Cornell CS 322 Introduction to Scientific Computing *out: Wed Mar 12, 2008*
Project #2: ShadowBox (and the SVD-powered X-Ray Glasses) *due: Sun Apr 13, 2008*
Prof. Doug James *(before midnight)*

1 Introduction

In this assignment, you will learn how to "see through paper" to infer mystery light sources inside a ShadowBox. "How can this incredible power become mine," you ask? The answer is that you will learn how to use the method of Least Squares and the too-good-to-be-true Singular Value Decomposition (SVD) to invert the nearly singular light transport process. Mathematically, you will solve $\mathbf{Ax} = \mathbf{b}$ problems with rank-deficient $\mathbf{A}$ matrices, and where the $\mathbf{x}$ and $\mathbf{b}$ vectors represent interesting images. The starter code provides a lighting simulation of the *ShadowBox* illustrated in Figure 1. The efficient lighting simulation effectively implements a matrix-vector multiply, $\mathbf{Ax}$, for an $\mathbf{A} \in \mathbb{R}^{10000\times 10000}$, so that you can render pretty pictures, $\mathbf{b} = \mathbf{Ax} \in \mathbb{R}^{10000}$, for any light source image, $\mathbf{x} \in \mathbb{R}^{10000}$. *The main task you must solve is as follows: given a rendered image,* $\mathbf{b}$, *what light source* $\mathbf{x}$ *did the ShadowBox use to render it?* Unfortunately, the $\mathbf{A}$ matrix is too big and ill-conditioned to invert directly (and only indirectly observable in this assignment) so we will take a different approach: we first render paper images for a calibrated light source basis, then find the linear combination of images that best matches the $\mathbf{b}$ we see, thus inferring the light source $\mathbf{x}$ that created the image.

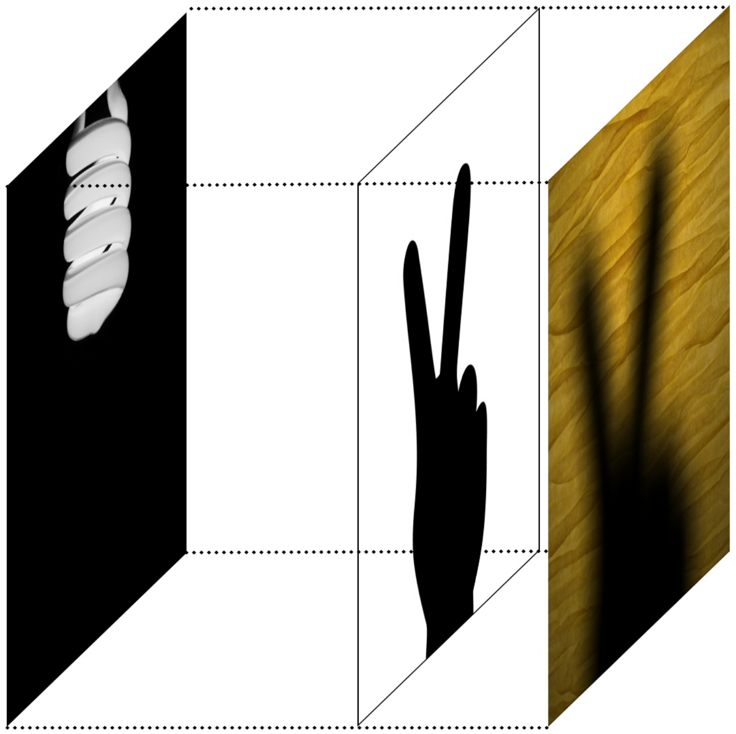


Figure 1: **The ShadowBox** *is a cube comprised of three parallel planes: (Left) a 100-by-100 light source image,* $\mathbf{x}$, *(Middle) a a 100-by-100 blocker image used in lighting simulation ("*$\mathbf{A}$*"), (Right) a 100-by-100 rendered image ("*$\mathbf{b}$*") of irradiated paper with soft shadows visible.*

1.1 ShadowBox Basics and Lighting Estimation

The ShadowBox is a cube with internal light/blocker/paper image planes as shown in Figure 1. We use square m-by-m images, each with $M = m^2$ pixels. In this assignment, we will use $m = 100$ throughout. For simplicity and efficiency, we use grayscale images. By exploiting the plane-parallel light/blocker/paper geometry, direct illumination (paper irradiance) can be computed efficiently using vectorized Matlab operations. The details are discussed in Appendix A for your information, but you are not required to understand light transport in this assignment.

Matlab Tip (Image matrix-vector conversion): Each m-by-m grayscale image can be interpreted as an M-dimensional state vector (recall $M = m^2$). To convert an m-by-m image matrix to an M-vector, you can use the Matlab `reshape()` function, e.g.,

```
vecImage = reshape(matImage, m², 1)
```

provides a column-major vector view of `matImage`, whereas

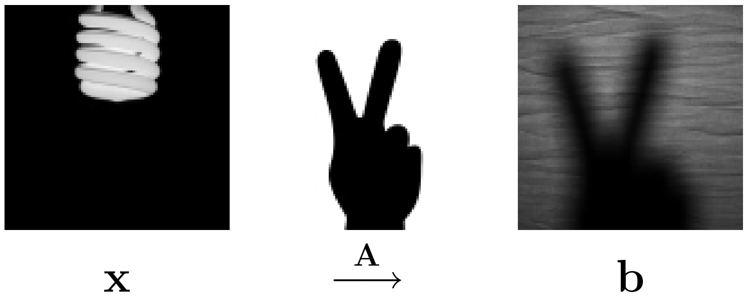
```
matImage = reshape(vecImage, m, m).
```

slices the vector `vecImage` into columns of the matrix `matImage`.

Light transport and $\mathbf{Ax} = \mathbf{b}$: By linearity of the light transport process (direct illumination), the observed paper image, $\mathbf{b} \in \mathbb{R}^M$, is linearly related to the light source image, $\mathbf{x} \in \mathbb{R}^M$, by a linear matrix operator, $\mathbf{A} \in \mathbb{R}^{M \times M}$:

$$\mathbf{Ax} = \mathbf{b}. \tag{1}$$

This linear algebraic relationship can be visualized as a light source image, $\mathbf{x}$, being converted by the linear light transport operator, $\mathbf{A}$, to a rendered image, $\mathbf{b}$:



We provide you with an efficient Matlab implementation of the rendering operator, $\mathbf{A}$.

Light and Blocker Images: All provided light and blocker images are grayscale JPG files stored in the `shadowBox/images` folder. Blocker images are black where the blocker exists, and white otherwise–grayscale values are allowed, and represent partial occlusion. Light images are stored as negative images, and when loaded the values on $[0, 1]$ are remapped by the transformation $b \leftarrow 1 - b$. The benefit of this arrangement is that all images can be used as either blockers or lights.

1.2 ShadowBox Inversion: Lighting Inference

In this assignment, we are interested in the inverse illumination process: given a blocker image and a rendered paper image, $\mathbf{b}$, what light source, $\mathbf{x}$, was used to create it? See Figure 2. Of course, you will also be able to observe $\mathbf{A}$, but we will not explicitly construct the dense matrix due to its inconvenient size: $\mathbf{A} \in \mathbb{R}^{M \times M} = \mathbb{R}^{10000 \times 10000}$ (which is about 800MB in Matlab).

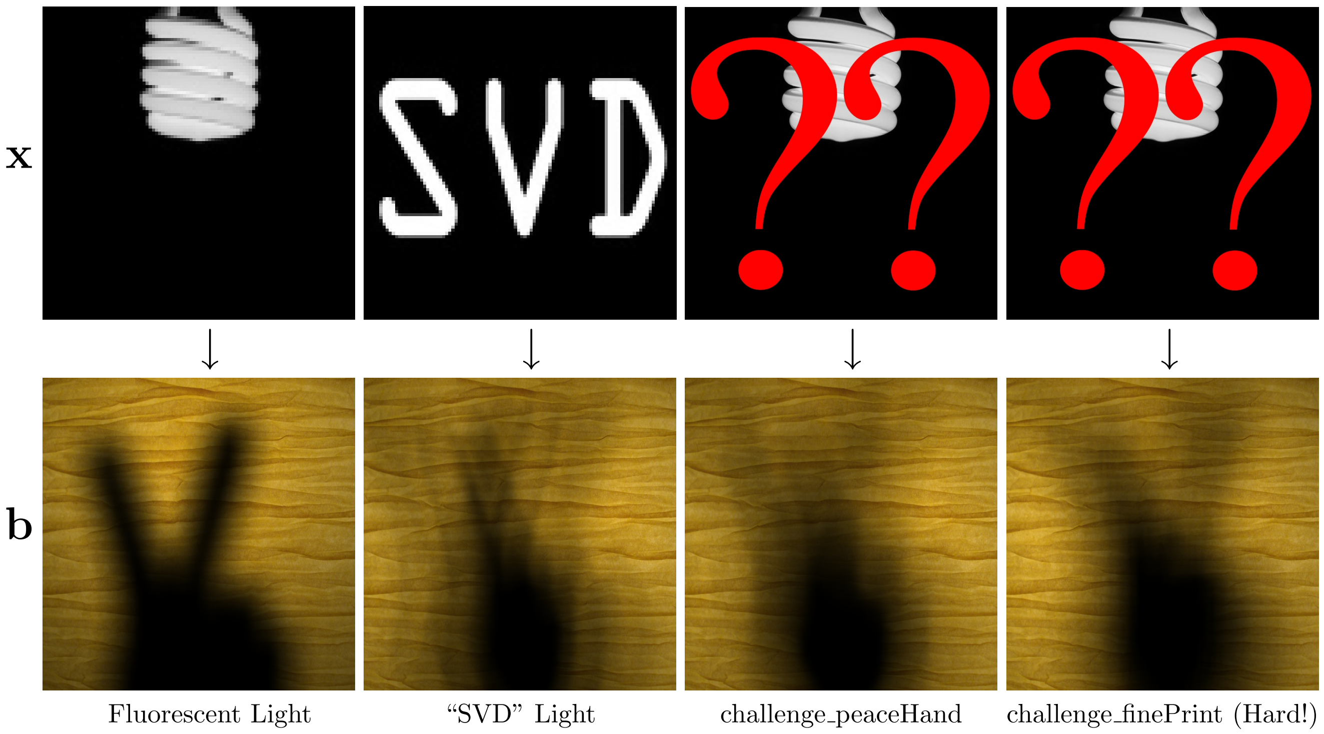


Figure 2: **Can you guess the light in these ShadowBoxes?:** *The compact fluorescent light casts a soft but clear shadow, whereas the "SVD" light is more complex. It turns out that by looking at these shadows, we can infer the light sources. Although this is hard to do in your head (especially for the cases on the right), you will be able to estimate the rough shape of the light source using least squares and SVD.*

1.2.1 LightGrid Basis

To infer the lighting that produced any image, we will observe the paper images produced by each light in a G-by-G grid of lights, or what we will call our LightGrid–see Figures 3 and 4. The LightGrid basis and optimized paper-image rendering, is implemented for you in the starter code. Mathematically, we seek to estimate light source images, $\mathbf{x}$, that can be represented in the LightGrid basis as

$$\mathbf{x} = \sum_{j=1}^{N} \mathbf{X}_{:j}\mathbf{c}_j = \mathbf{X}\mathbf{c}, \tag{2}$$

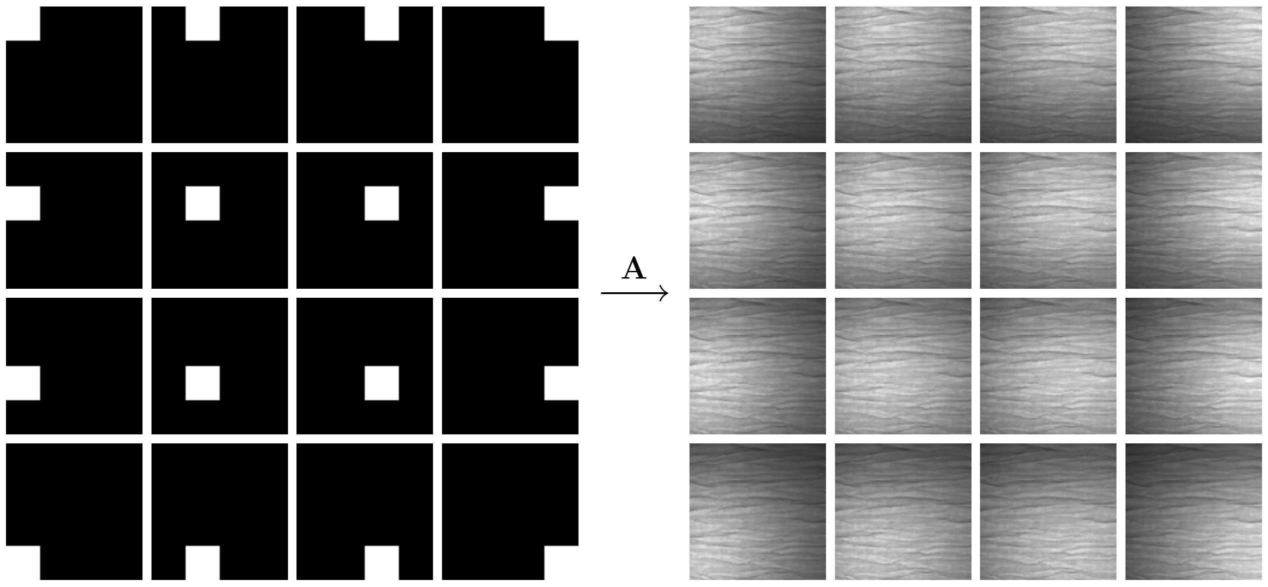
where the dimension of the LightGrid basis is $N = G^2$; the LightGrid basis functions, $\mathbf{X}_{:j}$, are 0 everywhere except at "on" pixels where they are 1; and $\mathbf{c} \in \mathbb{R}^N$ are the expansion coefficients of $\mathbf{x}$ in the LightGrid basis, $\mathbf{X}$.

Using the starter code, you will precompute a corresponding rendered-image basis, $\mathbf{B}$, by applying $\mathbf{A}$ to (i.e., rendering) each $\mathbf{X}_{:j}$.

$$\mathbf{B}_{:j} \equiv \mathbf{A}\mathbf{X}_{:j}. \tag{3}$$

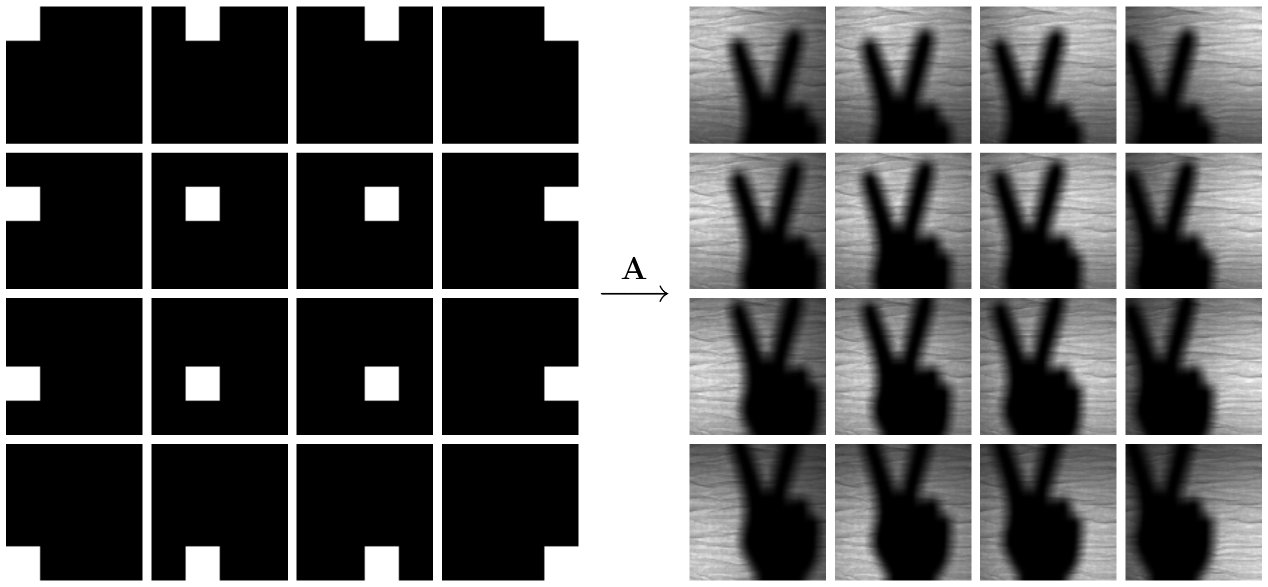
Both $\mathbf{X}$ and $\mathbf{B}$ are illustrated in Figures 3 and 4. These N images, each of size m-by-m, are currently returned as a large three-dimensional m-by-m-by-N tensor matrix–you may want to change this to return an M-by-N matrix. By default, the starter code will display, and then output both $\mathbf{X}$ and $\mathbf{B}$ JPG images to the `shadowBox/basisExport` folder for your personal observation–you may want to disable this for efficiency reasons.

Tip: Since the images are m-by-m=100-by-100, your grids will be most regular when using G values such as 2, 4, 5, 10, 20, 25, 50.



LightGrid basis images, $\mathbf{X}_{:j},\ j=1\ldots16$. Paper image basis, $\mathbf{B}_{:j}=\mathbf{AX}_{:j},\ j=1\ldots16$.

Figure 3: **LightGrid "pixel" basis (X), and resulting rendered image basis (B)** *for the "no blocker" case; here $G=4$ which results in an $N=G^2=16$ dimensional basis. By estimating what combination of paper image basis vectors,* $\mathbf{B}$*, best estimate what you see, you also find an estimate of the ligth source in the LightGrid pixel basis,* $\mathbf{X}$*.*



LightGrid basis images, $\mathbf{X}_{:j},\ j=1\ldots16$. Paper image basis, $\mathbf{B}_{:j}=\mathbf{AX}_{:j},\ j=1\ldots16$.

Figure 4: **LightGrid "pixel" basis (X), and resulting rendered image basis (B)** *for the "peace-Hand.jpg" blocker image; here $G=4$ which results in an $N=G^2=16$ dimensional basis.*

1.3 Least-Squares Lighting Estimation

Since any light source $\mathbf{x} = \mathbf{Xc}$ represented in the LightGrid (2) has a rendered image,

$$\mathbf{b} = \mathbf{Ax} = \mathbf{A} \sum_{j=1}^{N} \mathbf{X}_{:j} \mathbf{c}_j = \sum_{j=1}^{N} (\mathbf{AX}_{:j})\, \mathbf{c}_j = \sum_{j=1}^{N} \mathbf{B}_{:j} \mathbf{c}_j = \mathbf{Bc}. \tag{4}$$

Once we know $\mathbf{c}$ we can construct both $\mathbf{b}$ and $\mathbf{x}$ using the LightGrid renderings. Therefore, one strategy for estimating the light source, $\mathbf{x}$, from a rendered image, $\mathbf{b}$, is as follows:

1. Solve $\mathbf{Bc} = \mathbf{b}$ for $\mathbf{c}$. Since $\mathbf{b}$ does not, in general, lie in the span of $\mathbf{B}$, we will solve this equation in a least-squares sense for $\mathbf{c}$.

2. Reconstruct the estimated light source image, $\mathbf{x} = \mathbf{Xc}$.

Since the rendered-image basis, $\mathbf{B}$, can be effectively singular in finite precision, you will need a robust least-squares solver. You will solve $\mathbf{Bc} = \mathbf{b}$ using a truncated SVD (TSVD) solver as follows:

1. Construct the thin SVD of $\mathbf{B}$ (`svd(B,0)` in Matlab),

$$\mathbf{B} = \mathbf{USV}^T = \sum_{k=1}^{r} \mathbf{U}_{:k} \sigma_k (\mathbf{V}_{:k})^T, \tag{5}$$

   where σ_k is the k^{th} singular value, $\sigma = diag(\mathbf{S})$, and $r = \min(M, N)$.

2. Filter singular values with $\sigma_k < \varepsilon \sigma_1$, where σ_1 is the largest singular value ($\sigma_1 = \|\mathbf{B}\|_2$), and $\varepsilon \in (0, 1)$ is a relative truncation tolerance. Denote the new rank-$\tilde{r}$ approximation by

$$\tilde{\mathbf{B}} = \tilde{\mathbf{U}}\tilde{\mathbf{S}}\tilde{\mathbf{V}}^T = \sum_{k=1}^{\tilde{r}} \mathbf{U}_{:k} \sigma_k (\mathbf{V}_{:k})^T, \tag{6}$$

3. Construct the least-squares solution, $\tilde{\mathbf{c}}$, using

$$\tilde{\mathbf{c}} = \tilde{\mathbf{B}}^{\dagger}\mathbf{b} = \tilde{\mathbf{V}}\tilde{\mathbf{S}}^{-1}\tilde{\mathbf{U}}^T\mathbf{b} = \sum_{k=1}^{\tilde{r}} \mathbf{V}_{:k} \frac{1}{\sigma_k} (\mathbf{U}_{:k})^T \mathbf{b}, \tag{7}$$

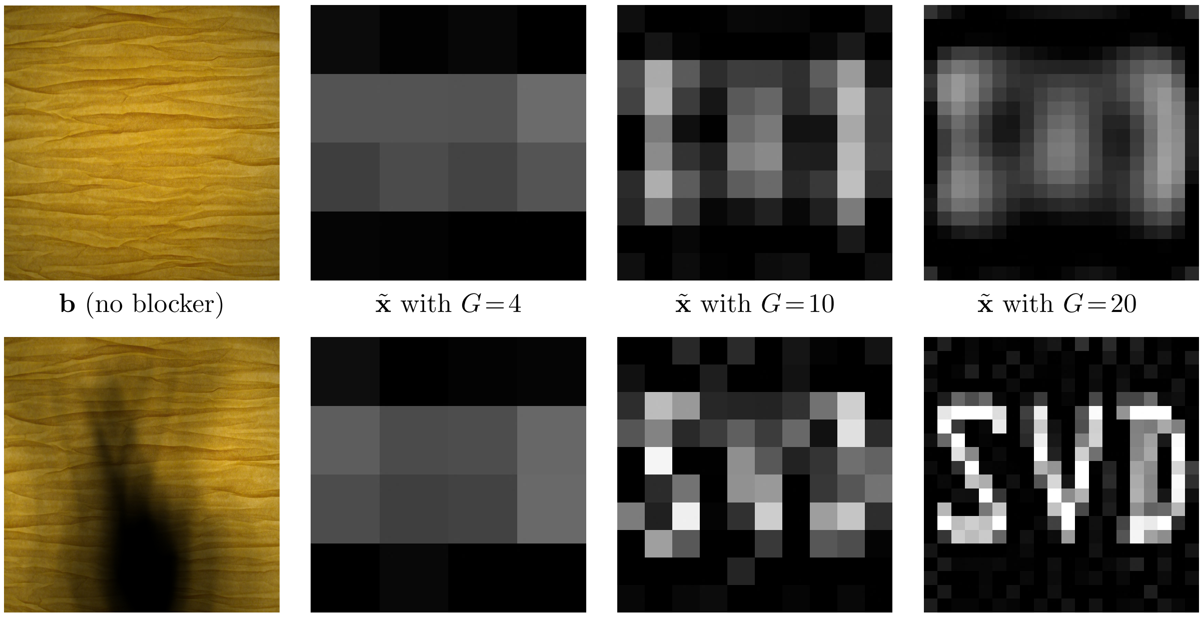
   which can be done efficiently using a sequence of matrix-vector multiplies. You should not construct the pseudoinverse $\tilde{\mathbf{B}}^{\dagger}$ to solve the linear system–analogous to solving $Ax = b$ using LU factorization and back-substitution.

Once you find $\tilde{\mathbf{c}}$, you can reconstruct the estimated light source image using $\tilde{\mathbf{x}} = \mathbf{X}\tilde{\mathbf{c}}$. Previews of this lighting estimation method are shown in Figures 5 and 6.

1.4 Filtering "Negative Light"

Physically speaking, the light image, $\mathbf{x}$, should be a nonnegative quantity since there is no such thing as negative emitted light. However, your least-squares estimate of $\mathbf{x}$ can introduce negative light values to reduce the squared residual error. We would like to avoid these negative values[1]. To do so, you can simply clamp your estimated light image values to be nonnegative, $\mathbf{x} \geq 0$. A simple Matlab command to clamp `x` to be nonnegative is to multiply it by a mask indicating nonnegative values: `xNonneg = (x >= 0) .* x;`

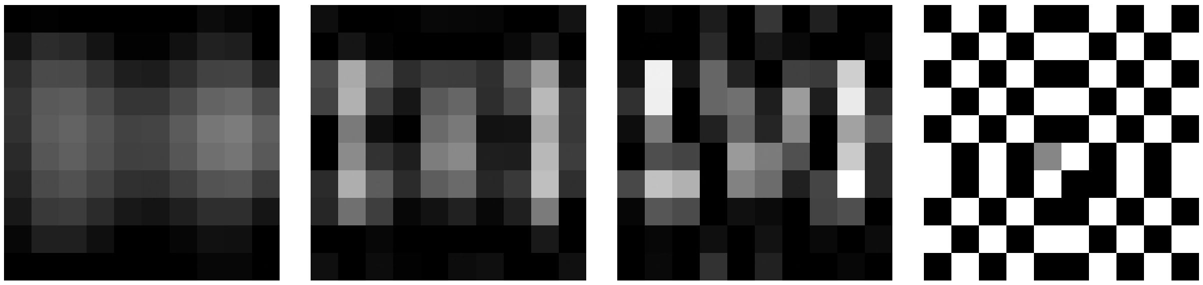
[1]It's also rumored that negative light can hurt your eyes–just kidding.



$\mathbf{b}$ (no blocker) — $\tilde{\mathbf{x}}$ with $G=4$ — $\tilde{\mathbf{x}}$ with $G=10$ — $\tilde{\mathbf{x}}$ with $G=20$

$\mathbf{b}$ (peaceHand blocker) — $\tilde{\mathbf{x}}$ with $G=4$ — $\tilde{\mathbf{x}}$ with $G=10$ — $\tilde{\mathbf{x}}$ with $G=20$

Figure 5: **Estimating "SVD" lighting using various LightGrid resolutions, G:** *(Top) No-blocker scenario ($\varepsilon=10^{-7}$), and (Bottom) "peaceHand" blocker ($\varepsilon=10^{-4}$). Lighting corresponds to the "SVD" light source image in Figure 2.*



$\tilde{\mathbf{x}}$ with $\varepsilon=10^{-4}$ — $\tilde{\mathbf{x}}$ with $\varepsilon=10^{-7}$ — $\tilde{\mathbf{x}}$ with $\varepsilon=10^{-10}$ — $\tilde{\mathbf{x}}$ with $\varepsilon=10^{-14}$

Figure 6: **Varying SVD truncation threshold, ε:** *TSVD estimates of the "SVD" light source using $G=10$ LightGrids, and* $\mathbf{c}$ *clamped to nonnegative values. TSVD solutions are shown for various ε TSVD truncation thresholds, and indicate that decreasing ε helps to resolve finer detail, but that decreasing it too far can produce nearly singular solutions, e.g., the $\varepsilon=10^{-14}$ approximation has "almost divided by zero."*

1.5 Evaluating Error

Given a light-source estimate, $\tilde{\mathbf{x}}$, you can evaluate its accuracy in two ways. First, you can always compute the *relative residual error*,

$$RelResidErr = \frac{\|\mathbf{b} - \mathbf{A}\tilde{\mathbf{x}}\|_2}{\|\mathbf{b}\|_2}. \tag{8}$$

by rendering your light estimate $\tilde{\mathbf{x}}$ with `sboxRender()` to get $\mathbf{A}\tilde{\mathbf{x}}$. Second, if you know the exact light solution, $\mathbf{x}$, you can compute the *relative lighting error*,

$$RelLightErr = \frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2}. \tag{9}$$

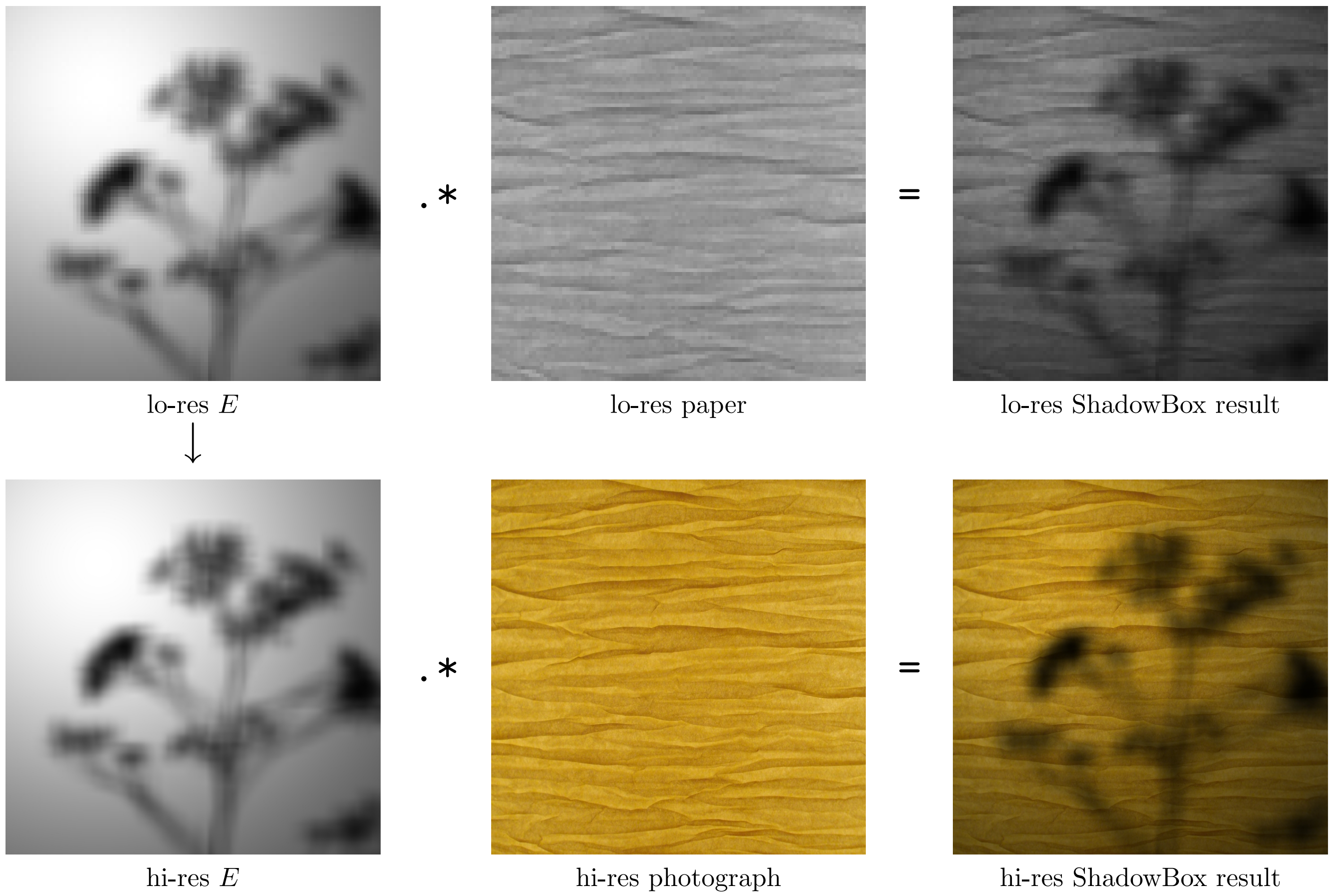


Figure 7: **Applying paper texture:** *We render low-resolution images (Top/Left) of paper irradiance, E, which are then (Top/Middle) composited with a low-resolution grayscale paper texture, to obtain the (Top/Right) rendered paper image. For high-quality display, the low-resolution irradiance is up-sampled (using bicubic interpolation) to obtain (Bottom/Left) high-resolution irradiance image, which is multiplied by the (Bottom/Middle) high-resolution color photograph of the paper texture, to obtain (Bottom/Right) the final high-resolution ShadowBox rendering. (Note: To obtain suitable 0-255 pixel ranges, the input irradiance E is scaled (tone-mapped) only for final display purposes.)*

2 Matlab Program Structure

The starter code consists of several Matlab files, and numerous JPG images (light, blocker, paper, challenge images). All starter-code functions have an `sbox` prefix for easy access (and command completion) in Matlab. The Matlab files are as follows:

- `sboxLoadBlockerImage.m` Loads an m-by-m blocker image–100-by-100 assumed throughout.
- `sboxLoadLightImage.m` Loads an m-by-m light image (transforming as specified in §1.1).
- `sboxMake.m` Constructs a ShadowBox, returning a Matlab structure with precomputed quantities needed for efficient rendering.
- `sboxRender.m` Renders a low-resolution grayscale paper image (with optional irradiance) for a specified blocker image, and light image.

- `sboxRenderLightBasis.m` Given a LightGrid resolution, G, and blocker image, renders and returns a paper image tensor corresponding to a G-by-G LightGrid basis (also returned). (Note: Implementation is not optimized for subsequent memory use (more sparsity comments below).)
- `sboxDraw*.m` Various routines to display rendered images, including high-quality renderings based on compositing paper irradiance with high-resolution color paper photographs. We now briefly mention one routine.
- `sboxDrawPaper.m` This function helps render using high-resolution colored paper. Low-resolution, grayscale, rendered-paper images are used for efficient estimation processing, however high-resolution colored paper is available for final renders–no, this is not a printer tutorial. Mathematically, we just up-sample the scalar-valued, low-resolution paper irradiance image, then multiply it by a high-resolution colored paper texture. The function to do this is `sboxDrawPaper(SB, E)`, where `E` is the irradiance image produced by `sboxRender`, e.g., `[PI, E] = sboxRender(SB, blockerImage, Phi)`. The process is illustrated in Figure 7
- `solveTSVD.m` An empty TSVD solver function placeholder for you to implement.
- `go.m` A starter script that illustrates the use of the aforementioned functions. In part, you need to extend this script to perform the estimation process.

3 Mystery-Light Challenges

As a challenge, we provide you with a collection of paper image renderings, **b**, using specified blocker images, but unknown light images, **x**. Can you infer which lights produced the images? The "Mystery Light Challenges" are shown in Table 1. In each case, the 100-by-100 BW paper image renderings (and related files) are available in the `challenges` subdirectory. The preferred analysis format is a Matlab binary file, `challenge_name.mat`, which contains a double-precision matrix loadable using a "`load challenge_name.mat`" Matlab command. (Note that image compression/quantization introduce nonlinearities can ruin the linear relationship between **x** and **b**, so use the double-precision matrix for analysis.)

4 Implementing Your Solution

In this section, we discuss what you need to implement for the assignment, and some Matlab programming tips.

4.1 Assignment Checklist

The assignment involves the following items for which you should provide code and an explanatory document with supporting images/plots (e.g., this document was created with pdflatex):

1. **Warm-up:** Familiarize yourself with the code by rendering shadows for some of the 100-by-100 light source and blocker images in the `images` subdirectory. Take a look at basis images rendered at different PixelGrid resolutions, G.
2. **Repack rendered basis images as an M-by-N matrix** ($M = m^2 = 10000$, $N = G^2$) to prepare for TSVD solution, and matrix-vector multiplication.
3. **TSVD Solver:** Implement a function for TSVD linear system solution, for a given ε value, that reuses the `[U,S,V]` result of the thin SVD. You will need this solver to estimate $\tilde{\mathbf{c}}$ as in (7). Your function should have the calling signature:

   `x = solveTSVD(U, S, V, b,` ε`);`

   where here `x` is the solution to a proverbial "$Ax = b$" problem (where $A = USV^T$), and is not to be confused with your particular notation.

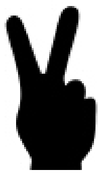
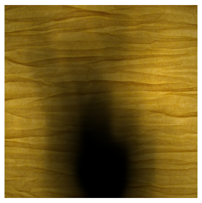
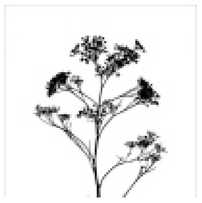
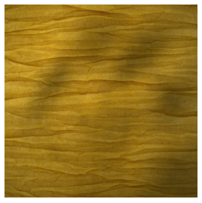
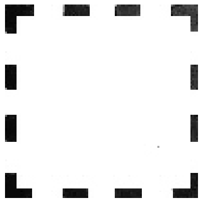
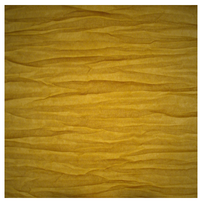
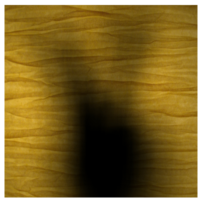
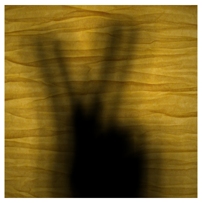
Difficulty	Blocker	Paper	Description
Fair	 `peaceHand.jpg`	 `challenge_peaceHand.jpg`	**peaceHand** Rendering of the "peace hand" for an unknown large light source. What is it?
Fair	 `plant2.jpg`	 `challenge_atTheBeach.jpg`	**atTheBeach** Rendering of a plant blocker for an unknown large and fictitious light source. What is it?
Medium	 `frame.jpg`	 `challenge_howManyFingers.jpg`	**howManyFingers** Rendering of a picture frame, with an unknown light source. "How many fingers am I holding up?"
Hard	`peaceHand.jpg`	 `challenge_finePrint.jpg`	**finePrint** Rendering of the "peace hand" for an unknown detailed light source. Can you "read the fine print?" *Warning: this one is hard.*
Impossible?	`peaceHand.jpg`	 `challenge_bonusQuestion.jpg`	**bonusQuestion** Rendering of the "peace hand" for an unknown detailed light source specifying a question from Cheney & Kincaid. What is the mystery question? *Warning: very challenging if not impossible.*

Table 1: **"Mystery-Light Challenges!"** *The goal in each case is to estimate the grayscale light source image,* $\mathbf{x}$*, that produced the observed paper image, for the specified blocker image. We provide you with the grayscale paper image vector,* $\mathbf{b}$*. Can you recognize/read the light source image? Some of the harder images, require estimating high-resolution light patterns from only the paper's shadow image, which can be tricky. Good luck!*

4. **Compare the TSVD solver to the normal equations approach** for the case of no-blocker image (`images/none.jpg`), and the "SVD" light source image (`images/SVD.jpg`). How do the normal equations compare to the TSVD solver, when you use a larger G value, such as $G = 20$? Plot the singular values of the $\mathbf{B}$ matrix used in the TSVD solver, as well as the singular values of the normal equations matrix, $\mathbf{B}^T\mathbf{B}$. Interpret your results.

5. **Plot RelLightErr (9) vs** ε for the previous ("SVD" lighting; no blocker) checklist item. Use an interesting range of ε and at least ten data points. Interpret your results.

6. **Mystery-Light Challenges:** Consider as many of these light-source estimation challenges as you can. You should be able to identify the images for some of them, but probably not all of them unless you make a special effort. For each challenge, you should document the code used, your resulting image, and supporting comments (what is it?) and facts, e.g., the relative residual error.

4.2 Matlab Programming Tips

The great part of Matlab is that its trivial to visualize and plot your data. You should do this often so that you can build intuition and understand what your equations and code are doing. Some tools that you will find useful are:

- `imagesc(A)` This routine allows you to look at a matrix `A` as an image. The colormap is automatically scaled (hence the "sc" suffix) to fit the matrix's min/max range. You can also specify the range explicitly, e.g., `imagesc(A, [0 1]);`

- `surf(A)` This routine allows you to look at a matrix `A` as a surface plot. This can be useful when you want to see the min/max magnitudes, e.g., to see if your light source estimate has introduced very much negative light.

- `The backslash ‘‘\’’ operator`
; This "left matrix divide" operator intelligently selects a linear system solver to allow the proverbial $Ax = b$ to be solved as easily as saying `A`
`b;`, which is algebraically equivalent to $A^{-1}b$, but algorithmically faster–type "help mldivide" for details. You could use this to help you solve the normal equations–when they work. Unfortunately, it does not handle rank deficient or ill-conditioned matrices such as you will encounter.

- `Use semicolon indices` to help you access matrix columns and rows. For example, to access column j of matrix A, just say `A(:,j)`; row i is just `A(i,:)`; if you want rows 1 through 10 of column j, use `A(1:10,j)`.

4.3 Matlab Memory Management Tips

To best resolve detailed light sources, a detailed light basis is needed. While basis rendering and SVD factorization are time consuming (approximately $O(M^2)$), the problem you will likely encounter is an "OUT OF MEMORY ERROR." Fortunately, you can optimize the starter code to be more memory efficient in many ways:

- **Avoid redundant copies:** The light source basis images are packed as a dense m-by-m-by-N tensor, but you will need an M-by-N matrix for SVD. Try to avoid keeping reformatted copies of these large data matrices/tensors, by constructing the matrix you really need in the first place–when they are rendered.

- **Use the thin SVD:** By default, `svd(A)` will construct a "fat" SVD, complete a memory-intensive basis for the null space–yikes! Remember to use the thin SVD, `svd(A,0)`, to save time and memory.

- **Exploit sparsity:** The light source basis images take up as much memory as the rendered paper images: each has MN doubles, or $80000N$ bytes ($m = 100$), so for $G = 50$ (half resolution) you would have 200 MB. Recall that once you find an $\tilde{\mathbf{x}}$ solution, you use the light basis matrix $\mathbf{X}$ to reconstruct your light source via a matrix-vector multiply. Fortunately, these light basis images are sparse: they are zero everywhere except at light source "pixels." You can use the `sparse()` function to help you assemble a sparse M-by-N light basis matrix, instead of the dense one–let alone the dense m-by-m-by-N tensor. Doing so will help you reduce its memory cost by about a factor of $N = G^2$!
- **Avoid allocating too many things at once:** You may run out of memory if you have too many matrices lying around. Use "`clear X`" to clear an unused variable `X` from memory before proceeding to the next calculation.
- **Save/Load basis images:** You may find it useful to use Matlab's `save` and `load` functionality to save/load LightGrid basis images (or other quantities) for a given blocker image. This can help you avoid having to re-render images, or other quantities.
- **Only use what you need:** If you are really clever (hint), you may realize that you may not need all the light basis images. Discarding unneeded basis images will help you use less memory, and make your SVD solver run faster.

4.4 What to Submit?

You should submit the following in a `zip` file via CMS:

1. **Your Matlab implementations, complete with directory structure** necessary to run your programs that implement the checklist items (§4.1). If you need to add any additional functions, you may do so in additional files (please give them meaningful names and comments).
2. **A brief report** describing your findings for the checklist items (§4.1), including any images/plots needed to support your findings. PDF documents are preferred, but can be made using any editor/program you choose, e.g., LaTeX/pdflatex.
3. **Do not submit huge files:** Do not submit `basisExport` images, since they are not needed for grading, and may be numerous. Provide code to generate any quantities that are needed in your estimation process from scratch.
4. **Code Documentation:** As in the first assignment, your program should be documented *thoroughly*. Numerical code without good documentation can be very hard to read, so it is important that you explain to us what you are doing via documentation in your program. If there is any point in your solution at which is is not completely clear *what* your are doing or *why* you are doing it, explain it using comments. If your approach has limitations or potential shortcomings, explain it using comments. If you do something clever in your program to speed it up, or make it more accurate, or make it more robust, explain it using comments. Part of your mark is based on how easy your code is to understand.

A Background: Direct Illumination

Our working model of light propagation in the ShadowBox is based on *direct illumination*: light falling on the back of the paper (irradiance) arrives *directly* from a light source position without any intermediate reflections [1]. This assumption is consistent with the box interior consisting of nonreflective black surfaces. Although we provide you with a working shadow box to start with, we now briefly discuss its inner workings so you better understand the assignment. However, we assure you that *you will not be tested on anything to do with illumination calculations!*

Irradiance of a point light: Given a small (point) light source at position $\mathbf{s} \in \mathbb{R}^3$ with pixel power $\Phi(\mathbf{s}) \in \mathbb{R}$ that emits light uniformly in all directions, the *irradiance*, $E \in \mathbb{R}$, falling on the back of the paper at position $\mathbf{p} \in \mathbb{R}^3$ is

$$E = \frac{\Phi(\mathbf{s})\cos\theta}{4\pi r^2} = \frac{\Phi(\mathbf{s}) Z_{si}}{4\pi r^3} \tag{10}$$

where $r = \|\mathbf{p} - \mathbf{s}\|_2$ is the distance between $\mathbf{p}$ and $\mathbf{s}$, $\cos\theta$ describes the cosine factor for light hitting the paper at angle θ, Z_{si} is the distance between the parallel light and paper planes.

Visibility: Due to the blocker between the light and the paper, not all light source positions $\mathbf{s}$ will contribute light at $\mathbf{p}$. We can encode this using a *visibility delta function*

$$V(\mathbf{s}, \mathbf{p}) = V(\mathbf{p}, \mathbf{s}) = \begin{cases} 1, & \text{if } \mathbf{s} \text{ is visible from } \mathbf{p}, \\ 0, & \text{otherwise.} \end{cases} \tag{11}$$

In practice, given $(\mathbf{s}, \mathbf{p})$ we can evaluate $V(\mathbf{s}, \mathbf{p})$ by first computing the point $\mathbf{b}$ on the blocker plane,

$$\mathbf{b} = (1 - \beta)\mathbf{s} + \beta\mathbf{p}, \tag{12}$$

where β is a constant implied by the blocker plane location. Second, a blocker texture lookup is done to determine if the ray passing through $\mathbf{b}$ is occluded. We provide an efficient vectorized implementation of these visibility tests for you. In terms of $\mathbf{b}$, we can denote the visibility function by

$$V(\mathbf{s}, \mathbf{p}) = \mathit{BlockerImage}(\mathbf{b}) \tag{13}$$

where this is achieved using an optimized texture look-up operation on the blocker image.

Integrating irradiance from visible lights: If we assume that pixel $i \in \{1, 2, \ldots, M\}$ of the light source image is a point light source at position s_i with pixel power Φ_i, then its irradiance contribution at paper point $\mathbf{p}$ is

$$E_i(\mathbf{p}) = \frac{Z_{si}\Phi_i V(\mathbf{s}_i, \mathbf{p})}{4\pi\|\mathbf{s}_i - \mathbf{p}\|^3}. \tag{14}$$

Finally, the total irradiance at $\mathbf{p}$ is just the sum of the $M = m^2$ light source contributions,

$$E(\mathbf{p}) = \sum_{i=1}^{M} E_i(\mathbf{p}). \tag{15}$$

Such summations are performed at all M paper pixels to determine the paper's irradiance image.

Finally, we can heavily optimize the sum in (15) to render images produced by LightGrid basis functions. In those cases, the light matrix Φ_i is very sparse, with only approximately M/N terms nonzero, with similar sparsity in (15). Consequently, instead of rendering one image using $O(M^2)$ operations, we can render the entire N basis images in $O(M^2)$ operations! This speedup (along with an optimized Matlab implementation) helped make this assignment possible.

References

[1] Philip Dutré, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination.* A K Peters Ltd., 2006.