

Diffuser Cam: Lensless Imaging Algorithms

Camille Biscarrat and Shreyas Parthasarathy
Advisors: Nick Antipa, Grace Kuo, Laura Waller

December 10, 2018

1 Introduction

This guide is meant as a tutorial for the lensless image reconstruction algorithms used in DiffuserCam. It provides a brief overview of the optics involved and how it was used to develop the most current version. See our other document (“How to build a (Pi) DiffuserCam”) for information on how to actually build and calibrate DiffuserCam.

1.1 Why Diffusers?

For most 2D imaging applications, lens-based systems have been optimized in design and fabrication to be the best option. However, *lensless* imaging systems have not been investigated nearly as much. DiffuserCam is a lensless system that replaces the lens element with a diffuser (a thin, transparent, lightly scattering material). See Figure 1 below.

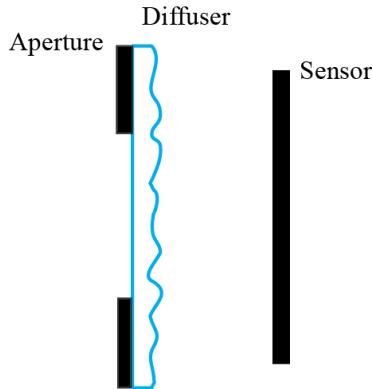


Figure 1: Cartoon schematic of DiffuserCam

Possible advantages include:

- *Lensless systems are lightweight.* Most of the weight and size of imaging systems comes from the physical constraints of lens design. Substituting lenses for a thin, flat material can allow for smaller, lighter imaging systems.
- *Diffusers require less precise fabrication.* We demonstrated that DiffuserCams (of varying quality) can be created by household scatterers such as Scotch tape. Since the structure of a diffuser is naturally random, you can create a DiffuserCam yourself without access to precise fabrication tools.

- *Possibility of 3D imaging/microscopy.* We've also shown that lensless cameras can capture 3D images and are robust to missing or dead pixels (see [this paper](#)), both of which are promising in the field of microscopy.

1.2 DiffuserCam

Every diffuser has a “focal plane”. Instead of mapping a faraway point source to a point in this plane (as lenses do), the diffuser maps a point source to a “caustic pattern” (see Fig. 2a) over the entire plane. So, replacing the lens in a camera with a diffuser of the same focal length creates a system that maps points in the scene to many points on the sensor (see Fig. 2b)

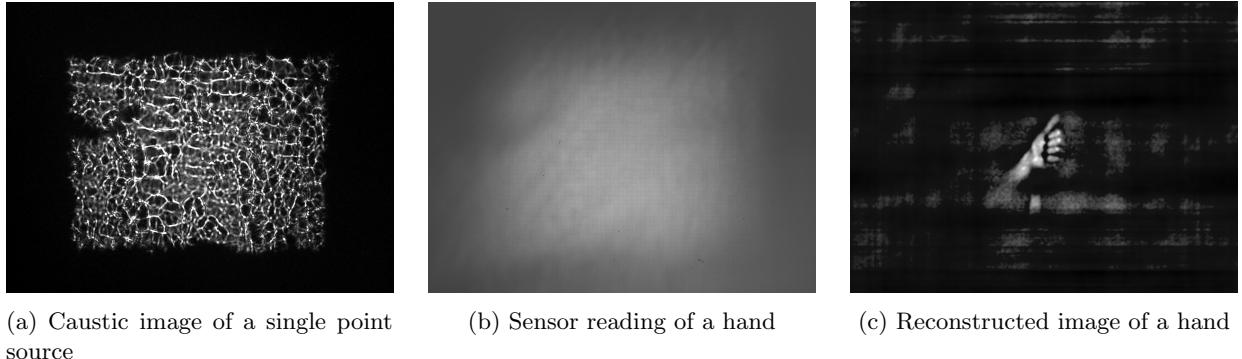


Figure 2: The 3 important steps in DiffuserCam’s operation.

The key to DiffuserCam’s operation is that, while light information is spread out over the sensor, none of that information is lost. You can see in Fig. 2b that the sensor reading won’t look like the object. However, we can recover the object image using a reconstruction algorithm that requires a single calibration measurement of the caustic produced by a point source. This measurement, called a point spread function (PSF), completely characterizes the scattering behavior of the diffuser (under certain assumptions).

1.3 Imaging Systems

To derive the algorithm and understand where these assumptions come from, it’s helpful to think of the imaging system as a function that maps objects in the real world to images on the sensor. More precisely, it is a function f that maps a 2D array v of light intensity values (the scene) to a 2D array of pixel values b on the sensor. Recovering the scene v from a sensor reading b is equivalent to inverting this function (though sometimes the function isn’t invertible):

$$f(v) = b \implies v = f^{-1}(b)$$

First, we need to describe f mathematically. In computational imaging, characterizing f (usually through a theoretical model of the optics involved) is known as constructing a “forward model,” and inverting it efficiently is known as the corresponding “inverse problem.” This tutorial covers DiffuserCam algorithms in roughly that order. Note that f is not always invertible, but that is usually because many v ’s can map to the same b . So, we often introduce *priors*, or assumptions that constrain the possible v ’s in order to construct an estimate for the scene.

2 Problem Specification

2.1 Forward Model

Roughly speaking, f is the composition of everything that happens to light as it travels from the object scene to the sensor. Each ray from a point in the scene propagates a certain distance to the diffuser and is locally refracted by the diffuser surface, then propagated again to the sensor plane. Whether or not the ray hits the sensor depends on how it was bent – we will start by ignoring this issue and addressing the finite sensor size after constructing the rest of the model.

We make the following approximations:

- *Shift invariance*: A lateral shift of the point source causes a lateral translation of the sensor reading.

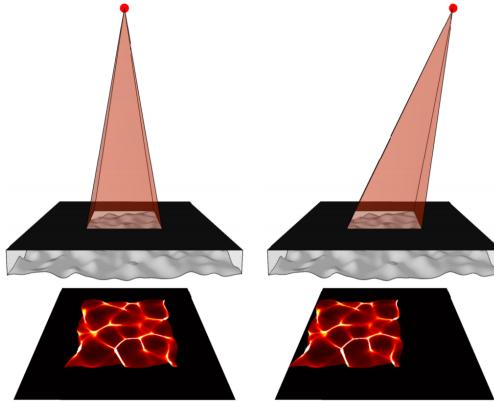
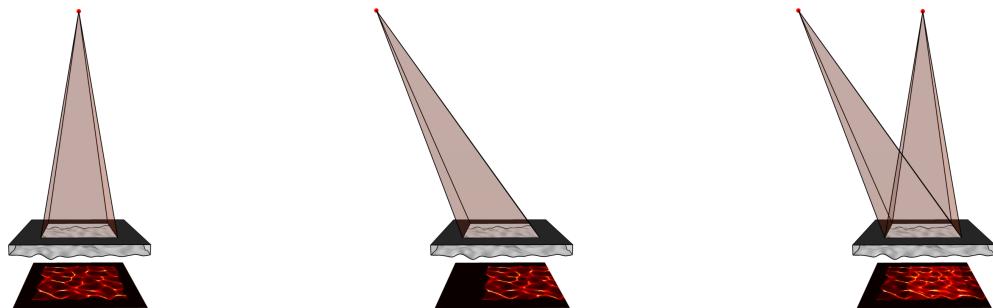


Figure 3: As the point source shifts to the right, the image on the sensor shifts to the left

- *Linearity*: Scaling the intensity of a point source corresponds to scaling the intensity of the sensor reading by the same amount. Also, the pattern due to two point sources is the sum of their individual contributions. These two assumptions amount to having *incoherent* light sources and a sensor that responds to light intensity linearly. Both of these conditions are often satisfied.



(a) Point source on axis

(b) Point source off axis

(c) Superposition of both point sources

Figure 4: Each point source creates a pattern on the sensor. When two point sources are present, the sensor reads the superposition of the patterns created by each individual point source.

In short, the diffuser system is assumed to be linear shift-invariant (LSI). We assume that v can be represented as the sum of many point sources of varying intensity and position. By the LSI property of the system, the output $f(v)$ corresponding to the input v can be represented as a 2D convolution with a single PSF h :

$$f(v) = h * v$$

Since f is linear, it is conceptually helpful to think of it as a matrix. However, matrices operate on vectors, not 2D images like v and b . We can get around this by *vectorizing* the images – creating a vector that contains the same information as the image by stacking all of the columns on top of each other. Thus our mathematical model can consistently treat these images as 1-dimensional vectors. For example, an $m \times n$ sensor reading would now be an mn -length vector. This trick allows us to represent our convolution as a 2D matrix \mathbf{M} where $h * v \iff \mathbf{M}v$. For all the following derivations, we will reserve lowercase letters for images, and bolded lowercase letters for the corresponding vectorized images. Function notation (with parentheses or braces denoting arguments) will be used to denote linear operators, and bolded uppercase letters will be used to denote the matrix representations of these operators.

Now that we've constructed a model for how the light propagates to the sensor plane, we need to account for the sensor's finite size. While all of the light rays hit the sensor plane, not all of them hit the physical sensor. So while the output of the diffuser system is a convolution, only part of that convolution is recorded on the sensor. In other words, the 2D sensor reading is a cropped convolution: $f(v) = \text{crop}(h * v)$. The equivalent vectorized formulation is

$$\begin{aligned} \text{crop}(h * v) &\iff \mathbf{C}\mathbf{M}v \\ f(v) &\iff \mathbf{A}v \end{aligned}$$

where \mathbf{C} is a matrix representation of cropping. We use \mathbf{A} as shorthand for $\mathbf{C}\mathbf{M}$. This equation serves as our forward model.

2.2 Inverse Problem

A first approach to solving for \mathbf{v} , which ignores the crop, would be to try Wiener deconvolution. This method is a common way to reverse convolution, but it relies on diagonalizing the measurement matrix, and cannot model the cropping behavior at all (see our [ADMM Jupyter notebook](#) for explanation of diagonalization). While Wiener deconvolution would work if \mathbf{A} were convolutional, i.e. $\mathbf{A} = \mathbf{M}$, adding in the crop makes \mathbf{A} too complex to invert analytically.

Instead, we must find an efficient numerical way to “invert” f . In general, f isn't invertible at all: multiple v 's can be mapped to the same b . We can see \mathbf{A} isn't invertible for two reasons:

- Information is lost in the crop operation, so \mathbf{C} is not an invertible matrix.
- Convolving with a fixed function, e.g. h , is not always invertible, so \mathbf{M} is not necessarily invertible.

The typical approach to solving $\mathbf{Av} = \mathbf{b}$ for non-invertible \mathbf{A} is to formulate it as an optimization problem, which has the same form regardless of whether \mathbf{A} is convolutional or not:

$$\mathbf{v}^* = \underset{\mathbf{v}}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{Av} - \mathbf{b}\|_2^2$$

When $\mathbf{v} = \mathbf{v}^*$, $\mathbf{Av}^* = \mathbf{b}$ and the objective function $\mathbf{Av} - \mathbf{b}$ is minimized.

It is worth noting that \mathbf{A} is extremely large, and scales with the area of the sensor. Our sensor has $\sim 10^6$ pixels, so \mathbf{A} would have on the order of $10^6 \times 10^6 = 10^{12}$ entries. While \mathbf{A} is useful mathematically, it's computationally useless to ever load/store it in memory. Whichever algorithm we choose to solve the minimization problem has to avoid ever loading \mathbf{A} in memory. Our general approach to addressing this issue will be to make sure the algorithm can be implemented in terms of the linear operators that make up f : crop and convolution. Both of these operations have fast implementations on 2D images that don't require loading their corresponding matrices.

3 Solving for \mathbf{v}

3.1 Gradient Descent

Gradient descent is an iterative algorithm that finds the minimum of a convex function by following the slope “downhill” until it reaches a minimum. To solve the minimization problem

$$\text{minimize } g(\mathbf{x}),$$

we find the gradient of g wrt \mathbf{x} , $\nabla_{\mathbf{x}}g$, and use the property that the gradient always points in the direction of steepest *ascent*. In order to minimize g , we go the other direction:

$$\begin{aligned}\mathbf{x}_0 &= \text{initial guess} \\ \mathbf{x}_{k+1} &\leftarrow \mathbf{x}_k - \alpha_k \nabla g(\mathbf{x}_k),\end{aligned}$$

where α is a step size that determines how far in the descent direction we go at each iteration.

Applied to our problem:

$$\begin{aligned}g(\mathbf{v}) &= \frac{1}{2} \|\mathbf{Av} - \mathbf{b}\|_2^2 \\ \nabla_{\mathbf{v}}g(\mathbf{v}) &= \mathbf{A}^H(\mathbf{Av} - \mathbf{b}),\end{aligned}$$

where \mathbf{A}^H is the adjoint of \mathbf{A} . Again, we want to write \mathbf{A} as a composition of linear operators that are easy to implement, so we never have to deal with \mathbf{A} itself. For a product of arbitrary linear matrices \mathbf{FG} , the adjoint is $(\mathbf{FG})^H = \mathbf{G}^H \mathbf{F}^H$. In our case:

$$\begin{aligned}\mathbf{Av} &= \mathbf{CMv} \\ \mathbf{A}^H \mathbf{v} &= \mathbf{M}^H \mathbf{C}^H \mathbf{v}\end{aligned}$$

We’ve reduced the problem of finding the adjoint of \mathbf{A} to finding the adjoints of \mathbf{M} and \mathbf{C} .

Finding the adjoint of \mathbf{M} : The adjoint of \mathbf{M} , a convolution, can be found by writing the operation using Fourier transforms. The convolution theorem states:

$$\mathbf{Mv} \iff h * v = \mathcal{F}^{-1}(\mathcal{F}(h) \cdot \mathcal{F}(v)),$$

where the \cdot denotes pointwise multiplication, and \mathcal{F} denotes the 2D Fourier transform operator. This theorem is also known as “convolution of two signals in real space is multiplication in Fourier space.” Next, we vectorize the previous statement by recognizing that 2D Fourier transforms are linear operators, so we have the equivalence $\mathcal{F}(v) \iff \mathbf{Fv}$. To fully write \mathbf{M} as a product of matrices, we must also convert the pointwise multiplication to a matrix multiplication:

$$\mathcal{F}(h) \cdot \mathcal{F}(v) \iff \text{diag}(\mathbf{Fh}) \mathbf{Fv}.$$

Also, $\mathbf{F}^H = \mathbf{F}^{-1}$ by “unitarity” of the Fourier transform. Finally, the adjoint of a diagonal matrix is formed by taking the complex conjugate of its entries.

In summary,

$$\begin{aligned}\mathbf{M}^H \mathbf{v} &= (\mathbf{F}^{-1} \text{ diag}(\mathbf{Fh}) \mathbf{F})^H \mathbf{v} \\ &= (\mathbf{F}^H \text{ diag}(\mathbf{Fh})^H (\mathbf{F}^{-1})^H)^H \mathbf{v} \\ &= \mathbf{F}^H \text{ diag}(\mathbf{Fh})^* \mathbf{F}(\mathbf{v}),\end{aligned}$$

where $*$ denotes complex conjugation.

Finding the adjoint of \mathbf{C} : Finally, we note that the adjoint of cropping, \mathbf{C}^H , is zero-padding (see section 2.4 the [appendix](#))

Plugging in to the formula for \mathbf{A}^H , we find

$$\begin{cases} \mathbf{A} = \mathbf{C}\mathbf{F}^{-1} \operatorname{diag}(\mathbf{F}\mathbf{h}) \mathbf{F} \\ \mathbf{A}^H = \mathbf{F}^{-1} \operatorname{diag}(\mathbf{F}\mathbf{h})^* \mathbf{F}\mathbf{C}^H \end{cases} \iff \begin{cases} f(v) = \operatorname{crop} [\mathcal{F}^{-1} \{\mathcal{F}(h) \cdot \mathcal{F}(v)\}] \\ f^H(x) = \mathcal{F}^{-1} \{\mathcal{F}(h)^* \cdot \mathcal{F}(\operatorname{pad}[x])\}, \end{cases}$$

where we have written \mathbf{A} in its matrix formulation (left) and the corresponding way it is implemented in code (right). Note that we converted efficient operations like pointwise multiplication to matrices purely for the derivation. See the [GD Jupyter notebook](#) for the actual implementation of these operators.

3.1.1 GD Implementation

The iterative reconstruction of \mathbf{v} looks like:

$$\begin{aligned} \mathbf{v}_0 &= \text{anything} \\ \mathbf{v}_{k+1} &\leftarrow \mathbf{v}_k - \alpha_k \mathbf{A}^H (\mathbf{A}\mathbf{v}_k - \mathbf{b}) \\ &\quad \text{Repeat forever} \end{aligned}$$

$\mathcal{F}(h)$ can be precomputed (because h is measured beforehand), and the action of $\operatorname{diag}(\mathbf{F}\mathbf{h})^H$ can be implemented as pointwise multiplication with the conjugate $\mathcal{F}(h)^*$. Since all the other operations involve only Fourier transforms, every operation in the gradient calculation can be efficiently calculated. For implementation details, see the [GD Jupyter notebook](#).

In our problem, we need to keep in mind the physical interpretation of \mathbf{v} . Since it represents an image, it must be non-negative. We can add this constraint into the algorithm by “projecting” \mathbf{v} onto the space of non-negative images. In short, we zero all negative pixel values in the current image estimate at every iteration.

One thing to keep in mind is the step size, α_k . We want it to be large at first – “coarse” jumps to get closer to the minimum quickly. As we get closer, large steps will cause the estimate to “bounce around” the minimum, overshooting it each time. Ideally we would want to decrease the step size with each iteration at a rate that would ensure continual progress. While varying step size might yield a faster convergence, it requires hand tuning and can be time consuming. A constant but sufficiently small step size is guaranteed to converge, with no parameter tuning necessary. In our case, it is possible to calculate the largest constant step size that guarantees convergence in terms of \mathbf{A} : $0 < \alpha < \frac{2}{\|\mathbf{A}^H \mathbf{A}\|_2}$, where $\|\mathbf{A}^H \mathbf{A}\|_2$ is the maximum singular value of $\mathbf{A}^H \mathbf{A}$ (see [this page](#) for why). The [GD Jupyter notebook](#) shows how we actually approximate this singular value (using \mathbf{M} instead).

Lastly, all convergence guarantees are for an infinite number of iterations: “repeat forever”. In practice, after a certain number of iterations (which varies by application) the updates are too small to change the estimate significantly. In our case, after incorporating the speedup techniques below, most of the progress is seen in the first 150-200 iterations. Sharper, more detailed images may require a few hundred more.

We also need to supply an initial “guess” of our image. It doesn’t actually matter what we use for this. Currently, we are using a uniform image of half intensity, but you could initialize with all 0’s or a random image.

Incorporating all of these details, we have:

$$\begin{aligned} \mathbf{v}_0 &= I/2 \\ \text{for } k = 0 \text{ to num_iters:} \\ \mathbf{v}'_{k+1} &\leftarrow \mathbf{v}_k - \frac{1.8}{\|\mathbf{A}^H \mathbf{A}\|} \mathbf{A}^H (\mathbf{A}\mathbf{v}_k - \mathbf{b}) \\ \mathbf{v}_{k+1} &\leftarrow \operatorname{proj}(\mathbf{v}'_{k+1})_{\mathbf{v} \geq 0} \end{aligned}$$

3.1.2 Gradient Descent Speedup

Gradient descent as written above works, but in practice, people always add a “momentum term” that incorporates the old descent direction into the calculation of the new descent direction. This guards against changing the descent direction too much and too often, which can be counterproductive. We implement momentum by introducing μ , a factor that determines how much the new descent direction is determined by the old descent direction. Typically $\mu = 0.9$ is a good place to start. Another common practice is to use “Nesterov” momentum, which involves an intermediate update \mathbf{p} . We call this method, along with the projection step, “accelerated projected gradient descent”.

```

 $\mathbf{v}_0 = I/2, \quad \mu = 0.9, \quad \mathbf{p}_0 = 0$ 
for  $k = 0$  to num_iters:
     $\mathbf{p}_{k+1} \leftarrow \mu \mathbf{p}_k - \alpha_k \text{grad}(\mathbf{v}_k)$ 
     $\mathbf{v}'_{k+1} \leftarrow \mathbf{v}_k - \mu \mathbf{p}_k + (1 + \mu) \mathbf{p}_{k+1}$ 
     $\mathbf{v}_{k+1} \leftarrow \underset{\mathbf{v} \geq 0}{\text{proj}}(\mathbf{v}'_{k+1})$ 

```

See [this page](#) for more details on parameter updates using momentum terms.

3.1.3 FISTA

Another way to speed up gradient descent is the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA). This also computes the accelerated projected gradient descent, but is more flexible about what the projection step (or more generally the “proximal” step p_L) does. For example, one can show that doing accelerated descent with ℓ_1 -regularization only requires exchanging the projection step with a soft-thresholding step. Enforcing sparsity in other domains (for instance, on the gradient of the image rather than the image itself) can be achieved via soft-thresholding transformations of the image. This algorithm is very useful for solving linear inverse problems in image processing.

Each iteration is as follows (see [this paper](#) for a derivation and explanation of each term):

```

 $\mathbf{v}_0 = I/2, \quad t_1 = 1, \quad x_0 = \mathbf{v}_0$ 
for  $k = 0$  to num_iters:
     $x_k \leftarrow p_L(\mathbf{v}_k)$ 
     $t_{k+1} \leftarrow \frac{1 + \sqrt{1 + 4t_k^2}}{2}$ 
     $\mathbf{v}_{k+1} \leftarrow x_k + \frac{t_k - 1}{t_{k+1}}(x_k - x_{k-1})$ 

```

3.2 ADMM

Although gradient descent is a reliable algorithm that is guaranteed to converge, it is still slow. If we want to process larger sets of data (e.g. 3D imaging), have a live feed of DiffuserCam, or just want to process images more quickly, we need to tailor the algorithm more closely to the optical system involved. While this introduces more tuning parameters (“knobs” to turn), speed of reconstruction can be drastically improved. Here we present (without proof) the result of using *alternating direction method of multipliers* (ADMM) to reconstruct the image.

We will only briefly motivate the use of ADMM and then provide the derivation of the update steps specific to our problem. For background on ADMM, please refer to sections 2 and 3 of: [Prof. Boyd’s](#)

[ADMM tutorial](#). To understand this document, background knowledge from Chapters 5 (Duality) and 9 (Unconstrained minimization) from [his textbook on optimization](#) may be necessary.

Recall the original minimization problem:

$$\hat{\mathbf{v}} = \underset{\mathbf{v} \geq 0}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{b} - \mathbf{A}\mathbf{v}\|_2^2, \quad (1)$$

where 2D images are interpreted as vectors. We seek to *split* the single minimization over the vector \mathbf{v} into separable minimizations – for example:

$$\begin{aligned} \hat{\mathbf{v}} &= \underset{w \geq 0, x}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{b} - \mathbf{C}x\|_2^2 \\ \text{s.t. } x &= \mathbf{M}\mathbf{v}, w = \mathbf{v}, \end{aligned} \quad (2)$$

where we have decomposed the action of DiffuserCam $\mathbf{C} = \mathbf{CM}$ into the convolution \mathbf{M} followed by a crop \mathbf{C} . The primary reason is to make the expression more amenable to the ADMM algorithm, which adds a set of “update steps” for each additional constraint. If we don’t find a nice decomposition, some of these updates will be inefficient to calculate.

In addition, because of these parallel update steps, we can add constraints (prior information) easily. A common useful prior we add is to encourage the gradient of the image to be sparse – most natural images can be approximated by piecewise constant intensities. Typically, gradient sparsity is enforced through “[total variation](#)” regularization, where we include the ℓ_1 -norm of the gradient in our objective function:

$$\begin{aligned} \hat{\mathbf{v}} &= \underset{w \geq 0, u, x}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{b} - \mathbf{Cx}\|_2^2 + \tau \|u\|_1 \\ \text{s.t. } x &= \mathbf{M}\mathbf{v}, u = \Psi\mathbf{v}, w = \mathbf{v}, \end{aligned} \quad (3)$$

where Ψ is a derivative (difference) operator.

The next step is to form the *augmented Lagrangian* (see section 2.3 in the [ADMM reference](#)), which can be directly read off from the constraints and objective function:

$$\begin{aligned} \mathcal{L}(\{u, x, w, \mathbf{v}\}, \{\xi, \eta, \rho\}) &= \frac{1}{2} \|\mathbf{b} - \mathbf{Cx}\|_2^2 + \tau \|u\|_1 \\ &\quad + \frac{\mu_1}{2} \|\mathbf{M}\mathbf{v} - x\|_2^2 + \xi^\top (\mathbf{M}\mathbf{v} - x) \\ &\quad + \frac{\mu_2}{2} \|\Psi\mathbf{v} - u\|_2^2 + \eta^\top (\Psi\mathbf{v} - u) \\ &\quad + \frac{\mu_3}{2} \|w - u\|_2^2 + \rho^\top (\mathbf{v} - w) \\ &\quad + \mathbb{1}_+(w), \end{aligned} \quad (4)$$

where the $\mathbb{1}_+(w)$ term arises from the implicit constraint $w \geq 0$:

$$\mathbb{1}_+(w) = \begin{cases} \infty & w < 0 \\ 0 & w \geq 0 \end{cases}$$

The Lagrangian dual approach to minimizing the objective function is to solve the following optimization problem:

$$\underset{\xi, \eta, \rho}{\operatorname{maximize}} \quad \underset{u, x, w, \mathbf{v}}{\operatorname{min}} \quad \mathcal{L}(\{u, x, w, \mathbf{v}\}, \{\xi, \eta, \rho\}) \quad (5)$$

The min above indicates that, ideally, we would want to jointly minimize over all the *primal* variables (u, x, w, \mathbf{v}) first, before performing the outer maximization over the *dual* variables (ξ, η, ρ) . The ADMM algorithm is a specific way of iteratively finding this optimal point. In reality, we only have estimates of

each of the variables, so the algorithm updates our estimates for the minimum primal variables during every iteration that solves for the maximum dual variables.

Based on this paradigm, we can write down all the intermediate updates that take place in one “global” update step:

$$\begin{aligned} \text{Primal Updates: } & \begin{cases} u_{k+1} \leftarrow \operatorname{argmin}_u \mathcal{L}(\{u, x_k, w_k, \mathbf{v}_k\}, \{\xi_k, \eta_k, \rho_k\}) \\ x_{k+1} \leftarrow \operatorname{argmin}_x \mathcal{L}(\{u_{k+1}, x, w_k, \mathbf{v}_k\}, \{\xi_k, \eta_k, \rho_k\}) \\ w_{k+1} \leftarrow \operatorname{argmin}_w \mathcal{L}(\{u_{k+1}, x_{k+1}, w, \mathbf{v}_k\}, \{\xi_k, \eta_k, \rho_k\}) \\ \mathbf{v}_{k+1} \leftarrow \operatorname{argmin}_{\mathbf{v}} \mathcal{L}(\{u_{k+1}, x_{k+1}, w_{k+1}, \mathbf{v}\}, \{\xi_k, \eta_k, \rho_k\}) \end{cases} \\ \text{Dual Updates: } & \begin{cases} \xi_{k+1} \leftarrow \xi_k + \mu_1(\mathbf{M}\mathbf{v}_k - x_{k+1}) \\ \eta_{k+1} \leftarrow \eta_k + \mu_2(\Psi\mathbf{v}_{k+1} - u_{k+1}) \\ \rho_{k+1} \leftarrow \rho_k + \mu_2(\mathbf{v}_{k+1} - w_{k+1}) \end{cases} \end{aligned}$$

Notice that each dual update step tries to solve the maximization problem via gradient *ascent*. In each global iteration, we make one step in the ascent direction.

Next, for each primal variable, the individual optimization problem only depends on the terms in the Lagrangian corresponding to that variable. For example, in the u -update, we only need to include the terms $\tau\|u\|_1, \frac{\mu_2}{2}\|u - \Psi\mathbf{v}\|_2^2$, and $\eta^\top(u - \Psi\mathbf{v})$; all the other terms are constant with respect to u . So, we have:

$$\begin{aligned} & \begin{cases} u_{k+1} \leftarrow \operatorname{argmin}_u \tau\|u\|_1 + \frac{\mu_2}{2}\|\Psi\mathbf{v}_k - u\|_2^2 + \eta_k^\top(\Psi\mathbf{v}_k - u) \\ x_{k+1} \leftarrow \operatorname{argmin}_x \frac{1}{2}\|\mathbf{b} - \mathbf{C}x\|_2^2 + \frac{\mu_1}{2}\|\mathbf{M}\mathbf{v}_k - x\|_2^2 + \xi_k^\top(\mathbf{M}\mathbf{v}_k - x) \\ w_{k+1} \leftarrow \operatorname{argmin}_w \frac{\mu_3}{2}\|\mathbf{v}_k - w\|_2^2 + \rho_k^\top(\mathbf{v}_k - w) + \mathbb{1}_+(w) \\ \mathbf{v}_{k+1} \leftarrow \operatorname{argmin}_{\mathbf{v}} \frac{\mu_1}{2}\|\mathbf{M}\mathbf{v} - x_{k+1}\|_2^2 + \frac{\mu_2}{2}\|\Psi\mathbf{v} - u_{k+1}\|_2^2 + \frac{\mu_3}{2}\|\mathbf{v}_{k+1} - w_{k+1}\|_2^2 \end{cases} \\ & \begin{cases} \xi_{k+1} \leftarrow \xi_k + \mu_1(\mathbf{M}\mathbf{v}_k - x_{k+1}) \\ \eta_{k+1} \leftarrow \eta_k + \mu_2(\Psi\mathbf{v}_{k+1} - u_{k+1}) \\ \rho_{k+1} \leftarrow \rho_k + \mu_2(\mathbf{v}_{k+1} - w_{k+1}) \end{cases} \end{aligned}$$

The primal minimization updates can be solved using standard convex optimization techniques, which are worked out in the DiffuserCam Derivations Supplement. The results are:

$$\begin{aligned} u_{k+1} & \leftarrow \mathcal{T}_{\frac{\tau}{\mu_2}}(\Psi\mathbf{v}_k + \eta_k/\mu_2) \\ x_{k+1} & \leftarrow (\mathbf{C}^\top \mathbf{C} + \mu_1 I)^{-1}(\xi_k + \mu_1 \mathbf{M}\mathbf{v}_k + \mathbf{C}^\top \mathbf{b}) \\ w_{k+1} & \leftarrow \max(\rho_k/\mu_3 + \mathbf{v}_k, 0) \\ \mathbf{v}_{k+1} & \leftarrow (\mu_1 \mathbf{M}^\top \mathbf{M} + \mu_2 \Psi^\top \Psi + \mu_3 I)^{-1} r_k, \\ \xi_{k+1} & \leftarrow \xi_k + \mu_1(\mathbf{M}\mathbf{v}_{k+1} - x_{k+1}) \\ \eta_{k+1} & \leftarrow \eta_k + \mu_2(\Psi\mathbf{v}_{k+1} - u_{k+1}) \\ \rho_{k+1} & \leftarrow \rho_k + \mu_2(\mathbf{v}_{k+1} - w_{k+1}) \end{aligned}$$

where

$$r_k = (\mu_3 w_{k+1} - \rho_k) + \Psi^\top(\mu_2 u_{k+1} - \eta_k) + \mathbf{M}^\top(\mu_1 x_{k+1} - \xi_k)$$