ECSE 343: Numerical Methods in Engineering

Assignment 1: Matrix Factorizations and Solving Linear Systems

Due: Thursday, February 3rd, 2022 at 11:59pm EST on myCourses

Final weight: **15%**

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

The assignment deliverables are designed to be completed in the order they are presented. That being said, we suggest you take time to read the entire handout *before* starting your work.

Contents

1 LU Solver

You will implement a simplified LU linear system solver. This will comprise a bare bones LU decomposition, as well as the forward and backward substitution algorithms. Later in the assignment, you'll test your solver on polynomial regression problems.

1.1 LU Decomposition

Your first task is to decompose a matrix $\mathbf{A}$ as the product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$. Your decomposition will not treat pivoting. In this setting, the elements of $\mathbf{L}$ and $\mathbf{U}$ can be expressed algebraically as:

$$U_{ij} = \begin{cases} A_{ij} & \text{for } i = 0, \\ A_{ij} - \sum_{k=0}^{i-1} L_{ik}U_{kj} & \text{i} > 0 \end{cases} \tag{1}$$

$$L_{ij} = \begin{cases} \dfrac{A_{ij}}{U_{jj}} & \text{for } j = 0, \\ \dfrac{A_{ij} - \sum_{k=0}^{j-1} L_{ik}U_{kj}}{U_{jj}} & \text{j} > 0 \end{cases} \tag{2}$$

☑ **Deliverable 1 [10%]**
Perform a simplified LU decomposition: Use equations **(1)** and **(2)** to complete `LU_Decomposition` in the base code. The routine takes an (n, n) `numpy.array` $\mathbf{A}$ and returns the lower and upper triangular matrices $\mathbf{L}$ and $\mathbf{U}$.

1.2 Forward and Backward Substitution

Given a lower triangular linear system $\mathbf{Ly} = \mathbf{b}$, forward substitution solves for $\mathbf{y}$ as:

$$y_i = \begin{cases} \dfrac{b_1}{L_{11}} & \text{for } i = 1, \\ \dfrac{1}{L_{ii}}\left(b_i - \sum_{j=1}^{i-1} L_{ij}y_j\right) & \text{otherwise.} \end{cases} \tag{3}$$

☑ **Deliverable 2 [10%]**
Perform forward substitution: Use equation **(3)** to complete `ForwardSubstitution` in the base code. The routine takes a lower triangular (n, n) `numpy.array` $\mathbf{L}$ and a $(n,)$ `numpy.array` $\mathbf{b}$; it returns an $(n,)$ `numpy.array` $\mathbf{y}$.

Given an upper triangular linear system $\mathbf{Ux} = \mathbf{y}$, backward substitution solves for $\mathbf{x}$ as:

$$x_i = \begin{cases} \dfrac{y_n}{U_{nn}} & \text{for } i = n, \\ \dfrac{1}{U_{ii}}\left(y_i - \sum_{j=i+1}^{n} U_{ij}x_j\right) & \text{otherwise.} \end{cases} \tag{4}$$

☑ **Deliverable 3 [10%]**
Perform backward substitution: Use equation **(4)** to complete `BackwardSubstitution` in the base code. The routine takes an upper triangular (n, n) `numpy.array` $\mathbf{U}$ and a $(n,)$ `numpy.array` $\mathbf{y}$; it returns an $(n,)$ `numpy.array` $\mathbf{x}$.

You now have all the pieces needed to piece together a linear system solver. A solution to the general system $\mathbf{Ax} = \mathbf{LUx} = \mathbf{b}$ can be obtained by solving the two simplified systems:

$$\mathbf{Ly} = \mathbf{b} \text{ and } \mathbf{Ux} = \mathbf{y}\ . \tag{5}$$

☑ **Deliverable 4 [10%]**
Put together a simple solver: Complete the routine `LU_solver` to solve the linear system $\mathbf{Ax} = \mathbf{b}$ using your simplified LU decomposition, forward and backward substitution.

2 Solving Polynomial Regression

Let's test out your linear solver on a few polynomial regression problems. Here, you'll formulate polynomial regression problems as solutions to linear systems of equations, before applying your solver to them.

2.1 Polynomial Regression System

Given m data points (t_i, b_i) we want to find the $n-1$ degree polynomial

$$p_{n-1}(t) = \sum_{j=1}^{n} x_j\, t^{j-1}$$

that best fits the data points and where the coefficient vector $\mathbf{x} = [x_1, \ldots, x_n]$ fully defines the polynomial. We can formulate this problem as the solution to a linear system of equations

$$\mathbf{Ax} = \begin{bmatrix} 1 & t_1 & \cdots & t_1^{n-1} \\ 1 & t_2 & \cdots & t_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & t_m & \cdots & t_m^{n-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \mathbf{b}$$

where the Vandermonde matrix $\mathbf{A}$ can be formed using the independent variables of our data points, t_i.

Deliverable 5 [10%]
Given an $(m,)$ `numpy.array` $\mathbf{t} = [t_1, \ldots, t_m]$ and positive integer n, complete the implementation of `GetVandermonde` to construct and return an (m, n) `numpy.array` Vandermonde matrix $\mathbf{A}$ for the degree $n-1$ polynomial.

2.2 Fully-constrained and Overdetermined Polynomial Fitting

If the number of data points m matches the number of unknowns n, we can perfectly solve $\mathbf{Ax} = \mathbf{b}$. Consequently, the Vandermonde matrix $\mathbf{A}$ here would be square.

If, on the other hand, we have more data points m than degrees of freedom for our polynomial, we have an overdetermined problem. A perfect fit will not generally exist, but we can solve for the fit that minimizes the squared residual 2-norm $||\mathbf{Ax} - \mathbf{b}||_2^2$. The *normal equations* allow us to express the least-squares solution using the modified system $\mathbf{A}^{\mathrm{T}}\mathbf{Ax} = \mathbf{A}^{\mathrm{T}}\mathbf{b}$.

Deliverable 6 [20%]
Solve the polynomial regression problem: Complete the implementation of `PolynomialFit`. It takes as input an $(m, 2)$ `numpy.array` of the m data point (t_i, b_i) and a positive integer n to denote the polynomial degree $(n-1)$. You should use your `GetVandermonde` and `LU_solver` routines.

3 Image Reconstruction using Deconvolution

Most real-world problems are sufficiently complex to require more robust solvers than what we developed above: without pivoting and careful performance-minded vectorization/implementation, your LU solver won't likely scale to larger problems.

Setting up a problem and *using* a solver is just as important as know how to *write* one. The final set of deliverables will focus on understanding a more complex problem, formulating its solution as a linear system, and solving it with an industry-caliber LU solver.

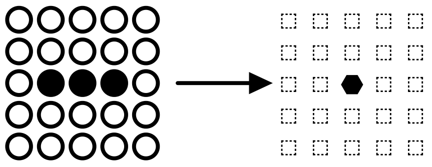
3.1 Image Filtering — a motivating example

Imagine taking a photo with your smartphone only to realize, after the fact, that the photo came out blurry. Luckily, your phone's accelerometer was able to register a horizontal motion at the time of capture.

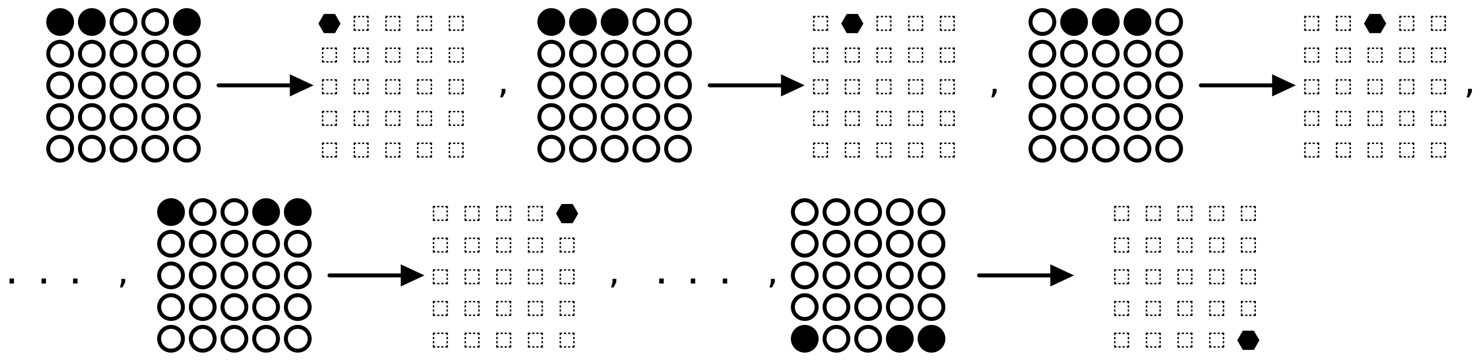
We could model the *forward process* that polluted the image as a linear operator. Specifically, by *convolving* the discretized (1D) horizontal blur *kernel* (that the accelerometer registered) along the horizontal axis of the *uncorrupted* image, we can arrive at the blurry image.

The blur kernel, which we'll visualize as a few dots ●●●, has an odd number of elements, each with a numerical weight. You can imagine centering the kernel atop every pixel in the original uncorrupted image, weighting each pixel it overlaps with by the corresponding kernel value, and summing accross the weighted pixel values in order to obtain a corrupted image pixel value (⬣, below). When any part of the kernel falls outside the bounds of the image, it "loops over" to the opposing end of the image.

For example below, to compute the ⬣ pixel value in the corrupted image on the right, we weight the intensities of solid pixels from the uncorrupted image on the left by the one overlapping element (of three elements, in this example) of the blur kernel.



By "sliding" the kernel, and repeating this weighted sum, along each pixel of each row of the uncorrupted image, we construct the final blurred image:



3.2 Blurring an Image

If we *flatten* the uncorrupted input image row-wise into a single $n \times 1$ column vector $\mathbf{x}$, where n is the number of image pixels, then we can model this linear corruption process with a matrix $\mathbf{A}$ that relies only on the $k \times 1$ blur kernel elements.

✓ **Deliverable 7 [20%]**
Construct the blur matrix: Complete the implementation of `CreateBlurMatrix`. It takes as input a $(k, 1)$ `numpy.array` with the 1D horizontal blur kernel elements, as well as two positive integers denoting the `width` and `height` of the images — we assume that the corrupted and uncorrupted images have the same dimensions.

Once you form $\mathbf{A}$, you can obtain the linearized corrupted image $\mathbf{b}$ (of size $n \times 1$) as $\mathbf{A}\,\mathbf{x}$.

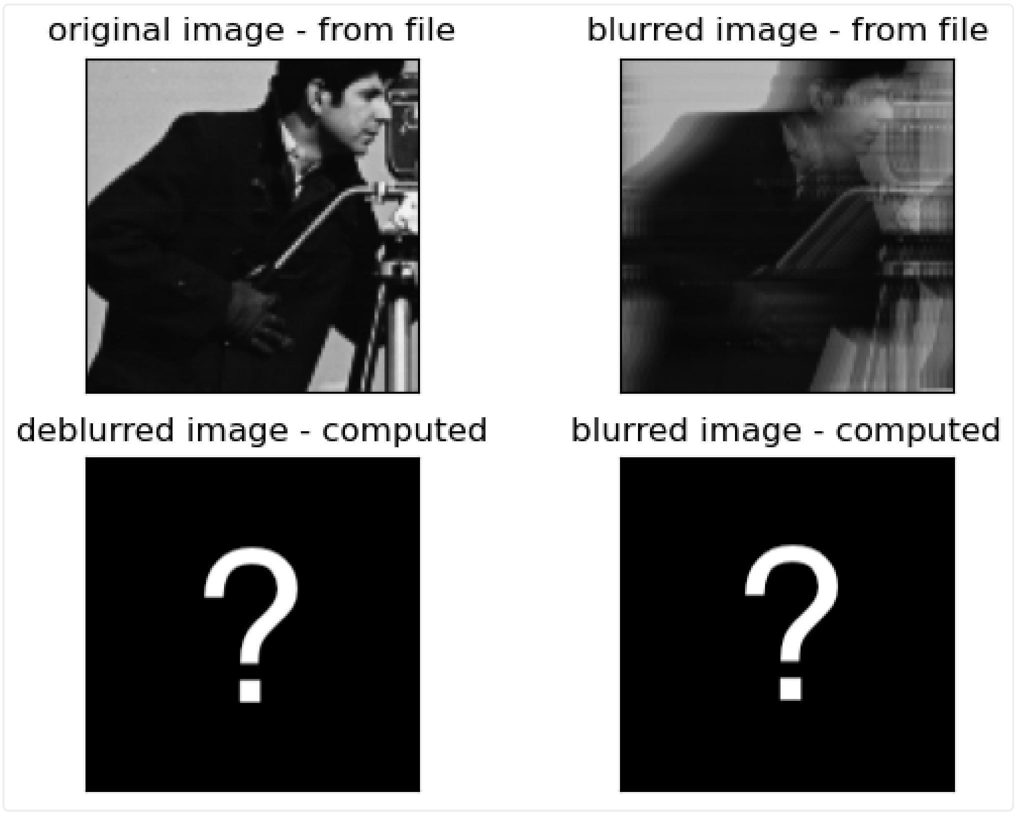
✓ **Deliverable 8 [5%]**
Blur an image: Complete the implementation of `BlurImage`. It takes as input a (n, n) `numpy.array` of the blur matrix computed with `CreateBlurMatrix` and a (`width`,`height`) `numpy.array` of the uncorrupted grayscale image. It should output a (`width`, `height`) `numpy.array` of the **corrupted** grayscale image.

3.3 Deblurring an Image

We're lucky that, in our setting, the accelorometer was able to provide us with an estimate of the blurring kernel. Given this, and the *forward model* of how the uncorrupted image was corrupted with the kernel, we can solve the *inverse* problem: given only the corrupted image and the blur kernel, we aim to **retrieve** the uncorrupted image. This problem is referred to a *non-blind deconvolution*; if we were **not** given the blur kernel, the (much harder) problem is referred to as *blind deconvolution*.

Here, we can retriev uncorrupted image $\mathbf{x}$ given the blur matrix $\mathbf{A}$ derived from the 1D horizontal blur kernel and the corrupted (blurred) image $\mathbf{b}$ by solving for $\mathbf{x}$ in $\mathbf{A}\mathbf{x} = \mathbf{b}$.

✓ **Deliverable 9 [5%]**
Deblur an image: Complete the implementation of `DeblurImage`. It takes as input a (n, n) `numpy.array` of the blur matrix computed with `CreateBlurMatrix` and a (`width`,`height`) `numpy.array` of the **corrupted** grayscale image. It should output a (`width`, `height`) `numpy.array` of the **uncorrupted** grayscale image. **Note: Your LU solver will not scale to the sizes of images we will be using. You should use `scipy`'s LU solving routines, which we have imported for you.**


original image - from file
blurred image - from file
deblurred image - computed
blurred image - computed
?
?

The test images we provide you for deliverables seven through nine. Your code will generate the two missing images, which should match those of the upper row.

4 You're Done!

Congratulations, you've completed the 1st assignment. Review the submission procedures and guidelines at the start of the Assignment 0 handout before submitting your `Python` script file assignment solution.