Optics EXPRESS

Matrix-masking to balance nonuniform illumination in microscopy

PONTUS NORDENFELT,[1] JONATHAN M. COOPER,[2] AND AXEL HOCHSTETTER[2,*]

[1]*Division of Infection Medicine, Department of Clinical Sciences Lund, Medical Faculty, Lund University, Lund, Sweden*
[2]*Division of Biomedical Engineering, School of Engineering, University of Glasgow, Glasgow G12 8LT, UK*
**axel_hochstetter@web.de*

Abstract: With a perfectly uniform illumination, the amount and concentration of fluorophores in any (biological) sample can be read directly from fluorescence micrographs. However, non-uniform illumination in optical micrographs is a common, yet avoidable artefact, often caused by the setup of the microscope, or by inherent properties caused by the nature of the sample. In this paper, we demonstrate simple matrix-based methods using the common computing environments MATLAB and Python to correct nonuniform illumination, using either a background image or extracting illumination information directly from the sample image, together with subsequent image processing. We compare the processes, algorithms, and results obtained from both MATLAB (commercially available) and Python (freeware). Additionally, we validate our method by evaluating commonly used alternative approaches, demonstrating that the best nonuniform illumination correction can be achieved when a separate background image is available.

OCIS codes: (100.0100) Image processing; (160.0180) Microscopy; (200.0200) Optics in computing.

References and links

1. A. Köhler, "Ein neues Beleuchtungsverfahren für mikrophotographische Zwecke," Z. Wiss. Mikrosk. **10**, 433–440 (1893).
2. Y. Lu, F. Xie, Y. Wu, Z. Jiang, and R. Meng, "No Reference Uneven Illumination Assessment for Dermoscopy Images," IEEE Signal Process. Lett. **22**(5), 534–538 (2015).
3. F. J. W.-M. Leong, M. Brady, and J. O. McGee, "Correction of uneven illumination (vignetting) in digital microscopy images," J. Clin. Pathol. **56**(8), 619–621 (2003).
4. D. H. Brainard and B. A. Wandell, "Analysis of the retinex theory of color vision," J. Opt. Soc. Am. A **3**(10), 1651–1661 (1986).
5. D. J. Jobson, Z. Rahman, and G. A. Woodell, "Properties and performance of a center/surround retinex," IEEE Trans. Image Process. **6**(3), 451–462 (1997).
6. J. M. Morel, A. B. Petro, and C. Sbert, "A PDE formalization of Retinex theory," IEEE Trans. Image Process. **19**(11), 2825–2837 (2010).
7. R. Kimmel, M. Elad, D. Shaked, R. Keshet, and I. Sobel, "A Variational Framework for Retinex," Int. J. Comput. Vis. **52**(1), 7–23 (2003).
8. J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," Pattern Recognit. **33**(2), 225–236 (2000).
9. J. C. Olivo-Marin, "Extraction of spots in biological images using multiscale products," Pattern Recognit. **35**(9), 1989–1996 (2002).
10. A. Hochstetter, E. Stellamanns, S. Deshpande, S. Uppaluri, M. Engstler, and T. Pfohl, "Microfluidics-based single cell analysis reveals drug-dependent motility changes in trypanosomes," Lab Chip **15**(8), 1961–1968 (2015).
11. P. Nordenfelt, J. M. Cooper, and A. Hochstetter, "Matrix-masking to balance nonuniform illumination in microscopy," https://nordlab.med.lu.se/?page_id=34.
12. T. Ferreira and W. Rasband, *ImageJ User Guide*, ImageJ/Fij (2012).
13. The MathWorks Inc, "Correcting Nonuniform Illumination," https://se.mathworks.com/help/images/examples/correcting-nonuniform-illumination.html.

#327006 https://doi.org/10.1364/OE.26.017279
Journal © 2018 Received 6 Apr 2018; revised 31 May 2018; accepted 2 Jun 2018; published 20 Jun 2018

1. Introduction

In most common microscopy configurations, including those used in fluorescence microscopy, the light source is aligned for Koehler illumination [1], which ensures that the structure of the light source (as either rectangles generated by the LEDs or a light bulb's coil) does not introduce optical artefacts in the microscopy pictures. Such illumination provides a clean light with a radial symmetric Gaussian intensity gradient and thus a non-uniform illumination. This uneven illumination can cause significant issues in many fields (for example it can lead to problems in the assessment of dermoscopy images in the clinical sciences [2]).

In the last decades, the correction of uneven illumination (including vignetting) in microscopy has become common, with techniques relying upon a number of techniques, including: the acquisition of additional images to subtract the background; the use of inherent image features by blurring [3]; or by extracting illumination data from the image (e.g. the variational framework for Retinex (VFR) [2,4–7]). These techniques often require the use of specific software and/or a deep mathematical understanding. Alternative approaches to background reduction include binarization [8] and spot detection [9] which can remove greyscale details of the original images (leading to micrographs that are unsuitable for detailed analysis or quantification).

Here, we present a simple method for matrix-based balancing of the nonuniform illumination and validate the technique in two common computing environments: the commercially available gold standard – MATLAB and its freeware alternative – Python. To demonstrate the practical application of this technique, we study images acquired using commercial fluorescent and brightfield microscopes. The image depict samples of high optical density [Fig. 4] or a microfluidic device [10] (originally designed to study the diffusion of fluorophores and single cell motility in laminar flows [10]), [Figs. 2 and 3].

2. Materials

Raw images and fluorescence micrographs were obtained using an Olympus BX 61 upright microscope (Olympus, Germany) with a Sensicam camera (PCO, Germany) [Fig. 3(a)]. A Xenon lamp was used as light source with a long pass filter set (λExc = 535 nm, AHF, Germany). Rhodamine B (0.5% aqueous solution) was used as the fluorophore, and images were acquired from within in a previously described microfluidic device [10]. A brief description of this device is shown in the Supporting Information [10]. Conventional bright field micrographs were collected using a Zeiss Observer A1, [Figs. 2(a) and 2(b)].

Image processing was performed with both MATLAB 2014 beta developer trial version (MathWorks, USA) and in Python 3.6 with the modules opencv2 to load the images into Python, Numpy to convert the images into arrays/matrices and process them, as well as the matplotlib.pyplot module to display the results using the Spyder and IPython 6.2.1 upgrade (all freeware). The code and all data used in this article is available open access for download [11] at https://nordlab.med.lu.se/?page_id=34.

3. Methodology

The image processing was performed using two central steps, namely, the acquisition or generation of background information and then, subsequently use this background data to process sample images. Depending on the available background data of the illumination, we will review four different approaches to balance out nonuniform illumination, as shown in the decision tree below [Fig. 1].

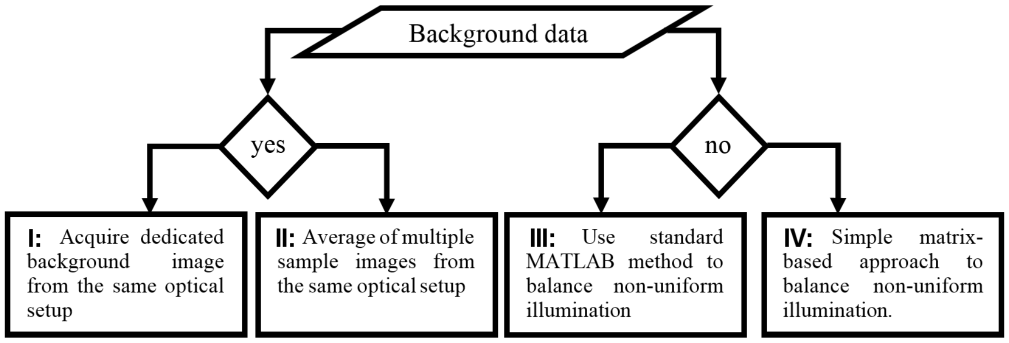

Background data
yes
no
I: Acquire dedicated background image from the same optical setup
II: Average of multiple sample images from the same optical setup
III: Use standard MATLAB method to balance non-uniform illumination
IV: Simple matrix-based approach to balance non-uniform illumination.

Fig. 1. Decision tree of which image processing could be chosen, depending on what background data is available.

4. Results

4.1 Acquisition of background images

In any microscopic analysis, it is typically easy to obtain background data. For example, taking a dedicated background image (**I**) of a blank glass slide (i.e. without any sample). Ideally, the settings for the lamp brightness and camera exposure would be the same as for taking pictures of samples (such as a cell, a particle or bubble). A normalization step (see Table 1) may be performed during image processing to remove differences in overall brightness and the gamma value.

Table 1. General MATLAB and Python scripts for illumination balancing of brightfield images

Step	MATLAB	Python with numpy and opencv2

Loading the background and sample image into MATLAB or Python.	background = imread('bg.tif');
	sample = imread('sample.tif');
		import cv2 as cv2
		import numpy as np
		Background = cv2.imread('bg.tif')
		Sample = cv2.imread('sampl.tif')

Converting the images to 2-D matrices	BG = im2double(background);
	SAMPLE = im2double(sample);
		BG = np.matrix(Background)
		SAMPLE = np.matrix(Sample)

Getting the average brightness value of the background and sample images	lvlBG = median(BG(:)];
	lvlSAMPLE = median(SAMPLE(:));
		lvlBG = np.mean(Background)
		lvlSAMPLE = np.mean(Sample)

Creating normalized versions of the images	nBG = BG .* lvlSAMPLE;
	nSAMPLE = SAMPLE .* lvlBG;
		nBG = BG * lvlSAMPLE
		nSAMPLE = SAMPLE * lvlBG

Calculating the image as resulting from even illumination	RESULT = nSAMPLE - nBG;	RESULT = nSAMPLE - nBG
Maximizing the contrast I: setting the lowest value to zero	RESULT =RESULT - min(RESULT(:));	RESULT = RESULT - np.min(RESULT)
Maximizing the contrast II: stretching the values from zero to one	RESULT = RESULT ./ max(RESULT(:));	RESULT =RESULT / np.max(RESULT)

Image processing steps and the respective MATLAB and python code examples. The results can be seen in Fig. 2. For ease of use we color-coded: loops, comments and filenames.

Often, the background data is obtained with different exposure times or different lamp powers, and thus, the average grey value of the pictures can be different [Figs. 1(a) and 1(b)]. Multiplying the sample image [Fig. 1(a)] with the average grey value of the background image [Fig. 1(b)] and vice versa normalizes the images [Figs. 1(c) and 1(d)]. Subsequently, subtraction of the normalized background [Fig. 1(d)] from the normalized sample image [Fig. 1(c)] yields an image with balanced illumination [Fig. 1 (e)]. To further improve the image, we can increase the contrast by first subtracting the matrix's lowest value from all the pixel's values and subsequently dividing all values by the highest value in the matrix. The resulting matrix has values which range from 0 (full black) to 1 (absolute white) as shown in [Fig. 1(f)]. Note, the codes for MATLAB and Python corresponding to these operations can be found in Table 1.

If it is not feasible to obtain a blank background image (e.g. one may be working with historical data sets), then it is possible to emulate (**II**) by recording multiple (hundreds) images of samples, collected with the same optical setup and then average them using e.g. Fiji (ImageJ) and its built-in feature Z-project (Image > Stack > Z-project) [12].

Similarly, but more cumbersome, images can be imported into bitmap-software like GIMP or Photoshop as layers and then be averaged. Both Matlab and Python could be used as well, by importing all relevant images, converting them to matrices, which can then be averaged. The random distribution of objects throughout the stack of images would ideally cancel out any bright and dark spots on the individual sample images. With the background image acquired, it is possible to balance nonuniform illumination, as described in Table 1, with the result shown in Fig. 2.

4.2 Image processing

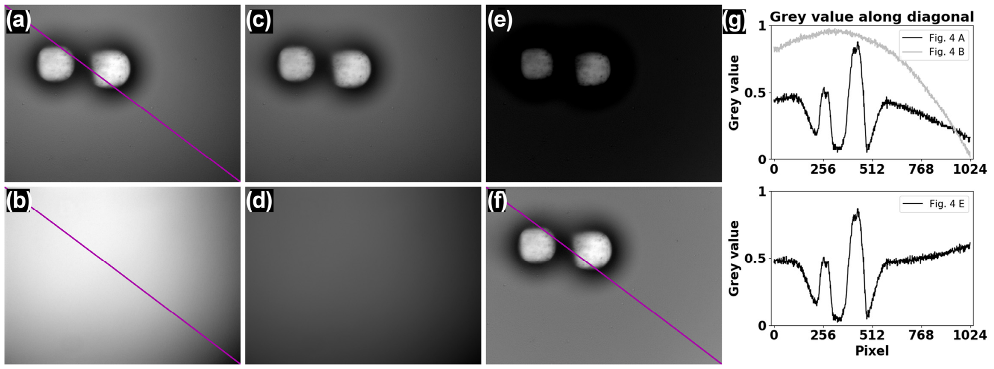


Fig. 2. The process of balancing uneven illumination with a background image of the same optical setup: (a) the original sample image, taken in a bright field setup of Koehler illumination, with obvious vignetting and a sample of high optical density. (b) background image, acquired using the same illumination with a sample-free stage. The brightness levels are evident to an uneven illumination with the brightest point to the left of the center, exhibiting a 2-dimensional normal distribution. (c) & (d) the sample image and the background image with normalized brightness, as interim results of the process. (e) interim result of subtracting the normalized background image (d) from the normalized sample image (c). (f) The final result, where the contrast has also been maximized. The sample now appears on a completely evenly lit background while all the details within the sample have been retained. (g) Graph depicting the grey value along the top-left to bottom-right diagonal (magenta line) of the original sample image [Fig. 1(a), top, black], the background image [Fig. 1(b), top, grey], and the resulting image [Fig. 1(f), bottom, black]. We chose to use a diagonal line for the image analysis to include both the center and the edges of the image, as well as a section of the sample and sample free parts of the image. Results using MATLAB and Python are essentially the same (see Fig. 5 for MATLAB results, Python results are shown here).

If, however, it is not possible to acquire a background (i.e. both (**I**) and (**II**) are not viable), for example, because the objects on the available sample images are not distributed randomly but always centered, or because there is no background data available, there are options for image processing that remain. The route which is generally recommended for MATLAB is shown as (**III**) [13], an approach which “blurs” the image to obtain a background image. This method generally works well for small objects that are evenly distributed over an entire image and do not introduce large patches of brighter or darker background.

For images, that do not meet these requirements, we present option (**IV**), as an alternative way to balance nonuniform illumination, by generating a background image *de novo* (see

Table 2), a technique that is especially suitable for fluorescence microscopy. The method works by selecting the brightest line from the fluorescence image [Fig. 3(a)] and smoothing it into a single vector. This vector represents the brightness (= the grey value) of each pixel along the brightest lines of the fluorescence image [Fig. 3(b)]. By multiplying the vector with itself (using the outer product) we generate a matrix which simulates the non-uniform illumination of the fluorescence image [Fig. 3(c)]. This can be inverted to compute a second matrix, which will cancel out the non-uniform illumination [Fig. 3(d)]. Simple multiplication of the normalized background with the sample image (see Table 3) provides a result which represents the original image taken with perfectly even and uniform illumination [Fig. 3(e)].

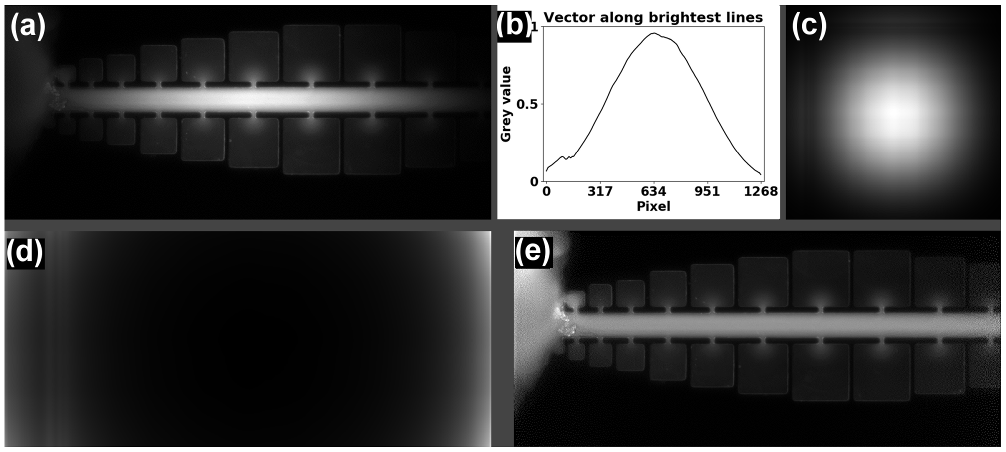


Fig. 3. The process of extracting the background image of an uneven, yet symmetrically illuminated image. (a) the original image of a solution of fluorescent rhodamine B in a microfluidic channel [10]. (b) the extracted grey value along the brightest lines of the image's long dimension, smoothed to avoid artefacts. (c) radially symmetric illumination matrix, which was obtained by multiplying the values shown in Fig. 3(b) with themselves. This is a reconstruction of the spatial intensity of the illumination source. (d) The creation of a matrix mask which can be used to cancel out the uneven illumination of the light source, fitted to the original image. (e) The resulting image of applying the mask to the original image. This image shows how the original image would have looked, had the light source had a perfectly even spatial light distribution. Results using MATLAB and Python are essentially the same (see Fig. 6 for MATLAB results, Python results are shown here).

Table 2. Algorithm to obtain background image from a single grey-value line measurement

Step	MATLAB	Python with numpy and opencv2

Loading the black and white sample image into MATLAB or Python.	sample = imread('sample.tif');
	SAMPLE = im2double(sample);
	[nx, ny] = size(SAMPLE);
		F = cv2.imread('sample.tif', -1)
		FCM = np.matrix(f)
		nx, ny = np.shape(f)
		X = np.linspace(1,nx,nx)
		XMax = np.linspace(1,nx,nx)

Taking the brightest line along the image's long axis as a vector.	x = SAMPLE(:,1);
	For i = 1:nx
	    x [i] = mean(SAMPLE(i,:)
	end
	k = find(x == max(x(:)]);
	xm = SAMPLE(:,k)
		for n in range(nx):
		    X[n] = np.max(FCM[n,:])
		for n in range(nx):
		    XMax[n] = np.sum(FCM[n,:])
		m = np.max(XMax)
		xm = [i for i, j in enumerate(XMax) if j == m]
		#Alternatively: finding the median of the top 90% value
		#xa = [i for i, j in enumerate(XMax) if j > m*0.9]
		v = np.linspace(1,ny,ny)
		for i in range(ny):
		    v[i] = np.mean([FCM[xa,i]])

Smoothing the vector reduces artefacts along the way.	s = smooth(v, 9)
	%MATLAB can automatically smooth over a given amount of adjacent values (e.g. 9) along a vector (e.g. v). Increasing the number (e.g. 100), gives a smoother appearance in MATLAB.
	%For Python however, we needed to write a code snippet:
		s = v #dedicated smoothed vector
		#for values with 4 neighbours to both sides:
		for i in range(4,ny-4):
		    s[i] = np.mean([v[i-4],v[i-3],v[i-2],v[i-1],v[i],v[i+1],
		    v[i+2],v[i+3],v[i+4]])
		s[0] = np.min(s[0:4])
		s[ny-1] = np.min(s[ny-4:ny-1])
		#for the first 4 values, we use a linear regression:
		for i in range (1,4):
		   s[i] = s[0]+i*(s[4]-s[0])*.2
		#for the last 4 values we also use linear regression:
		for i in range (ny-1,ny-4):
		   s[i] = s[ny-1]+i*(s[ny-4]-s[ny-1])/4

Inverting the vector.	si = 1./s;	si = 1/s;
Multiplying the vector with itself to generate a radial symmetric matrix of the image's illumination.	M = si' .* si %The order is important for MATLAB. % si .* si' ≠ si' .* si	M = np.outer(si,si)
Cutting the matrix to the size	BG = M(round((ny-nx+1)/2):ny-round((ny-nx+1)/2)+1,1:end);	BG=np.matrix(M[round((ny-nx)/2):ny-round((ny-nx)/2)+1,])

This table is to guide the user through the steps needed to generate a matrix that represents the nonuniform illumination in a sample picture, along with examples in both MATLAB and python to arrive at the same result. The greater library of built-in function in MATLAB ensures a leaner code. For ease of use we color-coded: loops, comments and filenames. Full code available at [11].

Table 3. General MATLAB and Python scripts for illumination balancing of fluorescence images

Step	MATLAB	Python with numpy and opencv2

Loading the background and sample image into MATLAB or Python.	background = imread('bg.tif');
	sample = imread('sample.tif');
		Background = cv2.imread('bg.tif')
		Sample = cv2.imread('.tif')

Converting the images to 2-D matrices	BG = im2double(background);
	SAMPLE = im2double(sample);
		BG = np.matrix(Background)
		SAMPLE = np.matrix(Sample)

Creating normalized version of the background	nBG = BG ./ max(BG(:)];	nBG = BG * lvlSAMPLE
Calculating the image as resulting from even illumination	RESULT = nSAMPLE.*nBG;	RESULT = nSAMPLE * nBG

Image processing steps and the respective MATLAB and Python code examples. The results can be seen in Fig. 3. For ease of use we color-coded: loops, comments and filenames. Full code available at [11].

A direct comparison between the MATLAB standard procedure (**III**) and our "matrix-mask" method (**IV**) shows a significant difference in the quality of balancing out the nonuniform illumination. If the same fluorescence image used for the approach described above (**IV**) is treated using (**III**), the results [Fig. 4] show an improved distribution of grey values all over the image but it introduces artefacts, especially around areas with a large variance of grey values within small areas [Fig. 4(e)]. We extracted the grey values along the brightest lines in the center (N = 19) and plotted them over their x-position in the image, for both our newly proposed approach **IV** [Fig. 3(c)] and the MATLAB standard procedure **III** [Fig. 4(f)]. We also calculated the standard deviation of these plotted vectors [Figs. 4(c) and 4(f)] to be 0.31 for the original image (uncorrected), 0.11 for the MATLAB approach (**III**), and less than 0.01 for our "matrix-mask" approach (**IV**), which correspond best to the constant concentration of fluorophore within the microfluidic channel.

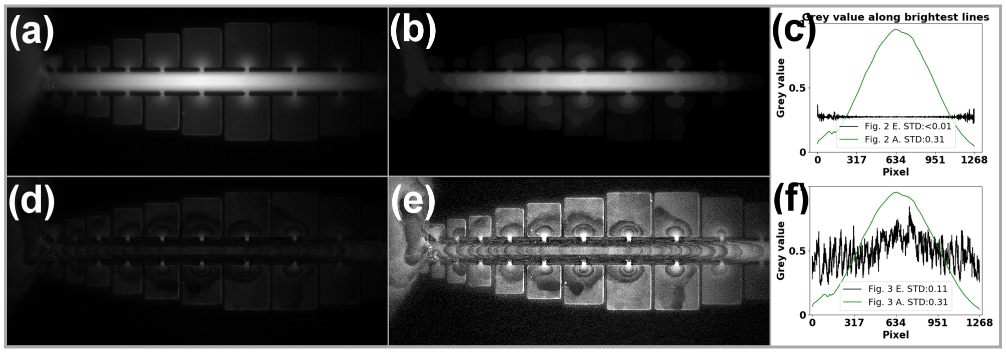


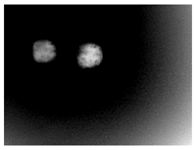
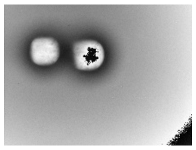
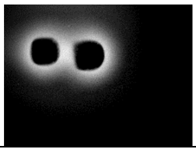
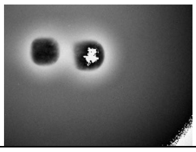
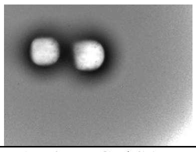
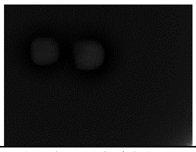
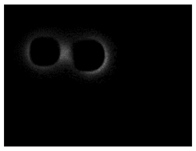
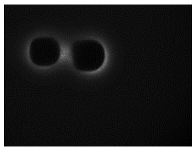
Fig. 4. Balancing of nonuniform illumination using the recommended procedure for MATLAB [13] for the same fluorescence image as in Fig. 2. (a) the original image of fluorescent rhodamine B in a microfluidic channel [10]. (b) The background image that was calculated by MATLAB. (c) The extracted grey value along the brightest lines of the original image (green, [Fig. 3(a)]) and of the corrected illumination image (black, [Fig. 3(e)]), analogous to Fig. 3(b). (d) The image, which results from subtracting the background image Fig. 4(b) from sample image Fig. 4(a). (e) The final result after maximizing the contrast. (f) The grey values along the same lines as above for Figs. 4(e) (black) and 4(a) (green) for comparison.

It is also possible to use this new "matrix-mask" approach to process images for which background data is available. The background image [Fig. 4 (b)] can be used to create a matrix for the illumination, which is typically nonuniform, while the sample image is also converted into a second matrix; each pixel in the images generates one value in the corresponding matrix, whilst the pixel's grey value is represented in the value of the individual data points.

4.3 Alternative matrix-based approaches

The presented method proved to be the best and most reliable matrix-based approach using both MATLAB and Python. There are shorter operations that can be carried out in order to balance out uneven illumination, which are intuitively correct, but lead to sub-optimal results (for example, simple pixel-by-pixel subtraction or division of the background image from the sample image, or vice versa). Since these operations will result in mostly black images with poor contrast, a contrast maximization akin to the one presented on the bottom of Table 1 was carried out right afterwards. In Table 4, we have compiled all of these operations, the algorithms needed to perform them in both MATLAB and Python, together with the results obtained from these operations, demonstrating that our Python bundle with numpy and the opencv2 module often automatically performs a contrast maximization, especially after subtractions.

Table 4. MATLAB and python scripts for worse matrix approaches to illumination balancing

Bad solution	MATLAB	Python with numpy and opencv2
Simply subtract the background from the sample image, followed up with contrast maximization to brighten up the dark result (MATLAB only): The contrast maximization always follows the route BAD2 = BAD2-mean(mean(BAD2)]; BAD2 = BAD2./max(max(BAD2)]; *the opencv2 module with numpy automatically does the contrast maximization after + / - operations leading to different results	BAD1 = SAMPLE – BG;  MATLAB contrast maximization always follows the route: BAD = BAD-mean(BAD(:)]; BAD = BAD ./ max(BAD(:)];	BAD1 = SAMPLE – BG  The python contrast maximization always follows the route: BAD = BAD – np.min(BAD) BAD = BAD / np.max(BAD)
Subtracting the sample image from the background image followed up with contrast maximization to brighten up the dark result (MATLAB only) resulted in a worse resolution, contrast and blur.	BAD2 = BG – SAMPLE; 	BAD2 = BG – SAMPLE 
A pixel-by-pixel division of the sample image by the background image resulted in a generally dark image, dominated by a bright white edge.	BAD3 = SAMPLE ./ BG; 	BAD3 = SAMPLE / BG 
Dividing the background image by the sample image resulted in an overly white image with a dark corner. A subsequent normalization resulted in a restoration of the original image, yet with inversed coloration and increased noise artefacts.	BAD4 = BG ./ SAMPLE; 	BAD4 = BG / SAMPLE 

Alternative approaches to balance out uneven illumination might feel intuitive yet lead to worse results. Due to different internal handling of the matrices, the same operations can lead to different results for MATLAB and Python, as shown above.

5. Discussion

The results shown in Fig. 3 demonstrate the effectiveness of this simple matrix-based approach to balance out uneven illumination, and also show, that it is possible to extract the illumination information from a single line of one image. The quality of the obtained illumination matrix can be further improved by averaging over several lines – preferably through the point of highest light intensity – and by additional smoothing to avoid the introduction of artefacts [Fig. 3]. This approach, however, only works when the sample has the same brightness value along the selected line (e.g. due to the same amount of fluorophore present along the line). In the sample we chose, some artefacts to the left of the channel result in additional reflection and thus a non-normal distribution in the extracted vector [Fig. 3(b)] and ultimately the illumination mask [Fig. 3(d)]. The ideal solution for balancing out uneven illumination in micrographs is nonetheless, first taking a dedicated, sample free background image with the same optical parameters and subsequently performing the recipe shown in Table 1 [see also Fig. 2]. Nonetheless, our approach yielded better results than the gold-standard MATLAB procedure [Fig. 4], and had so few artefacts that the grey value of the image could be taken to visualize the concentration of fluorophore within the device (see supplementary data of [10]).

Additionally, other matrix-based approaches have been tested and shown to yield worse results in regards of balancing out uneven illumination, as shown in Table 4. For example, simply subtracting the background image from the sample image (see Table 4, "BAD1"), or vice versa (see Table 3, "BAD2") leads to images with poor contrast, loss of details and introducing artefacts. In the results obtained from Python, artefacts were found where the sample image was brighter than the background image, in unexpected switching of black and white.

Both, commercial MATLAB and freeware Python, offer solutions that ultimately lead to comparable results. Differences in the resulting images (see Table 4) can be explained by the different algorithms MATLAB and Python use in their cores for handling the matrices, which represent the images. As an example, there are salient spots in the first two right-hand side images in Table 4, areas of bright white surrounded by darkness and areas of deep black in a generally bright area. These black/white inversions could stem from misinterpreting negative numbers or a data compression step, where the matrices are no longer handled point-by-point but segmented to safe calculation space.

With the correct coding, however, both programs can be used to greatly improve image quality of micrographs, be it with dedicated background data, or by extracting illumination information from the sample images themselves and applying our matrix-mask method. This opens up many possibilities, including fast and easy balancing of sample micrographs illumination, or of using the grey value of a fluorescent image to measure the concentration of a fluorophore at any position in fluorescence micrographs.

Appendix A.

MATLAB processing results, for comparison with Figs. 2 and 3.

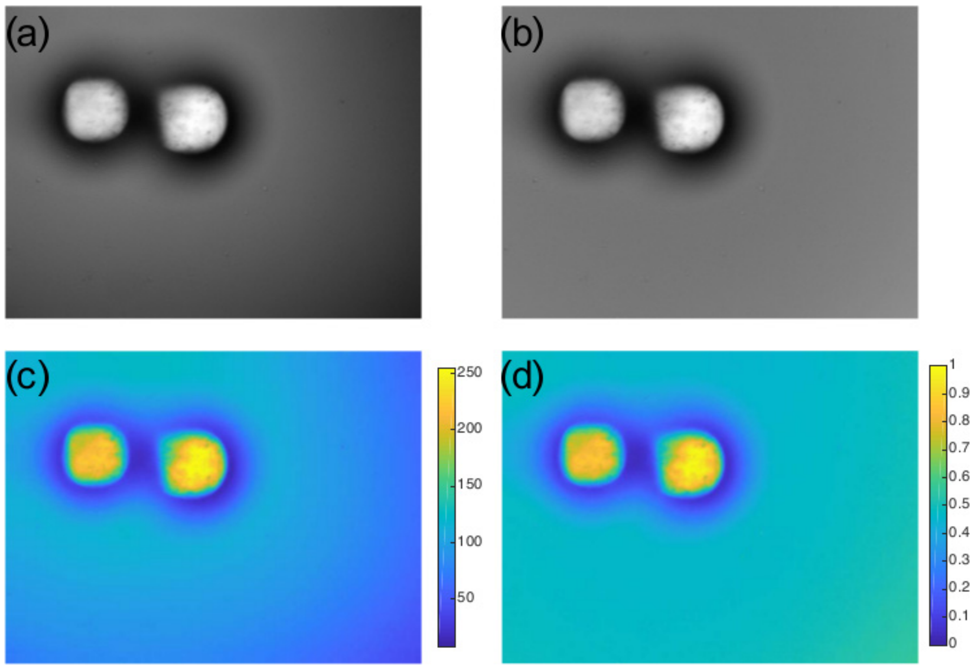


Fig. 5. The figure shows MATLAB processing results of Fig. 2(a), comparable to the python processing shown in Fig. 2. (a) before (b) after the processing. (c) & (d) are pseudo-colored versions of (a) and (b).

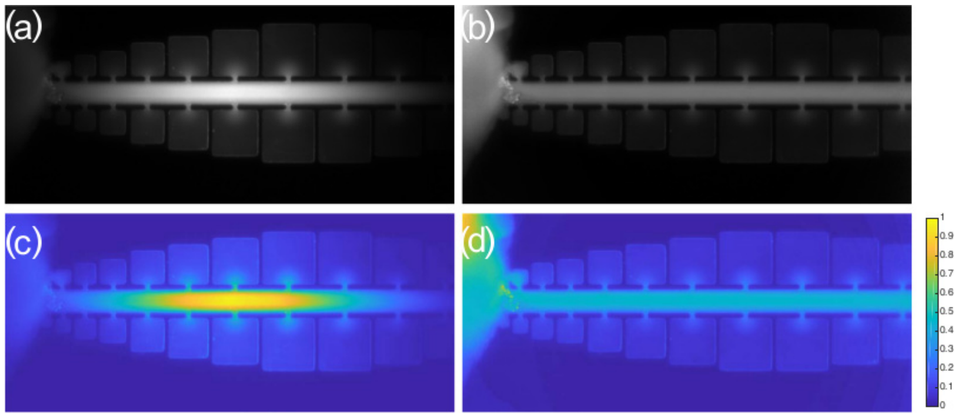


Fig. 6. The figure shows MATLAB processing results of Fig. 3(a), comparable to the python processing shown in Figs. 3(a) and 3(e). (a) before (b) after the processing. (c) & (d) are pseudo-colored versions of (a) and (b).

Funding

EPSRC (EP/K027611/1) and ERC Advanced Grant 340117.

Acknowledgments

AH worked on it in his free time, using freeware and a beta developer testing version. Jon Cooper acknowledges EPSRC (EP/K027611/1) and ERC Advanced Grant 340117. Axel Hochstetter would like to thank Thomas Pfohl for fruitful discussions.

Disclosures

The authors declare that there are no conflicts of interest related to this article.