ECSE 343: Numerical Methods in Engineering

Assignment 3: 2D Deconvolution, Regularization and SVD

Due: Thursday, March 17th, 2022 at 11:59pm EST on myCourses
Final weight: **20%**

This assignment follows the *same* policies and processes as Assignment 0 (refer to section 1 in the A0 handout).

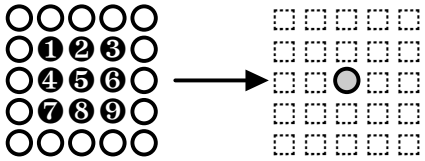
We suggest you take time to read the entire handout *before* starting your work.

Contents

1 Revisiting Image Convolution & Deconvolution

Let's revisit and *generalize* the image convolution and deconvolution problems we saw in Assignment 1. This time, instead of only consider a discrete 1D horizontal blur kernel, we will consider more general 2D blurs (i.e., horizontal *and* vertical).

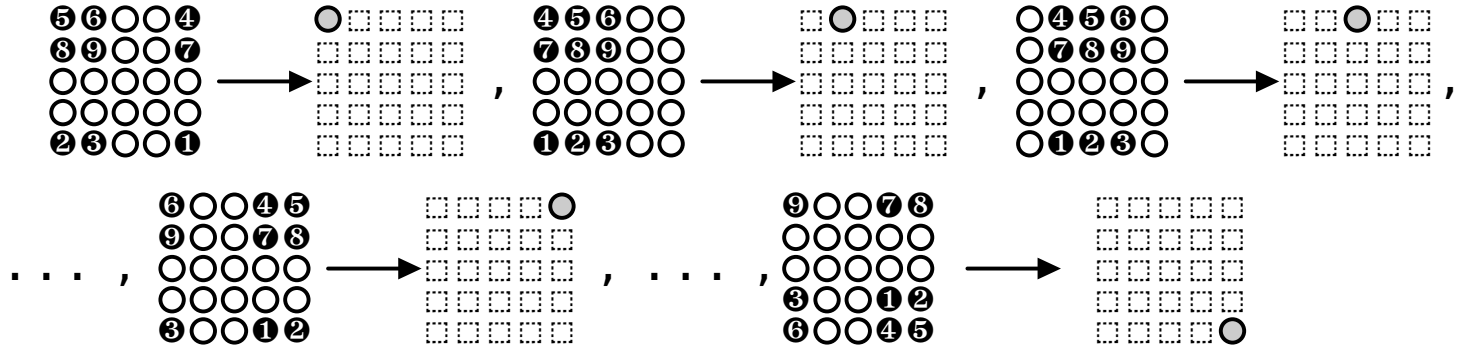
To illustrate the forward blurring process we extend our previous diagrams, however the "sweep and stamp" process still applies. Our diagram below illustrates a few instances of applying a 3×3 blur kernel (i.e., with 9 elements, which we've numbered for clarity: ❶, ❷, …, ❾) to a 5×5 input image, and generating a 5×5 output image, e.g.,



You can assume that we will only test **square** kernels, input/output image resolutions greater than or equal to the kernel size, and only square images. That being said, writing your code in a manner that *doesn't* assume square kernels can help with debugging/testing.

The kernel is centered — this time horizontally *and* vertically — on an input pixel at the location of the corresponding output pixel (illustrated as a circle with black contour and grey fill), and if portions of the kernel happen to fall outside the bounds of the input image, they wrap around (vertically *and* horizontally, now).

By "sliding" the kernel along each pixel of the uncorrupted input image, weighting the underlying source/input pixels by the overlapping kernel values and summing across these weighted input pixels, we construct the final blurred image one pixel at a time:



Recall, as in Assignment 1, while the diagrams above illustrate the blurring process *in 2D and for 2D images and kernels*, you will express this linear map as a matrix — and thus — must treat the input (and output) images as "flattened" 1D vectors. Treating this change of indexing from 2D pixel coordinates to flattened 1D "image" vector elements is one part of the learning objective for this task.

Deliverable 1 [25%]
Construct the blur matrix: Complete the implementation of `CreateBlurMatrix`. It takes as input a (k, k) `numpy.array` with the 2D blur kernel elements, as well as two positive integers denoting the `width` and `height` of the image the matrix will be applied to — you can assume that the corrupted and uncorrupted images have the same dimensions, and that all images are square (i.e., `width = height`). The routine returns the blur matrix as a `numpy.array` with shape (`width` $\times$ `height`, `width` $\times$ `height`).

1.1 Isotropic Gaussian Blur Kernel

One common blur kernel is the discrete isotropic Gaussian kernel. For one-dimensional blurs, the continuous kernel is defined as

$$G_{1\mathrm{D}}(x;\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \tag{1}$$

where σ is the standard deviation of the Gaussian, which defines its fall-off rate, and the continuous kernel is defined over the entire real line, i.e., for $-\infty \leq x \leq \infty$. In practice, one way to take this continuous kernel and discretize it to have length k is to evaluate the kernel at **integer** lattice values $x \in \left\{-\lfloor \frac{k}{2} \rfloor, \ldots, \lfloor \frac{k}{2} \rfloor\right\}$.

For two-dimensional blur kernels, the continuous isotropic Gaussian is

$$G_{2D}(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

and its discretized $k \times k$ version is obtained by evaluating on a 2D (x, y) integer lattice with $x \in \{-\lfloor k/2 \rfloor, \ldots, \lfloor k/2 \rfloor\}$ and $y \in \{-\lfloor k/2 \rfloor, \ldots, \lfloor k/2 \rfloor\}$. Note that G_{2D} can be decomposed as the product of two 1D Gaussians, a fact you can optionally leverage to simplify your implementation.

Deliverable 2 [10%]
2D Gaussian Blur Kernel: Complete the implementation of `Gaussian2D` to generate the 2D discretized Gaussian blur kernel. It takes as input the (odd) kernel extent k (i.e., k for a square 2D kernel of size (k, k)) and the standard deviation σ for the isotropic Gaussian `sigma`. It outputs a (k, k) `numpy.array` of the 2D Gaussian evaluated at integer lattice locations.

Consider different ways that you can test your `Gaussian2D` implementation, as well as how it can be used to test your implementation of `CreateBlurMatrix`.

1.2 Conditioning & Regularization

As in Assignment 1, we provide you with a benchmark image set. Here, in addition to providing the clean and blurred images, we also provide a version of the blurred image polluted with subtle noise. We will use this noisy blurred image to explore the conditioning of your blur operator.

In the current scenario, with a fully-constrained blur matrix $\mathbf{A}$ and blurred image $\mathbf{b}$ solving for the least-squares unpolluted image $\mathbf{x}$ as

$$\min_{\mathbf{x}} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2 \tag{3}$$

amounts to solving the original linear system as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. For many blurring kernels, the blur matrix can become ill-conditioned, meaning the solution can be catastrophically sensitive to minor deviations in the input $\mathbf{b}$.

We can employ regularization strategies to try to mitigate the poor conditioning of the system. One common strategy is Tikhonov regularization, which aims instead to solve the modified least-squares problem in

$$\min_{\mathbf{x}} \left(\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda^2 \|\mathbf{x}\|_2^2 \right) = \min_{\mathbf{x}} \left\| \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{A} \\ \lambda \mathbf{I} \end{bmatrix} \mathbf{x} \right\|_2^2, \tag{4}$$

where λ is the Tikhonov *regularization parameter* that allows you to control the degree of regularization. This *regularized* least-squares problem can be expressed as the solution to the following *augmented* system of linear equations:

$$(\mathbf{A}^{\mathrm{T}}\mathbf{A} + \lambda^2 \mathbf{I}^{\mathrm{T}}\mathbf{I})\,\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{b}\,. \tag{5}$$

Deliverable 3 [10%]
Regularized deblurring: Complete the implementation of `DeblurImage`. It takes as input the *unregularized* blur matrix, the (potentially noisy) blurred image, and the Tikhonov λ regularization constant `lambda_`. The function should construct the augmented/regularized linear system of equations and solve them (using whichever `numpy` routine you like) to solve for and output the (unflattened, 2D) deblurred image.

- You may find it useful to test your routines on *smaller* images.
- You may want to complete the rest of the assignment before returning to complete the next deliverable.

Tikhonov regularization is arguably the most basic regularization scheme but, even then, a judicious choice of its λ parameter can yield much better results than a more naïve setting. One generalization of the Tikhonov formulation,

$$\min_{\mathbf{x}} \left(\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_2^2 + \lambda^2 \|\mathbf{D}\mathbf{x}\|_2^2 \right) = \min_{\mathbf{x}} \left\| \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{A} \\ \lambda \mathbf{D} \end{bmatrix} \mathbf{x} \right\|_2^2, \tag{6}$$

admits an additional $\mathbf{D}$ operator — a so-called *regularization matrix* — that can be designed to specialize the regularization based on the needs of a specific application. In the case of image deblurring, some choices for $\mathbf{D}$ that can outperform vanilla Tikhonov regularization include local derivative and/or Laplacian operators.

Deliverable 4 [15%]
Deblurring competition: `DeblurImage` implements the basic Tikhonov regularization scheme, however other regularization methods for deblurring exist. Complete the implementation of `DeblurImageCompetition` to implement *any set of regularization approaches* with the goal of generating your best possible deblurred image result — here, you have **full latitude** in how you go about regularizing, augmenting, etc. the linear system in order to improve the deblurring result. The function signature is the same as `DeblurImage` from A1 (i.e., the same as `DeblurImage` above, but without `lambda_`; you should **hardcode** any regularization constants you use within your `DeblurImageCompetition` function body). Your grade in this deliverable will be based on *how well* your deblurring works compared to your colleagues', and how well you document your solution. Deblurring error will be computed as the L_2 difference between your deblurred image and the ground truth deblurred (i.e., source) image.

2 Solving Linear Systems with Singular Value Decomposition

Let's explore how **singular value decompositions** (SVDs) can be used to solve (*implicitly regularized*) linear systems. As a warm up, we will implement a simple *singular vector solver*, before moving on to the deconvolution problem using `numpy`'s built-in SVD algorithm. Recall that the SVD of $\mathbf{A}$ is $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$ and has $k \leq \min(m, n)$ non-zero singular values.

2.1 Power Iteration

We discussed how the Power Iteration Method could be used to compute the eigenvectors associated to the largest and smallest eigenvalues in a non-singular matrix $\mathbf{A}$. We can extend this approach to compute the left and right singular vectors associated to the largest and smallest singular values of $\mathbf{A}$, by applying the method to two separate eigenvector settings:

- by applying the power iteration method to $\mathbf{A}\mathbf{A}^{\mathrm{T}}$ you can compute the left singular vectors in $\mathbf{U}$ associated to the largest ($\mathbf{u}_1$ for σ_1) or smallest ($\mathbf{u}_k$ for σ_k) singular values, and
- by applying the power iteration method to $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ you can compute the left singular vectors in $\mathbf{V}$ associated to the largest ($\mathbf{v}_1$ for σ_1) or smallest ($\mathbf{v}_k$ for σ_k) singular values.

☑ **Deliverable 5 [15%]**
Power Iteration: Complete the implementation of `PowerMiniSVD` to return the left and right singular vectors associated to the **largest** singular value, $\mathbf{u}_1$ and $\mathbf{v}_1$. The function takes an input matrix and number of power iterations as input.

2.2 Rank-reduced Deconvolution

We can use the SVD to solve linear systems. Consider taking the SVD of your blur matrix as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$ and then solving for $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ in the unregularized deblurring problem. Clearly if we use the full-rank reconstruction of $\mathbf{A}$ as $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}$ we will run into conditioning-related instabilities when, e.g., the blurred image $\mathbf{b}$ is polluted by noise.

Instead of using the **full rank** SVD reconstruction of $\mathbf{A}$, you can implement a rank-reduced reconstruction that retains only a subset (i.e., a percentage up to 100%) of the system's "energy". To do so whilst retaining $0 \leq R \leq 1$ of the system's energy, you can form an effective rank-reduced approximation of $\mathbf{A}$ that only retains those singular values above the R-percentage threshold — and zeros out the remaining singular values.

Specifically, your rank-reduced approximation of $\mathbf{A}$ will include only those singular values $\{\sigma_1, \ldots, \sigma_r\}$ that satisfy this inequality:

$$\left(\sum_{i=1}^{r} \sigma_i^2 \Big/ \sum_{j=1}^{k} \sigma_j^2\right) \leq R\,. \tag{7}$$

Your rank-reduced $\hat{\mathbf{A}} = \mathbf{U}\hat{\mathbf{\Sigma}}\mathbf{V}^{\mathrm{T}}$ has $\hat{\mathbf{\Sigma}} = \mathrm{diag}(\sigma_1, \ldots, \sigma_r, 0, \ldots, 0)$ and can be used to solve

$$\mathbf{x} = \hat{\mathbf{A}}^{-1}\mathbf{b} = \mathbf{V}\hat{\mathbf{\Sigma}}^{-1}\mathbf{U}^{\mathrm{T}}\mathbf{b}\,. \tag{8}$$

Here, we assume that $r < k$ (i.e., $R < 1$), otherwise the degree to which $\hat{\mathbf{\Sigma}}$ is zero-padded would differ. Keep in mind that, while implementing Equation 8 as-is will not lead to incorrect results, there are several avenues for optimizing its performance without sacrificing any accuracy — in fact, some optimizations may *improve* the numerics of the solution.

Deliverable 6 **[25%]**
SVD Deblurring: Complete the implementation of `DeblurImageSVD` which takes as input the SVD factors of a blur matrix (computed outside the function), the blurred (and potentially noisy) input image, the amount of energy R to retain in the rank-reduced reconstruction of the blur matrix, and returns the deblurred image as in Equation 8.

Some notes:
• Computing the SVD is expensive. You may want to implement a caching scheme in your test code using `numpy.save` and `numpy.load`
• Do your rank-reduced image deblurring results *require* additional explicit regularization? You don't have to actually answer this in your solution, but it's worthwhile reflecting on.

3 You're Done!

Congratulations, you've completed the 3rd assignment. Review the submission procedures and guidelines at the start of the Assignment 0 handout before submitting your `Python` script file assignment solution.