Published in Image Processing On Line on 2022–12–30.
Submitted on 2022–10–07, accepted on 2022–10–07.
ISSN 2105–1232 © 2022 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2022.438

Image Unprocessing: A Pipeline to Recover Raw Data from sRGB Images

Valéry Dewil

Université Paris-Saclay, ENS Paris-Saclay, Centre Borelli, Gif-sur-Yvette, France
valery.dewil@ens-paris-saclay.fr

Communicated by Jean-Michel Morel *Demo edited by* Valéry Dewil

Abstract

Access to high quality datasets is an essential condition for data-driven methods as it is known that mismatches between the distributions of training and test data may cause learning-based methods to fail. This issue has led to one of the most active research subjects in learning-based image restoration. For instance neural networks trained on *unrealistic* synthetic data may not generalize to real data even if they perform well on those synthetic data. This is specially problematic for image and video processing tasks, such as denoising, which are performed on raw data, since acquiring real raw datasets is not straightforward and is even impossible in some cases (acquiring a video dataset of real noise with clean ground-truth, for instance). Consequently, CNNs are often trained on synthetic data. Synthesizing *realistic* raw data is a difficult task and requires to invert properly the image processing pipeline. This paper focuses on the backward pipeline proposed by Brooks et al. [Unprocessing images for learned raw denoising, CVPR 2019] which aims at producing raw data from sRGB images.

Source Code

The source code and documentation for this algorithm are available from the web page of this article[1]. Usage instructions are included in the `README.txt` file of the archive. The original implementation of the method is available here[2].
This is an MLBriefs article, the source code has not been reviewed!

Keywords: unprocessing; raw; pipeline

[1]https://doi.org/10.5201/ipol.2022.438
[2]https://github.com/timothybrooks/unprocessing

1 Introduction

After acquisition, raw images (or videos) are processed by a pipeline which aims to render the final sRGB output. The pipeline stages, as well as the order, differ between camera-makers or softwares, but all of them share the same main essential elements such as demosaicking, denoising, color correction, tone-mapping, etc.

However in image or video processing, many tasks are performed in the raw domain (demosaicking or denoising tasks, for example).

In recent years, data-driven methods have proved very efficient in many image processing tasks for which they set the current state of the art. In these methods, neural network (NNs) are trained with supervision on large datasets consisting of clean-noisy pairs. Although a very large amount of sRGB images and videos can be found easily on the internet, there are unfortunately not so many datasets for raw data. Consequently, most data-driven methods are trained on synthetic data. Of course, the performance of the NNs at test time highly depends on the data used for training. The distribution of the training and test (on which the evaluation is done) data should be as close as possible.

Generating *realistic* raw data from sRGB images requires to invert *properly* the processing pipeline. In 2018, Brooks et al. [2] trained a denoising CNN (convolutional neural network) for the task of raw denoising with real noise. They proposed a reverse pipeline for generating realistic raw images by "unprocessing" sRGB ones. In this paper, we describe the unprocessing method introduced in [2] and show some results obtained with it. The next section is dedicated to the methodology and the technical details for the inversion of the pipeline. In Section 3, we give the pseudo-code of the method as it is used in the demo. Results of the method are presented in Section 4. Finally in Section 5, we present the demo itself and we show how to use it. This includes a brief discussion about the parameters of the demo.

2 Method

The unprocessing pipeline proposed by Brooks et al. consists of a few main steps. In Figure 1, we summarize them in the order they appear in the procedure. For each step, we provide an example image of the output.

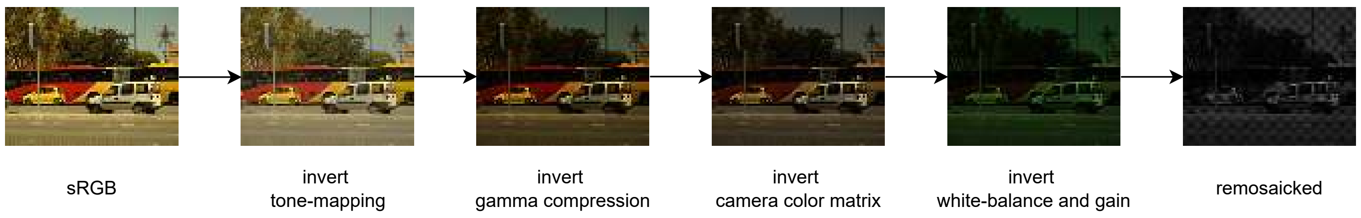


Figure 1: Overall unprocessing pipeline: visualization of the typical output of each unprocessing step. Each step takes as input the output of the previous stage.

Basically, the pipeline consists of:

1. Inverting the tone-mapping curves.
2. Inverting the gamma compression.
3. Inverting the color matrix multiplication (from RGB to the camera color-space).
4. Inverting white-balance and brightening.

5. Mosaicking the 3 channels image.

Starting with a clean sRGB image, this pipeline produces a realistic clean raw image. In the demo, we also include the possibility to add synthetic but realistic real noise. The next paragraphs detail the pipeline steps in more depth: we start by explaining how each step is operated in the forward processing pipeline and then how it can be inverted in the unprocessing pipeline.

Tone-mapping. In an image processing pipeline (from raw to sRGB), the final output is rendered with a contrast enhancement curve. Although some *local* enhancements can be done, we consider in this step a *global* tone-mapping. Basically, the purpose of the global tone-mapping is to stretch the dynamics by pushing up the high values while darkening the dark areas further. This curve (sometimes denoted as *"S-curve"*) can be approximated by a cubic polynomial. For images between 0 and 1, an approximation can be $x \mapsto -2x^3 + 3x^2$ [2], which has the reciprocal function

$$x \mapsto \frac{1}{2} - \sin\left(\frac{\operatorname{asin}(1-2x)}{3}\right).$$

Therefore, this reciprocal function can be directly used for the unprocessing pipeline. In Figure 2, we show the stretching effect (or crushing effect for the inverse) of the global tone-mapping.

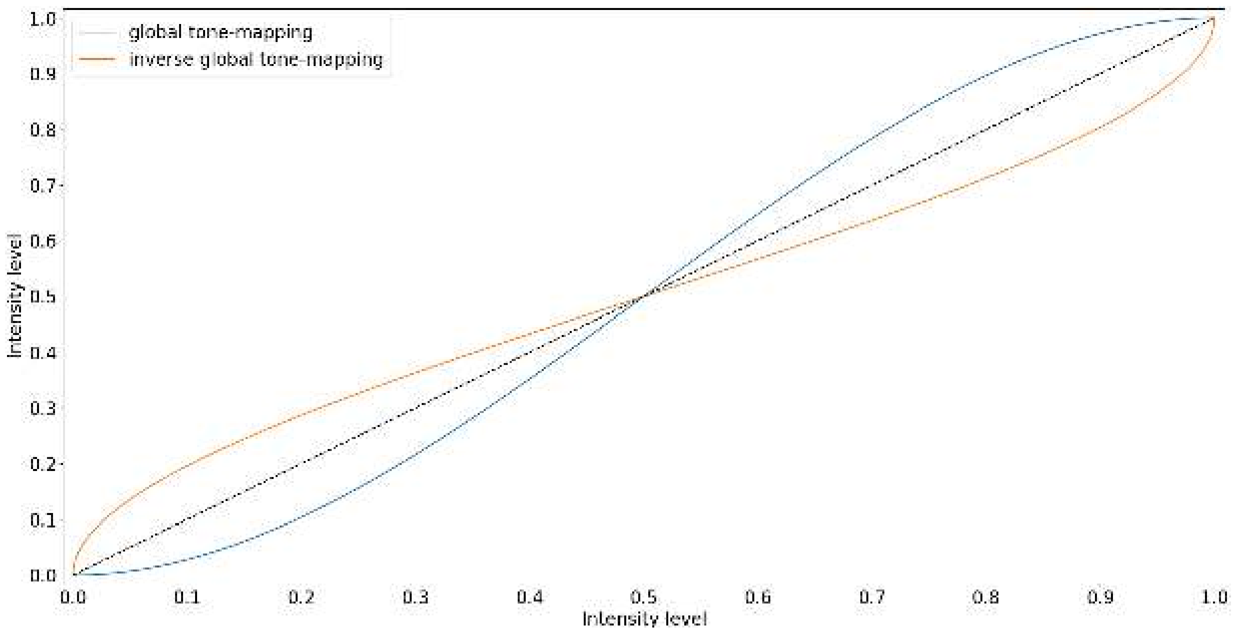


Figure 2: Illustration of how the global tone-mapping (respectively its inverse) stretches (respectively crashes) the dynamic range.

Gamma compression. Human perception of brightness is not a linear function of the luminosity level. It has a higher sensitivity to small changes of brightness in the dark areas than in the lightest ones. Furthermore, two times more light is only perceived as slightly brighter. On the other hand, cameras have a linear perception since in this case, the signal output by the camera is simply doubled when two time more photons hit the sensor. In order to compensate that difference of behavior, a *gamma correction* (also called gamma compression or sometimes gamma encoding) is performed: basically, it ensures that the images taken by the camera appear as they do to the human eye. Another advantage is that, when images are stored with quantization (finite number of levels), it optimizes the brightness encoding by using more levels to encode the low intensity range of the dynamics (for which the human eye is more sensitive to changes), leading then to more details in the dark areas.

The gamma corrected image is simply obtained from the linear one by applying a power function $v = u^{\frac{1}{\gamma}}$ where v is the gamma corrected image and u is the linear one. In the unprocessing pipeline, the goal of this step is to go back to the linear space and the inversion is straightforward: $u = v^{\gamma}$. In practice, a standard value of $\gamma = 2.2$ is usually used [2, 8] and data are clipped to 0 to avoid negative values.

Figure 3 shows the stretching of the dynamic range for an image $u \in [0.255]$, and one example of a linear and a gamma corrected gradient of luminosity.

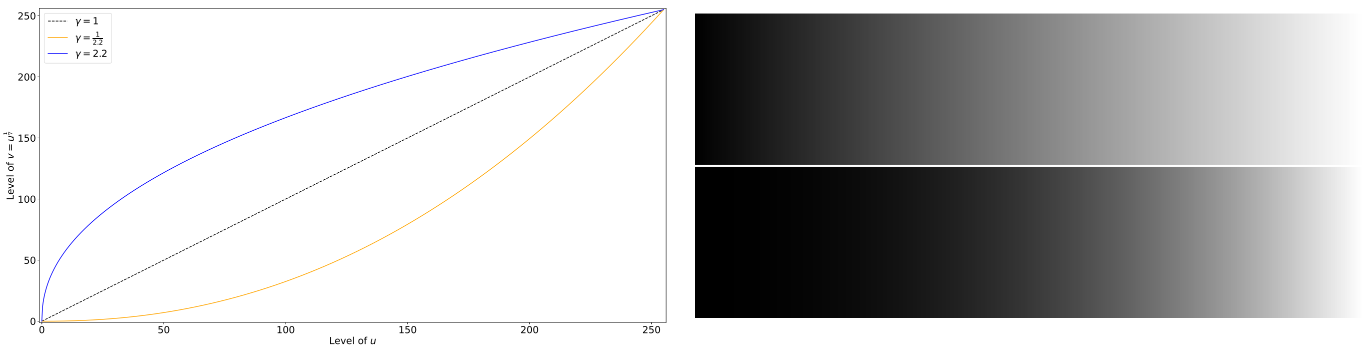


Figure 3: Left, illustration of the gamma stretching of the dynamics. Top right, the linear dynamics $[0, 255]$. Bottom right, the same dynamic range stretched with $\gamma = 2.2$.

Camera Color Matrix. Usually, the final rendered color space after the processing pipeline is different from the camera color space. The latter is camera dependent. For converting from the internal camera color space to the sRGB one, camera-makers have tuned a 3×3 invertible matrix, called the *"Camera Color Matrix"* (CCM). This matrix depends on the camera. The Darmstadt Noise Dataset [8] is a dataset of real raw images taken with four different consumer cameras for which the CCM matrix is available. In [2], the authors trained a denoising network which was afterwards evaluated on this dataset. In order to yield a good generalization, they synthesized realistic raw data by simulating a camera which has a CCM as a random convex combination of these four CCMs. In the demo, we propose either to apply this randomly synthesized CCM or any of these four consumer cameras. For inverting the color space change, once the CCM has been sampled (either randomly computed or as a fixed and pre-selected one), the CCM is inverted and applied to the intermediate image in the unprocessing pipeline.

White-balance and digital gain. In the processing pipeline, the image exposure is corrected using a *global* digital gain (which is applied to all the channels). Unfortunately, the value of this gain is image-dependent. In [2], the authors estimated an average value by looking at the digital gain on the Darmstadt Noise Dataset and found the value 0.8. When they trained their denoising network, they sampled a gain from a normal distribution centered on this value with a small standard deviation of 0.1, leading to roughly spanning the interval $0.8 \pm 3 \times 0.1 = [0.5, 1.1]$. In the demo, we propose to select the global gain (called g^{rgb}) in this range.

Channel-wise gains are also applied in the processing pipeline for balancing color channels together. Basically, the goal of a white balance is to display areas that look white when seen by humans in the scene as white in the screen thus balancing any color cast due to the illumination source light. Furthermore, the white balance compensation not only depends on the scene (the real colors in the scene) but also on the light condition, thus it has to be set individually for each image.

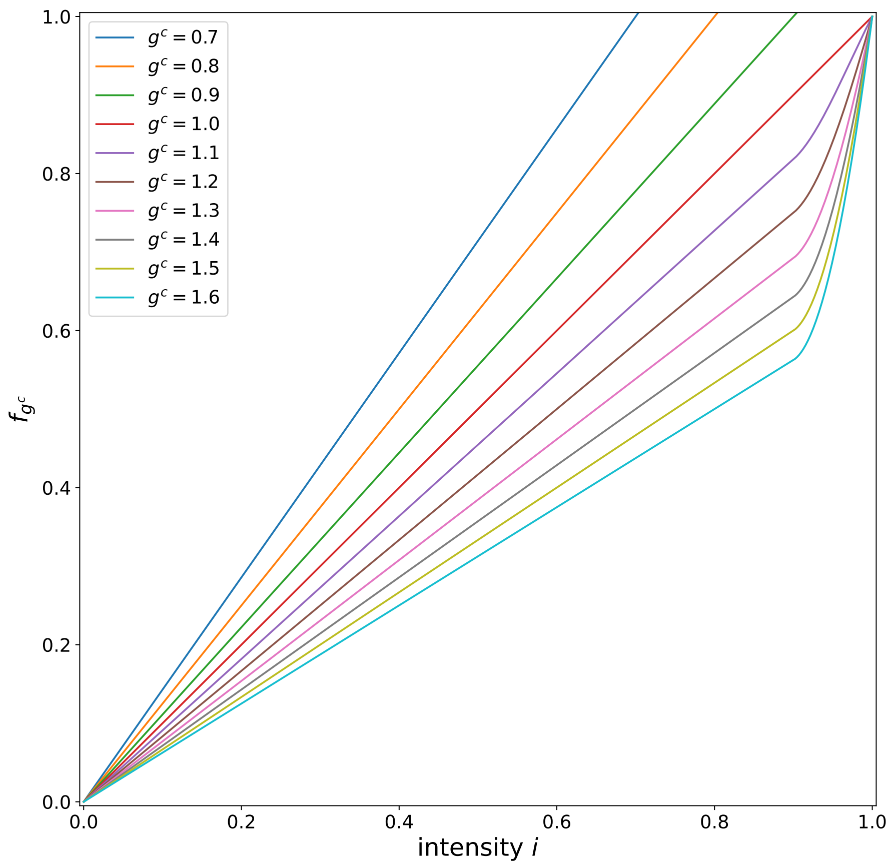


Figure 4: Graph of the function f_{g^c} (x, g). This function is applied channel-wise. It allows to create saturation even when the inverse gain of a given channel c is smaller than 1, by pushing up the values toward saturation (for intensities $i \geq 0.9$).

Since a global gain (applied to all the channels) is used, instead of having three channel-wise gains (for the red, green and blue channels), we can formulate the color balancing for only two channels as a ratio with respect to the third channel (set as a reference channel). The green channel has often higher intensities than the others (it is in the green wavelengths that the sun emits most of its radiation, and nature reflects a rather green hue). Therefore the green channel is set as reference and a white-balance algorithm estimates the gains which need to be applied to the red and blue channels (equivalent to fix the green gain to 1). When the gains used are unknown, inverting this is a very difficult task. Instead, Brooks et al. proposed to estimate an average red and blue gain on the Darmstadt Noise Dataset for which the gains have been saved in the metadata. They determined that the red gain g^{red} spans the range $[1.9, 2.4]$ and the blue gain g^{blue} spans $[1.5, 1.9]$. They sampled white-balance gains uniformly in their respective interval for synthesizing their training data. In the demo, the user can specify the value of these gains in the corresponding range.

Finally, for inverting the color balance in the unprocessing pipeline, we divide instead of multiplying by the color balance coefficients of each channel: $[\frac{1}{g^{red}g^{rgb}}, \frac{1}{g^{rgb}}, \frac{1}{g^{blue}g^{rgb}}]$. However, given the range values of g^{rgb}, g^{red} and g^{blue}, the per-channel inverse gain (of the form $1/g$) is sometimes smaller than one, and consequently the intensity in channel c is likely to be decreased after the inverse gain $1/g^c$. As a result, the synthesized raw image is unlikely to contain any highlights or saturated areas and may be therefore in some ways *unrealistic*. In [2], the authors used the inverse gain $1/g^c$ if it is greater than one and then does not prevent the creation of saturated areas. Otherwise, they applied it only if the intensity i was smaller than 0.9 (for images in the range $[0, 1]$), and if not (then close to saturation) they used a cubic polynomial function which is 1 for intensity 1 and is $0.9/g^c$ for intensity 0.9 to ensure continuity. Each intensity i of channel c with inverse gain $1/g^c$ is then transformed with the continuous function

$$f_{g^c} : i \mapsto \begin{cases} \frac{i}{g^c} & \text{if inverse gain } \frac{1}{g^c} \geq 1 \\ \frac{i}{g^c} & \text{if inverse gain } \frac{1}{g^c} < 1 \text{ and intensity } i \leq 0.9 \\ \frac{i}{g^c} + 100i(i-0.9)^2\left(1-\frac{1}{g^c}\right) & \text{if inverse gain } \frac{1}{g^c} < 1 \text{ and intensity } i \geq 0.9. \end{cases}$$

Note that in [2], the value of 0.9 is a threshold value that the authors determined to add enough highlights. Figure 4 shows a plot of this function for the intensities in the range $[0, 1]$.

Remosaicking. In a camera sensor, each pixel is like a single intensity sensor which counts the number of photons (the light intensity) that hit it during the exposure. The count is independent of the color and there is only a single counting per pixel: a *color filter array* (CFA) is placed in front of the sensor, determining the intensity of the light for one given color per pixel. The CFA consists in a mosaic of color filters (either *red*, *green* or *blue*), arranged in a Bayer pattern such as RGGB, for instance. This results in a single channel output with sparse color samples. In the processing pipeline, the 3 channels (RGB) image is computed by interpolating the missing color channel [5]. Consequently, in the unprocessing pipeline, we need to remosaick the RGB image, that is, undo this interpolation. This can be done straightforwardly by omitting two thirds of the color values according to the Bayer pattern.

Simulating realistic noise. Real noise has very complex characteristics. Although modeling the distribution of real noise is still a research subject, the sources of noise are well known. Basically, there are two mains sources: the *shot noise* (the photons hitting the sensor) and *read* (or *readout*) *noise* (sensor-based sources of noise). There are other sources of noise like row noise (banding pattern), dark-current noise, thermal noise, quantization noise, dead pixels, non-linear behavior of the sensor (e.g. clipping), etc.

Shot noise is signal-dependent and follows a Poisson distribution [6]. Read noise can be approximated with a Gaussian distribution [4]. A simplifying assumption (which holds especially true in the case of non extreme low-light conditions [9]) consists in modeling both shot noise and read noise with a single heteroscedastic Gaussian noise [7]. Although simplistic, a recent work [3] showed that for normal illumination conditions, a network trained directly on real data or on synthetic data degraded using this noise model has a similar performance.

In the demo, we propose to add heteroscedastic Gaussian noise with variance $\sigma^2 = a \cdot u + b$, where u is the clean image and a, b are two noise parameters pre-selected by the user.

3 Algorithm

The Python code is split in two main functions: unprocessing of noise-free data and adding a realistic noise (optional). Both functions are written in pseudo-code in Algorithm 1. Starting from the RGB image, the noise-free raw output is computed with the function *unprocessing*. If the user wants to add noise, this output is fed to the function *add_noise* to get the noisy raw output.

4 Experiments

In Figure 5, we display an example of the results obtained using the default parameters and to which we add noise. The default parameters are set to produce a realistic result. For some applications,

Algorithm 1: Image unprocessing

```
1  function unprocessing(rgb)
       Input $rgb$: a png 8-bits RGB image
       Param $bl$: black-level of the raw output image
       Param $wl$: white-level of the raw output image
       Param $\gamma$: $\gamma$ value used in the gamma-decompression
       Param camera: camera model used for the CCM coefficients. A convex combination of
                     several cameras can be simulated
       Param $g^{red}$: digital gain for the red channel used in the white-balance inversion
       Param $g^{blue}$: digital gain for the blue channel used in the white-balance inversion
       Param $g^{rgb}$: global gain used for brightening inversion
       Output $raw$: the clean raw unprocessed output
2      $rgb \leftarrow rgb/255$ # put the data in $[0, 1]$
3      $smooth \leftarrow 0.5 - \sin(\arcsin(1 - 2 \times rgb)/3)$ # invert the tone-mapping
4      $linear \leftarrow smooth^{\gamma}$ # invert the gamma compression
5      $\text{CCM} \leftarrow \text{find\_CCM}(\text{camera})$ # load the pre-recorded CCM of the selected camera
6      $linear \leftarrow \text{apply\_ccm}(linear, \text{CCM})$ # apply the CCM matrix to each RGB triplet
7      $color\_balanced \leftarrow \text{invert\_gain}(linear)$ # invert the gains as shown in Section 2
8      $color\_balanced \leftarrow \min(\max(color\_balanced, 0), 1)$ # clip data to $[0, 1]$
9      $Bayer\_CFA \leftarrow \text{mosaick}(color\_balanced)$ # remosaick the 3 channels image
10     $raw \leftarrow Bayer\_CFA \times (wl - bl) + bl$ # switch to the range $[bl, wl]$
11     return $raw$
12 function add_noise(raw)
       Input $raw$: the clean raw unprocessed image.
       Param $a$: signal-dependent parameter of the heteroscedastic Gaussian noise variance.
       Param $b$: constant parameter of the heteroscedastic Gaussian noise variance.
       Output $\tilde{raw}$: the noisy raw unprocessed image.
13     $n \sim \mathcal{N}(0, 1)$ # sample a normal Gaussian distribution
14     $\tilde{raw} = raw + \sqrt{a \cdot raw + b} \times n$ # add heteroscedatic Gaussian noise
15     return $\tilde{raw}$
```

it is interesting to tune the white-balance gain parameters, for instance when the user wants to simulate a specific lighting condition (or knows the color cast of the illumination which produces the sRGB image). However, modifying those gains is a tricky task. As explained in Section 2, applying the inverse gain is not a trivial multiplication, but a function which aims at producing highlights. In Figure 6, we show an example for which the choice of white-balance gains can create many saturations. To produce this example, we set the white-balance gains to the extreme cases with the minimum values ($g^{red} = 1.9, g^{blue} = 1.5, g^{rgb} = 0.85$) (top of the figure) and the maximum values ($g^{red} = 2.4, g^{blue} = 1.9, g^{rgb} = 2.05$) (bottom of the figure).

In Figure 7, we show another example with the clean RGB image and the results of the clean unprocessed images with small and large inverse gains. Again, we used the extreme values. In the sRGB image, the alley on the right and the house on the top left are saturated; the house in the background is close to saturation. After unprocessing, the house on the top left is still uniformly saturated, but the alley contains some discontinuities, due to the highlight-preserving function used to invert the gains. With small inverse gains (inverse gain $1/g < 1$), this function creates a jump toward saturation (on high values of intensity). This means that a small variation of intensity is highly amplified. In this latest case (Figure 7c) the alley contains much more highlights and the

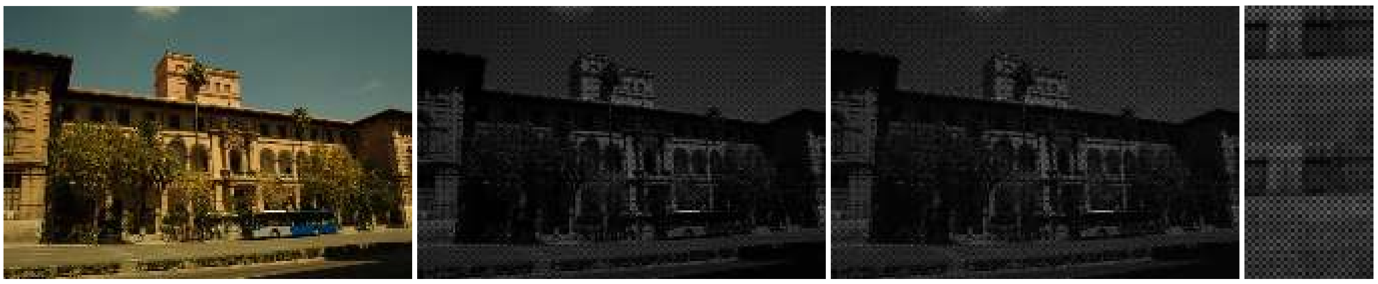
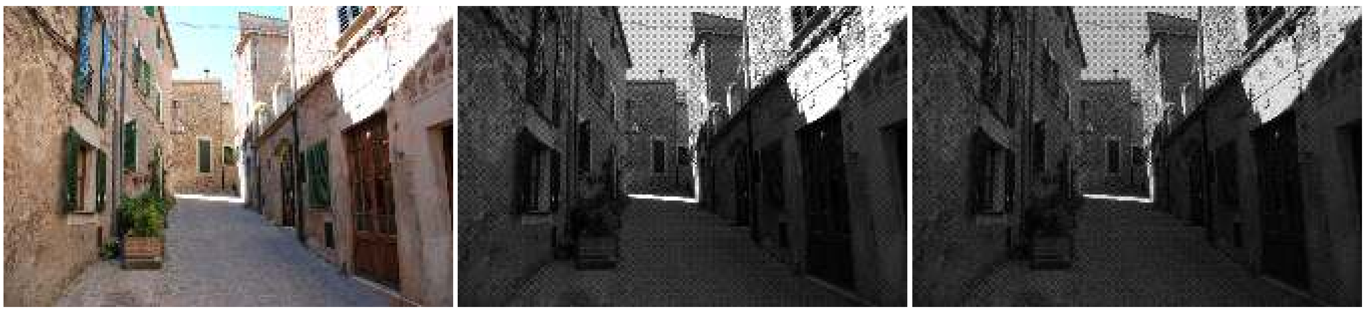

(a) sRGB input (b) clean raw output (c) noisy raw output (d) crops

Figure 5: Unprocessing of a sRGB image with the default parameters of the demo. In (d), we show crops of the clean raw (top) and the corresponding noisy (bottom)


(a) sRGB input (b) clean raw output (c) noisy raw output

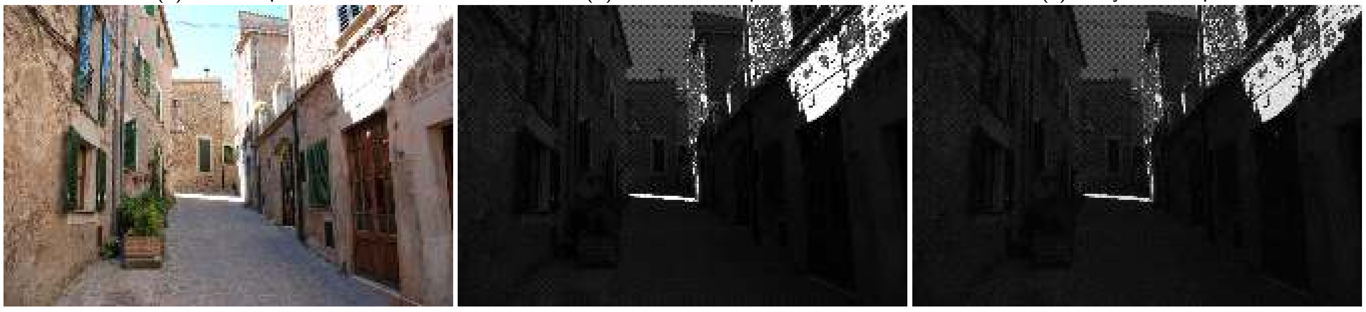

(a) sRGB input (b) clean raw output (c) noisy raw output

Figure 6: Top, the set of gains used does not create more saturation. Bottom, with a different set of gains, more saturation is created.

house in the background has also some highlights. This is wanted by the authors of [2], who designed a function for inverting gain which is likely to produce highlights with small inverse gains. The demo allows to produce this type of results by tuning the gain values. For example, if one wants to evaluate a CNNs on (real) data with highlights, it is possible to create datasets with many highlights. Alternatively, one can generate datasets with few, or even no highlights.

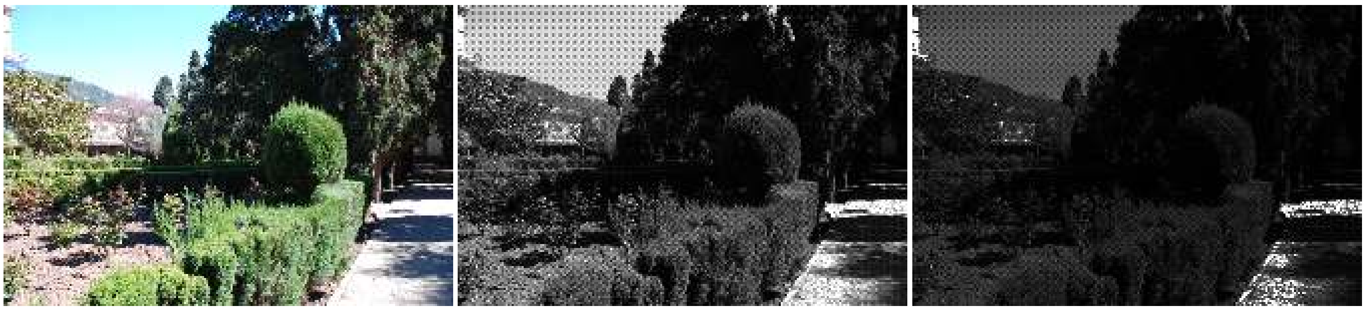

(a) sRGB input (b) clean raw output (large inverse gains) (c) clean raw output (small inverse gains)

Figure 7: Highlights created by the highlight-preserving gain function.

5 Demo

Taking as input a sRGB image saved in 8-bits format, the demo aims to produce a noise-free raw image. Additionally, the user can also choose to add noise using the heteroscedastic Gaussian noise model described in Section 2. For that, the parameters a and b of the variance should be specified. For the default values, we estimated the heteroscedastic Gaussian noise of the sensor IMX385 at ISO level 3200 from the *Captured Raw Video Dataset* (CRVD) [10], which consists of natural and outdoor videos (in particular with real noise). The user is free to change the noise parameters.

The demo displays the input as well as the clean (and noisy) generated raw output. For displaying purposes a contrast stretching is applied to the output, mapping the minimum-maximum interval to the full range. The raw image with Bayer pattern RGGB without contrast stretching can be downloaded in *.dng* format using 16-bits integers [1]. The final range (black-level and white-level) are specified by the user. The downloaded raw images contain metadata such as *width, height, color-matrix, Bayer pattern, black-level, white-level* and *white-balance coefficients*.

The demo allows the user to select camera-based parameters, which are optional. By default, they correspond to the average values determined by Brooks et al. in [2]. Among them are the gamma value in the gamma compression inversion (set to $\gamma = 2.2$), color-balance gains (g^{red} and g^{blue}) and the global gain g^{rgb} (see Section 2). The parameter named *"camera"* selects a specific camera model from those used in the Darmstadt Noise Dataset. The four camera sensors used in this dataset are the *Sony A7R, Olympus OMD E-M10, Sony RX100* and *Nexus 6PF*. The Python code will load a specific and pre-recorded CCM. In [2], the authors sample a new CCM as a convex combination of those four. We keep this possibility and use it as the default behavior of the demo (parameter $camera$ = '*convex*').

6 Conclusion

In this work, we provide an implementation of the unprocessing pipeline introduced in [2]. While there are very little to even no datasets of real raw data in certain specific conditions (videos denoising with clean-noisy pairs for instance), there are billions of images available, for example, on the web. This method allows to create datasets easily. It can be used for image or even video processing. The pipeline is decomposed in many steps which we detailed in this paper, and we give the possibility of tuning those steps independently in the demo. First, for common users, the parameters set by default are those estimated on a real image dataset and thus lead to *realistic* raw data without tuning them. Second, the demo offers more flexibility and freedom than the initial implementation of [2] for the users who need to tune the parameters for very specific cases. In particular, a given camera model among several can be fixed, which helps reproducibility; but also the user can change the color parameters or control the range of the output data. This is particularly useful to create datasets for data-driven based methods (CNNs), since we can therefore tailor them so that they match real data on which the CNNs are evaluated afterwards. For the purpose of denoising, we also add the possibility of degrading the data with a realistic noise. The level of noise is adjusted by the user.

Image credits

All images are from the denoising template on IPOL[3], Miguel Colom, CC-BY.

[3] http://mcolom.perso.math.cnrs.fr/pages/no_noise_images/

References

[1] S.B. Ashbrook, *Photoshop CS2 RAW*, PSA Journal, 72 (2006), pp. 37–38.

[2] T. Brooks, B. Mildenhall, T. Xue, J. Chen, D. Sharlet, and J.T. Barron, *Unprocessing images for learned raw denoising*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 11036–11045. https://doi.org/10.1109/CVPR.2019.01129.

[3] V. Dewil, A. Barral, G. Facciolo, and P. Arias, *Self-supervision versus synthetic datasets: which is the lesser evil in the context of video denoising?*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2022, pp. 4900–4910.

[4] A. Foi, M. Trimeche, V. Katkovnik, and K. Egiazarian, *Practical Poissonian-Gaussian noise modeling and fitting for single-image raw-data*, IEEE Transactions on Image Processing, 17 (2008), pp. 1737–1754. https://doi.org/10.1109/TIP.2008.2001399.

[5] M. Gharbi, G. Chaurasia, S. Paris, and F. Durand, *Deep joint demosaicking and denoising*, ACM Transactions on Graphics (ToG), 35 (2016), pp. 1–12. https://doi.org/10.1145/2980179.2982399.

[6] S.W. Hasinoff, *Photon, Poisson Noise*, Springer US, 2014, pp. 608–610. https://doi.org/10.1007/978-0-387-31439-6_482.

[7] S.W. Hasinoff, F. Durand, and W.T. Freeman, *Noise-optimal capture for high dynamic range photography*, in IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, 2010, pp. 553–560. https://doi.org/10.1109/CVPR.2010.5540167.

[8] T. Plotz and S. Roth, *Benchmarking denoising algorithms with real photographs*, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 1586–1595. https://doi.org/10.1109/CVPR.2017.294.

[9] K. Wei, Y. Fu, Y. Zheng, and J. Yang, *Physics-based noise modeling for extreme low-light photography*, IEEE Transactions on Pattern Analysis and Machine Intelligence, (2021). https://doi.org/10.1109/TPAMI.2021.3103114.

[10] H. Yue, C. Cao, L. Liao, R. Chu, and J. Yang, *Supervised raw video denoising with a benchmark dataset on dynamic scenes*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020, pp. 2301–2310. http://dx.doi.org/10.1109/CVPR42600.2020.00237.