

## HW04 Hints

Check out the tutorial/demonstration videos on Gaussian elimination found in Module 2 Supplementary Resources page.

2. Read Section 2.1–2.7 of **NCM**, which is freely available at <https://www.mathworks.com/moler/chapters.html>. The link is also found in Module 2 Supplementary Resources page. The code `lutx` found in Section 2.7 may be particularly helpful.

In addition, the exercise on p. 46 of Module 2 lecture slides will provide a good starting point since PLU factorization is equivalent to the *pivoted* Gaussian elimination. Below is an answer to the exercise.

```
function x = GEpp(A, b)
% GEPP pivoted Gaussian Elimination (instructional version)
% Input:
%   A = square matrix
%   b = right-hand side vector
% Output:
%   x = vector solving A*x = b
%   row reduction to upper triangular system
S = [A, b]; % augmented matrix
n = size(A, 1);
for j = 1:n-1
    [~, iM] = max(abs(A(j:end, j))); % index of pivot element
    iM = iM + j - 1; % adjusted index
    if j ~= iM % pivot if necessary
        piv = [j iM];
        S(piv, :) = S(flip(piv), :).
    end
    for i = j+1:n
        mult = -S(i, j)/S(j, j); % R_i --> R_i + (-S(i, j)/S(j, j))
                               *R_j
        S(i, :) = S(i, :) + mult*S(j, :);
    end
end
% back subs (pretending that there is no 'backsub' routine)
U = S(:, 1:end-1);
beta = S(:, end);
x(n) = beta(n)/U(n, n);
for i = n-1:-1:1
    x(i) = (beta(i) - U(i, i+1:end)*x(i+1:end))/U(i, i);
end
end
```

Note that it is written as a self-contained program which contains a hard-coded backward substitution routine. As a last bit of hint, think about how the multipliers  $a_{i,j}/a_{j,j}$  are related to  $\ell_{i,j}$ , the  $(i,j)$ -entry of  $L$ .

3. For part (a), recall the following properties of the determinant from linear algebra:

- $\det(AB) = \det(A)\det(B)$ .
- $\det(T) = t_{11}t_{22}\cdots t_{nn} = \prod_{i=1}^n t_{ii}$ , where  $T \in \mathbb{R}^{n \times n}$  is a triangular matrix.

For part (b), try to vectorize your code. The function can be written in only couple lines (excluding the function header and the end statement) without using any loop. It would be also useful to recall that if you only want to generate the second output of a certain function, you put `~` in place of the first output argument as a placeholder. For instance, if you only need `U` from `mylu(A)`, instead of calling it with `[L,U] = mylu(A)`, use

```
[~, U] = mylu(A);
```

4. Consider the following example question.

**Question.** How would you code  $\mathbf{x} = AB\mathbf{b}$  most efficiently in MATLAB, and how many *flops* are needed?

**Answer.** There are only two ways<sup>1</sup> to code it: `x=A*(B*b)` or `x=(A*B)*b`. If the former is used, MATLAB

- (a) carries out `B*b`: matrix-by-vector multiplication ( $\sim 2n^2$  *flops*)
- (b) left-multiplies the previous result by `A`: another matrix-by-vector multiplication ( $\sim 2n^2$  *flops*)

In total, it takes  $\sim 4n^2$  *flops*. On the other hand, the latter version requires the matrix-by-matrix multiplication `A*B`, which already takes  $\sim 2n^3$  *flops*. So the former is the efficient one.

**Lesson.** Avoid matrix-by-matrix multiplication as much as possible!

Also remember to use the backslash operator if matrix inversion is required, instead of `inv`. When `\` is invoked, MATLAB in general does a pivoted Gaussian elimination, which takes  $\sim (2/3)n^3$  *flops*.

5. The following hints come from **LM**.

- For part (b), the alternate definition given in Equation (10.20) on p. 1481 is useful.
- For part (e), the last bullet point on p. 1481 is helpful.

---

<sup>1</sup>This is the same as `x = A*B*b`.