CSCI 2033 Assignment 2: LU Decomposition

Posted: Friday, October 20
Last updated: Wednesday, October 25

Due date: Thursday, November 2, 11:55 pm

In this assignment, you will implement a MATLAB function to decompose a matrix into lower and upper triangular matrices (**L** and **U**), i.e., **PA** = **LU** where **P** is a row permutation matrix, and apply it to solve a computational physics problem.

1 Download

For Section 6, we provide codes that can compute forces in a simple physical system and visualize the results. **This code has recently been updated.** Download the file `sec6code.zip` from the Moodle submission page, and extract it into the folder where you will be working on the assignment. You should obtain the files `get_forces.m`, `draw.m`, and a directory `+defo`.

2 Submission Guidelines

You will submit a zip file that contains the following `.m` files on Moodle:

- `replacement_lu.m`
- `interchange_lup.m`
- `my_lu.m`
- `my_lup.m`
- `build_matrix.m`
- `get_displacements.m`

Note 1: Each file must be named as specified above.

Note 2: Each function must return the specified value.

Note 3: You may use MATLAB's high-level array manipulation syntax, e.g. `size(A)`, `A(i,:)`, and `[A,B]`, but the built-in linear algebra functions such as `inv`, `lu`, `rref`, and `A\b` are *not* allowed (except in Section 6). Please contact the TAs for further questions.

Note 4: No collaboration is allowed. Plagiarism cases will be directly reported to the University.

3 Elementary Row Operations (4 points)

You will implement two simple modular functions that will be useful for LU decomposition. These are very similar to the functions you implemented in Assignment 1.

Notation: for subsequent sections, A_i indicates the i^{th} row of $\mathbf{A}$, and A_{ij} indicates the (i, j) entry of $\mathbf{A}$.

Specification:

`function [U_new, L_new] = replacement_lu(U, i, j, s, L)`
Input: two square matrices $\mathbf{U}$ and $\mathbf{L}$, two integers i and j, and a scalar s.
Output:

- $\mathbf{U}_{\text{new}}$: updated $\mathbf{U}$ by subtracting s times row j from row i, i.e. performing the row replacement operation $U_i \leftarrow U_i - sU_j$.
- $\mathbf{L}_{\text{new}}$: updated $\mathbf{L}$ by filling in its (i, j) entry with s, i.e., $L_{ij} \leftarrow s$.

Algorithm 1 `replacement_lu`

1: $n \leftarrow$ the number of columns of U
2: $L_{ij} \leftarrow s$
3: **for** $1 \leq k \leq n$ **do**
4: $\quad U_{ik} \leftarrow U_{ik} - sU_{jk}$ ▷ Row replacement
5: **end for**
6: return L, U

Warning! If you are adapting your code from Assignment 1, be careful that we are now *subtracting* s times row j from row i instead of adding it.

`function [U_new, L_new, P_new] = interchange_lup(U, i, j, L, P)`
Input: three same size square matrices $\mathbf{U}$, $\mathbf{L}$, and $\mathbf{P}$, and two integers i and j.
Output:

- $\mathbf{U}_{\text{new}}$: updated $\mathbf{U}$ by swapping rows i and j, i.e. $U_i \leftrightarrow U_j$.
- $\mathbf{L}_{\text{new}}$: updated $\mathbf{L}$ by swapping *only* the first $(i-1)$ entries of rows i and j, i.e., $L_{i,1} \leftrightarrow L_{j,1}$, $\ldots$, $L_{i,(i-1)} \leftrightarrow L_{j,(i-1)}$ as shown in Figure 1.
- $\mathbf{P}_{\text{new}}$: updated $\mathbf{P}$ by swapping rows i and j, i.e. $P_i \leftrightarrow P_j$.

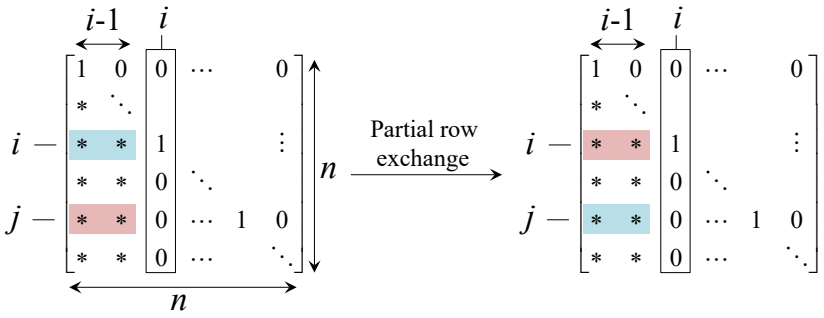

Figure 1: Partial row exchange in $\mathbf{L}$.

Algorithm 2 `interchange_lup`

1: $U_i \leftrightarrow U_j$ ▷ Row interchange
2: $P_i \leftrightarrow P_j$ ▷ Row interchange
3: **if** $i > 1$ **then**
4: **for** $1 \leq k \leq i-1$ **do**
5: $L_{ik} \leftrightarrow L_{jk}$ ▷ Partial row interchange
6: **end for**
7: **end if**
8: return L, U, P

4 LU Decomposition **(4 points)**

In this part, you will write a function that performs LU decomposition without pivoting. We will deal with pivoting in the next part of the assignment.

Specification:

`function [L, U] = my_lu(A)`
Input: an $n \times n$ square matrix $\mathbf{A}$.
Output:

- $\mathbf{L}$: an $n \times n$ lower triangular matrix where the diagonal entries are all one, e.g., $\begin{bmatrix} 1 & 0 & 0 \\ * & 1 & 0 \\ * & * & 1 \end{bmatrix}$ where $*$ is a potentially nonzero element.
- $\mathbf{U}$: an $n \times n$ upper triangular matrix.

To get full credit, your function should handle the following case:

- Early termination: Due to round-off error, there may exist free variables whose pivots are extremely small but not precisely zero. You should terminate your LU decomposition if the absolute value of a pivot is less than 10^{-12}.

The process of LU decomposition uses Gaussian elimination that transforms $\mathbf{A}$ to an upper triangular matrix $\mathbf{U}$ while recording the pivot multipliers in a lower triangular matrix $\mathbf{L}$.

1. Initialize $\mathbf{L}$ to the identity matrix, and $\mathbf{U}$ to $\mathbf{A}$. You can use MATLAB's built-in function `eye(n)`.
2. At the i^{th} step, for each row k below the i^{th} row,
   (a) Record the pivot multiplier to $\mathbf{L}$ at (k, i), i.e., $\mathbf{L}_{k,i} = \mathbf{U}_{k,i}/\mathbf{U}_{i,i}$ as shown in Figure 2. Note that this fills in the i^{th} column of $\mathbf{L}$.
   (b) Reduce the k^{th} row of $\mathbf{U}$ using the pivot multiplier, i.e., $\mathbf{U}_k = \mathbf{U}_k - (\mathbf{U}_{k,i}/\mathbf{U}_{i,i})\mathbf{U}_i$ where $\mathbf{U}_i$ is the i^{th} row.

$$\mathbf{L} = \begin{array}{c} \\ i - \\ \\ \\ k - \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccc} & & i & & & \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ * & \ddots & & & \\ * & * & 1 & & \vdots \\ * & * & \vdots & \ddots & \\ * & * & U_{ki}/U_{ii} & \cdots\ 1 & 0 \\ * & * & \vdots & \cdots & \ddots \end{bmatrix} \end{array} \qquad \mathbf{U} = \begin{array}{c} \\ i - \\ \\ \\ k - \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccc} & & i & & & \end{array} \\ \begin{bmatrix} U_{11} & * & * & \cdots & * \\ 0 & \ddots & & & \\ 0 & 0 & U_{ii} & & \vdots \\ 0 & 0 & 0 & \ddots & \\ 0 & 0 & 0 & \cdots\ * & * \\ \vdots & \vdots & \vdots & \cdots & \ddots \end{bmatrix} \end{array}$$

Figure 2: LU decomposition at the i^{th} step.

We provide pseudocode for LU decomposition in Algorithm 3.

Algorithm 3 LU decomposition of A

$n \leftarrow$ the number of columns of A
$L \leftarrow I_n$ where I_n is $n \times n$ identity matrix.
$U \leftarrow A$
for $1 \leq i \leq n-1$ **do**
    if U_{ii} is nearly zero* **then**
        return L, U ▷ Early termination
    end if
    for $i+1 \leq k \leq n$ **do**
        $p \leftarrow U_{ki}/U_{ii}$
        $[U, L] =$ `replacement_lu`(U, k, i, p, L)
    end for
end for
return L, U

*Consider a number to be nearly zero if its absolute value is smaller than 10^{-12}.

Note: When terminating early, $\mathbf{L}$ is not *unique*, i.e., the order of rows corresponding to zero rows in $\mathbf{U}$ can be different. As long as the resulting decomposition satisfies $\mathbf{A} = \mathbf{LU}$, the solution $\mathbf{L}$ and $\mathbf{U}$ are valid.

Test cases:

- `[L,U] = my_lu([4,-2,2;-2,5,3;2,3,9])`
  $\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 0.5 & 1 & 1 \end{bmatrix}$, $\mathbf{U} = \begin{bmatrix} 4 & -2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 4 \end{bmatrix}$
- Early termination:
  `[L,U] = my_lu([4,-2,2,1;-2,5,3,3;4,-2,2,1;-2,5,3,3])`
  $\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -0.5 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ -0.5 & 1 & 0 & 1 \end{bmatrix}$, $\mathbf{U} = \begin{bmatrix} 4 & -2 & 2 & 1 \\ 0 & 4 & 4 & 3.5 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$.
  Without early termination, it will return $\mathbf{L}$ and $\mathbf{U}$ with `NaN` (Not-a-Number) entries.
- You can test your algorithm by constructing a random $n \times n$ matrix $\mathbf{A}$, e.g., `A=rand(10,10)`, and testing whether multiplying $\mathbf{L}$ and $\mathbf{U}$ reconstructs $\mathbf{A}$. You also need to check whether $\mathbf{L}$ is a lower triangular matrix with diagonal entries equal to 1, and $\mathbf{U}$ is an upper triangular matrix.

5 LU Decomposition with Partial Pivoting
(4 points)

Based on your `my_lu`, you will write numerically stable LU decomposition with *partial pivoting*. At the i^{th} step of LU decomposition (i^{th} pivot column), you will find the row that has the largest absolute value in the pivot column (say row j), and swap the i^{th} and j^{th} rows of $\mathbf{U}$ as usual. Simultaneously, you will swap the *partial* entries of the i^{th} and j^{th} rows of $\mathbf{L}$, and record the row exchange in a permutation matrix $\mathbf{P}$. For further details, please see

`http://www.math.kent.edu/~reichel/courses/intr.num.comp.1/fall09/lecture9/lecture4.pdf`

Specification:

`function [L, U, P] = my_lup(A)`
Input: an $n \times n$ square matrix $\mathbf{A}$.
Output:

- $\mathbf{L}$: an $n \times n$ lower triangular matrix where the diagonal entries are all 1.
- $\mathbf{U}$: an $n \times n$ upper triangular matrix.
- $\mathbf{P}$: an $n \times n$ permutation matrix.

The process of LU decomposition with partial pivoting needs to compute an additional row permutation matrix $\mathbf{P}$.

1. Initialize $\mathbf{L}$ and $\mathbf{P}$ to the identity matrix, and $\mathbf{U}$ to $\mathbf{A}$. You can use MATLAB's built-in function `eye(n)`.
2. At the i^{th} step,
   (a) Similar to Assignment 1, perform partial pivoting in $\mathbf{U}$.
   (b) Record the row exchange to the permutation matrix $\mathbf{P}$ by exchanging its corresponding rows.
   (c) Exchange the corresponding row entries in $\mathbf{L}$ to reflect the row exchange in $\mathbf{U}$. Note that this exchange only involves with the row entries that have been recorded, i.e., not entire row exchange.
   (d) For each row k below the i^{th} row, record the pivot multiplier to $\mathbf{L}$ and replace the row in $\mathbf{U}$ using the pivot multiplier, like in the previous part of this assignment.

We provide pseudocode for LUP decomposition in Algorithm 4.

Algorithm 4 LUP decomposition of A

```
n ← the number of columns of A
L ← I_n where I_n is n × n identity matrix.
P ← I_n                                          ▷ Permutation initialization
U ← A
for 1 ≤ i ≤ n − 1 do
    j ← the row index of the largest absolute value among rows i to n in the i^th column of U
    [U, L, P] = interchange_lup(U, i, j, L, P)
    if U_ii is nearly zero* then
        return L, U, and P                       ▷ Early termination
    end if
    for i + 1 ≤ k ≤ n do
        p ← U_ki/U_ii
        [U, L] = replacement_lu(U, k, i, p, L)
    end for
end for
return L, U, P
```

The red lines specify the major modification for partial pivoting.
*As before, consider a number to be nearly zero if its absolute value is smaller than 10^{-12}.

Note: When terminating early, **L** and **P** are not *unique*, i.e., the order of rows corresponding to zero rows in **U** can be different. As long as the resulting decomposition satisfies $\mathbf{PA} = \mathbf{LU}$, the solution **L** and **P** are valid.

Test cases:

- Partial pivoting:
  `[L,U,P] = my_lup([-2,5,3;2,3,9;4,-2,2])`
  $\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.5 & 1 & 1 \end{bmatrix}$, $\mathbf{U} = \begin{bmatrix} 4 & -2 & 2 \\ 0 & 4 & 8 \\ 0 & 0 & -4 \end{bmatrix}$, and $\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$
- Early termination:
  `[L,U,P] = my_lup([4,-2,2;-2,5,3;4+5E-13,-2+2E-13,2+7E-13])`
  $\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ -0.5 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$, $\mathbf{U} = \begin{bmatrix} 4 & -2 & 2 \\ 0 & 4 & 4 \\ 0 & 0 & 0 \end{bmatrix}$, and $\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$
- You can test your algorithm by comparing its results with MATLAB's built-in function, `[l, u, p] = lu(A)`, on a randomly generated $n \times n$ matrix A, e.g., `A=rand(10,10)`.

6 Force-Displacement Computations (3 points)

Numerical linear algebra is used extensively in computational physics, with applications in a wide range of fields including science, engineering, robotics, and animation. Often, we represent a physical object as a collection of vertices, and we have a computational model that can predict the internal forces that would result from any given deformation of the object (i.e. any given displacement of the vertices). Linear algebra allows us to do the reverse: apply any specified external forces on the object, and predict how it will deform as a result.

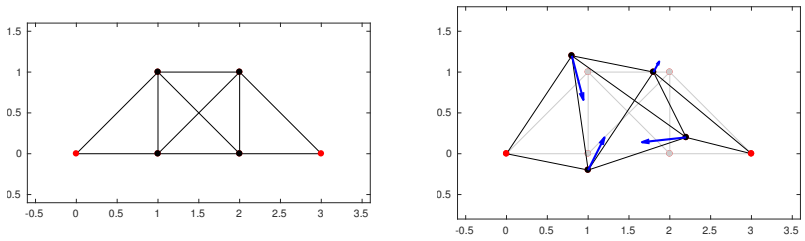

Figure 3: Left: The rest shape of a simple system with four free vertices. Right: A displacement of the vertices, and the resulting internal forces.

For example, consider the object above which has four free vertices (in black) that can move around, and two constrained vertices (in red) that remain fixed. We can collect the 2D displacements of the 4 free vertices into a single 8-dimensional vector $\mathbf{d}$. We have provided for you a function `f = get_forces(d)` that takes this vector of displacements $\mathbf{d}$ as input and returns the resulting forces $\mathbf{f}$ on the vertices, similarly packed into an 8-dimensional vector. In fact, the forces are linear in the displacements, so the function is really a linear transformation `get_forces` : $\mathbb{R}^8 \to \mathbb{R}^8$, and there exists some 8×8 matrix $\mathbf{A}$ such that $\mathbf{f} = \mathbf{Ad}$.

To visualize this relationship, we have provided a function `draw()`, which can either be called without arguments to draw the rest shape, or with two arguments `draw(d,f)` to visualize a deformed shape with forces as arrows. Try choosing a vector $\mathbf{d} \in \mathbb{R}^8$ with small entries and calling `draw(d, get_forces(d))` to produce something like Fig. 3 (right). (Actually, you may have to draw `0.1*get_forces(d)` because the forces are very large.)

Your task is to implement a function `d = get_displacements(f, ...)` that computes the inverse of this transformation: given the internal forces on the free vertices, find the displacements that would give rise to them. To do this, you will have to find the matrix $\mathbf{A}$ for the transformation, compute its LUP factorization, and then use the factorization to quickly solve $\mathbf{Ad} = \mathbf{f}$ to get $\mathbf{d}$ for any given $\mathbf{f}$.

Specification:

`A = build_matrix()`
Input: None.
Output: the 8×8 matrix A corresponding to the linear transformation `get_forces`. For any vector $\mathbf{d} \in \mathbb{R}^8$, the matrix-vector product `A*d` should equal `get_force(d)`.
Hint: Construct the vector $\mathbf{e}_1 = [1, 0, \ldots, 0]^T$ and call `get_forces` on it. What you get must be the first column of $\mathbf{A}$. Do the same for the rest of the columns.

Once you have $\mathbf{A}$, use either `my_lup` or MATLAB's `lu` to compute its LUP factorization. You will need it for the following function:

`d = get_displacements(f, L, U, P)`
Input: a vector of forces $\mathbf{f} \in \mathbb{R}^8$, and the LUP decomposition of $\mathbf{A}$.
Output: the vector of vertex displacements $\mathbf{d}$ that result in these internal forces, i.e. the solution of $\mathbf{Ad} = \mathbf{f}$.

Since $\mathbf{PA} = \mathbf{LU}$, we can write

$$\mathbf{PAd} = \mathbf{Pf} \qquad \Longleftrightarrow \qquad \mathbf{L}\overbrace{\mathbf{Ud}}^{\mathbf{y}} = \overbrace{\mathbf{Pf}}^{\mathbf{g}}.$$

Thus we can compute $\mathbf{d}$ in three steps: compute the vector $\mathbf{g} = \mathbf{Pf}$, solve $\mathbf{Ly} = \mathbf{g}$ for $\mathbf{y}$, and then solve $\mathbf{Ud} = \mathbf{y}$ for $\mathbf{d}$. In this function, you may use MATLAB's backslash operator `A\b` to perform the two solves, since Matlab automatically performs backsubstitution if the matrix is triangular.

> **Note:** In the original code we provided, it turned out that the system matrix you obtained from `build_matrix` did not require any pivoting, so you did not actually need $\mathbf{P}$ to get the right answer. We have now provided an updated version of the code which does induce pivoting. (It still models the same system, but the ordering of the entries in $\mathbf{f}$ is different.) Download the updated code and verify that your solution still works on it.

Once you have these functions working, you can compute the deformation caused by any *external* forces $\mathbf{f}_{\text{ext}}$: The resulting displacement is the one where the internal and external forces cancel out, so `d = get_displacements(-f_ext, ...)` with a negative sign. For example, choose $\mathbf{f}_{\text{ext}} = [0, 0, 0, 0{-}0.2, -0.2, -0.2, -0.2]^T$ to apply a downward gravity force to all vertices, compute $\mathbf{d}$, and visualize the result using `draw(d, f_ext)`.

Test cases:

These have been updated for the new version of the provided code.

For verification, we give the top left 2×2 block of the matrix A you should obtain:

$$A \approx \begin{bmatrix} -23.5355 & -3.5355 & \cdots \\ 0 & 0 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

Pulling the fourth vertex upwards with 1 unit of force, that is, `f_ext = [0;0;0;0; 0;0;0;1]`, should produce a displacement `d = get_displacements(-f_ext, L,U,P)` of approximately

$$\mathbf{d} \approx \begin{bmatrix} 0.0185 \\ 0.1432 \\ -0.0269 \\ 0.1212 \\ 0.0149 \\ 0.1950 \\ 0.0177 \\ 0.2063 \end{bmatrix}.$$