) .* ATA(2,1,:) .* ATA(3,3,:) .* ATA(4,2,:) + ...
    ATA(1,4,:) .* ATA(2,2,:) .* ATA(3,1,:) .* ATA(4,3,:) + ...
    ATA(1,4,:) .* ATA(2,3,:) .* ATA(3,2,:) .* ATA(4,1,:) - ...
    ...
    ATA(1,1,:) .* ATA(2,2,:) .* ATA(3,4,:) .* ATA(4,3,:) - ...
    ATA(1,1,:) .* ATA(2,3,:) .* ATA(3,2,:) .* ATA(4,4,:) - ...
    ATA(1,1,:) .* ATA(2,4,:) .* ATA(3,3,:) .* ATA(4,2,:) - ...
    ...
    ATA(1,2,:) .* ATA(2,1,:) .* ATA(3,3,:) .* ATA(4,4,:) - ...
    ATA(1,2,:) .* ATA(2,3,:) .* ATA(3,4,:) .* ATA(4,1,:) - ...
    ATA(1,2,:) .* ATA(2,4,:) .* ATA(3,1,:) .* ATA(4,3,:) - ...
    ...
    ATA(1,3,:) .* ATA(2,1,:) .* ATA(3,4,:) .* ATA(4,2,:) - ...
    ATA(1,3,:) .* ATA(2,2,:) .* ATA(3,1,:) .* ATA(4,4,:) - ...
    ATA(1,3,:) .* ATA(2,4,:) .* ATA(3,2,:) .* ATA(4,1,:) - ...
    ...
    ATA(1,4,:) .* ATA(2,1,:) .* ATA(3,2,:) .* ATA(4,3,:) - ...
    ATA(1,4,:) .* ATA(2,2,:) .* ATA(3,3,:) .* ATA(4,1,:) - ...
    ATA(1,4,:) .* ATA(2,3,:) .* ATA(3,1,:) .* ATA(4,2,:);

B = zeros(4,4,size(ATA,3));
B(1,1,:) = ATA(2,2,:) .* ATA(3,3,:) .* ATA(4,4,:) ...
    + ATA(2,3,:) .* ATA(3,4,:) .* ATA(4,2,:) ...
    + ATA(2,4,:) .* ATA(3,2,:) .* ATA(4,3,:) ...
    - ATA(2,2,:) .* ATA(3,4,:) .* ATA(4,3,:) ...
    - ATA(2,3,:) .* ATA(3,2,:) .* ATA(4,4,:) ...
    - ATA(2,4,:) .* ATA(3,3,:) .* ATA(4,2,:);

B(1,2,:) = ...
      ATA(1,2,:) .* ATA(3,4,:) .* ATA(4,3,:) ...
    + ATA(1,3,:) .* ATA(3,2,:) .* ATA(4,4,:) ...
    + ATA(1,4,:) .* ATA(3,3,:) .* ATA(4,2,:) ...
    - ATA(1,2,:) .* ATA(3,3,:) .* ATA(4,4,:) ...
    - ATA(1,3,:) .* ATA(3,4,:) .* ATA(4,2,:) ...
    - ATA(1,4,:) .* ATA(3,2,:) .* ATA(4,3,:);

B(1,3,:) = ...
      ATA(1,2,:) .* ATA(2,3,:) .* ATA(4,4,:) ...
    + ATA(1,3,:) .* ATA(2,4,:) .* ATA(4,2,:) ...
    + ATA(1,4,:) .* ATA(2,2,:) .* ATA(4,3,:) ...
    - ATA(1,2,:) .* ATA(2,4,:) .* ATA(4,3,:) ...
    - ATA(1,3,:) .* ATA(2,2,:) .* ATA(4,4,:) ...
    - ATA(1,4,:) .* ATA(2,3,:) .* ATA(4,2,:);

B(1,4,:) = ...
      ATA(1,2,:) .* ATA(2,4,:) .* ATA(3,3,:) ...
    + ATA(1,3,:) .* ATA(2,2,:) .* ATA(3,4,:) ...
    + ATA(1,4,:) .* ATA(2,3,:) .* ATA(3,2,:) ...
    - ATA(1,2,:) .* ATA(2,3,:) .* ATA(3,4,:) ...
```

```
    - ATA(1,3,:) .* ATA(2,4,:) .* ATA(3,2,:) ...
    - ATA(1,4,:) .* ATA(2,2,:) .* ATA(3,3,:);

B(2,2,:) = ...
      ATA(1,1,:) .* ATA(3,3,:) .* ATA(4,4,:) ...
    + ATA(1,3,:) .* ATA(3,4,:) .* ATA(4,1,:) ...
    + ATA(1,4,:) .* ATA(3,1,:) .* ATA(4,3,:) ...
    - ATA(1,1,:) .* ATA(3,4,:) .* ATA(4,3,:) ...
    - ATA(1,3,:) .* ATA(3,1,:) .* ATA(4,4,:) ...
    - ATA(1,4,:) .* ATA(3,3,:) .* ATA(4,1,:);

B(2,3,:) = ...
      ATA(1,1,:) .* ATA(2,4,:) .* ATA(4,3,:) ...
    + ATA(1,3,:) .* ATA(2,1,:) .* ATA(4,4,:) ...
    + ATA(1,4,:) .* ATA(2,3,:) .* ATA(4,1,:) ...
    - ATA(1,1,:) .* ATA(2,3,:) .* ATA(4,4,:) ...
    - ATA(1,3,:) .* ATA(2,4,:) .* ATA(4,1,:) ...
    - ATA(1,4,:) .* ATA(2,1,:) .* ATA(4,3,:);

B(2,4,:) = ...
      ATA(1,1,:) .* ATA(2,3,:) .* ATA(3,4,:) ...
    + ATA(1,3,:) .* ATA(2,4,:) .* ATA(3,1,:) ...
    + ATA(1,4,:) .* ATA(2,1,:) .* ATA(3,3,:) ...
    - ATA(1,1,:) .* ATA(2,4,:) .* ATA(3,3,:) ...
    - ATA(1,3,:) .* ATA(2,1,:) .* ATA(3,4,:) ...
    - ATA(1,4,:) .* ATA(2,3,:) .* ATA(3,1,:);

B(3,3,:) = ...
      ATA(1,1,:) .* ATA(2,2,:) .* ATA(4,4,:) ...
    + ATA(1,2,:) .* ATA(2,4,:) .* ATA(4,1,:) ...
    + ATA(1,4,:) .* ATA(2,1,:) .* ATA(4,2,:) ...
    - ATA(1,1,:) .* ATA(2,4,:) .* ATA(4,2,:) ...
    - ATA(1,2,:) .* ATA(2,1,:) .* ATA(4,4,:) ...
    - ATA(1,4,:) .* ATA(2,2,:) .* ATA(4,1,:);

B(3,4,:) = ...
      ATA(1,1,:) .* ATA(2,4,:) .* ATA(3,2,:) ...
    + ATA(1,2,:) .* ATA(2,1,:) .* ATA(3,4,:) ...
    + ATA(1,4,:) .* ATA(2,2,:) .* ATA(3,1,:) ...
    - ATA(1,1,:) .* ATA(2,2,:) .* ATA(3,4,:) ...
    - ATA(1,2,:) .* ATA(2,4,:) .* ATA(3,1,:) ...
    - ATA(1,4,:) .* ATA(2,1,:) .* ATA(3,2,:);

B(4,4,:) = ...
      ATA(1,1,:) .* ATA(2,2,:) .* ATA(3,3,:) ...
    + ATA(1,2,:) .* ATA(2,3,:) .* ATA(3,1,:) ...
    + ATA(1,3,:) .* ATA(2,1,:) .* ATA(3,2,:) ...
    - ATA(1,1,:) .* ATA(2,3,:) .* ATA(3,2,:) ...
    - ATA(1,2,:) .* ATA(2,1,:) .* ATA(3,3,:) ...
    - ATA(1,3,:) .* ATA(2,2,:) .* ATA(3,1,:);

B(2,1,:) = B(1,2,:);
B(3,1,:) = B(1,3,:);
B(3,2,:) = B(2,3,:);
B(4,1,:) = B(1,4,:);
B(4,2,:) = B(2,4,:);
B(4,3,:) = B(3,4,:);

% Clean the space
ATA = [];
IdetATA = 1./ detATA;
flag = reshape(IdetATA > 0.000001,size(B,3),1);
IdetATA = repmat(reshape(IdetATA,1,1,size(B,3)),4,4,1);
B = IdetATA .* B;
IdetATA = [];
B = B .* repmat(ATb,4,1,1);
u = reshape(sum(B,2),4,size(B,3))';
u(flag == 0,:) = 0;
u(:,3) = u(:,3) ./ ratio; %Recover from the normalization
```

```
% Recover Z, xdot, ydot, zdot
Z = mu_s.^2 .* Sigma.^2 .* u(:,3) ./ (mu_s.^2 .* Sigma.^2 .* u(:,3)./ mu_f - mu_f .* u(:,4));

xdot = -u(:,1) .* Z ./ mu_s;
ydot = -u(:,2) .* Z ./ mu_s;
zdot = -u(:,3) .* Z;
Z = reshape(Z,(szy - szyW) * 2 + 1,(szx - szxW) * 2 + 1);
xdot = reshape(xdot,(szy - szyW) * 2 + 1,(szx - szxW) * 2 + 1);
ydot = reshape(ydot,(szy - szyW) * 2 + 1,(szx - szxW) * 2 + 1);
zdot = reshape(zdot,(szy - szyW) * 2 + 1,(szx - szxW) * 2 + 1);
end
```

References

[1] Alexander, E., Guo, Q., Koppal, S., Gortler, S., Zickler, T.: Focal flow: Measuring distance and velocity with defocus and differential motion. In: European Conference on Computer Vision (ECCV). Springer (2016)

[2] Levin, A., Fergus, R., Durand, F., Freeman, W.T.: Image and depth from a conventional camera with a coded aperture. In: ACM Transactions on Graphics (TOG). Volume 26., ACM (2007) 70

[3] Dana, K.J., van Ginneken, B., Nayar, S.K., Koenderink, J.J. In: Reflectance and Texture of Real-world Surfaces