Factorizations

One of the main tools to solve linear systems: factorizations.

Idea: break up a matrix into pieces “easier to invert”.

For instance: $A\mathbf{x} = \mathbf{b}$, with $A \in \mathbb{R}^{m \times m}$ square invertible.

- Compute $A = QR$. Now $\mathbf{x} = A^{-1}\mathbf{b} = R^{-1}(Q^T\mathbf{b})$. $\frac{4}{3}m^3$
- Compute $\mathbf{c} = Q^T\mathbf{b}$. $O(m^2)$
- Compute $\mathbf{x} = R^{-1}\mathbf{c}$ by solving $R\mathbf{x} = \mathbf{c}$ via back-substitution. $O(m^2)$

Similarly you can use $A = U\Sigma V^T$ and $A = LU$.

Added benefit: sometimes the same factorization can be reused to solve more than one linear system.

Review of LU / Gaussian elimination

Gaussian elimination can be seen as a factorization, too: $A = LU$.

Add multiples of row 1 to rows 2…n to eliminate $A_{2:end,1}$:

$$\begin{bmatrix} 1 & & & & \\ * & 1 & & & \\ * & & 1 & & \\ * & & & 1 & \\ * & & & & 1 \end{bmatrix} \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * & * \\ \color{red}{0} & * & * & * & * \\ \color{red}{0} & * & * & * & * \\ \color{red}{0} & * & * & * & * \\ \color{red}{0} & * & * & * & * \end{bmatrix}$$

$$L_1 A = A_1.$$

$$(L_1)_{k1} = -\frac{A_{k1}}{A_{11}}, \quad k = 2, 3, \ldots, m.$$

LU factorization

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & * & 1 & & \\ & * & & 1 & \\ & * & & & 1 \end{bmatrix} \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}$$

$$L_1 A_1 = A_2,$$

$$(L_2)_{k2} = \frac{(A_1)_{k2}}{(A_1)_{22}}, \quad k = 3, \ldots, m.$$

Then go on:

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & * & 1 & \\ & & * & & 1 \end{bmatrix} \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix}$$

$$\begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & * & 1 \end{bmatrix} \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}$$

LU factorization

At the end, we have

$$L_{m-1}L_{m-2}\dots L_1 A = U,$$

with U upper triangular, or

$$A = \underbrace{L_1^{-1}L_2^{-1}\dots L_{m-1}^{-1}}_{=L} U.$$

where U upper triangular and L lower triangular.

Theorem

Any matrix $A \in \mathbb{R}^{m\times m}$ for which we do not encounter zero pivots in the algorithm admits a factorization $A = LU$, where L is lower triangular with ones on its diagonal, and U is upper triangular.

‘Stroke of luck’ (as Trefethen–Bau put it)

The product of the L_i^{-1}’s can be computed with zero operations:

$$\begin{bmatrix} 1 &&&& \\ -a_2 & 1 &&& \\ -a_3 && 1 && \\ -a_4 &&& 1 & \\ -a_5 &&&& 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 &&&& \\ & 1 &&& \\ & -b_3 & 1 && \\ & -b_4 && 1 & \\ & -b_5 &&& 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 &&&& \\ & 1 &&& \\ && 1 && \\ && -c_4 & 1 & \\ && -c_5 && 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 &&&& \\ & 1 &&& \\ && 1 && \\ &&& 1 & \\ &&& -d_5 & 1 \end{bmatrix}^{-1}$$

$$= \begin{bmatrix} 1 &&&& \\ a_2 & 1 &&& \\ a_3 & b_3 & 1 && \\ a_4 & b_4 & c_4 & 1 & \\ a_5 & b_5 & c_5 & d_5 & 1 \end{bmatrix}$$

Theorem

Suppose we can perform Gaussian elimination without row exchanges on $A \in \mathbb{R}^{m \times m}$, obtaining an upper triangular matrix U. Then, $A = LU$, where L is lower triangular with ones on the diagonal, containing the ‘multipliers’ used when computing

$$(\text{row } i) \leftarrow (\text{row } i) - L_{ik}(\text{pivot row } k).$$

LU factorization — code

```
function [L, U] = lu_factorization(A)
m = size(A, 1);
L = eye(m);
U = A;
for k = 1 : m - 1
  % invariant: L*U == A
  % compute multipliers
  L(k+1:end, k) = U(k+1:end, k) / U(k, k);
  % update U
  U(k+1:end, k) = 0;
  U(k+1:end, k+1:end) = U(k+1:end, k+1:end) ...
                         - L(k+1:end, k) * U(k, k+1:end);
end
```

Cost (for a dense matrix): $\frac{2}{3}m^3 + O(m^2)$: half as much as QR factorization.

Use to solve linear systems

```
function x = solve_system_lu(A)
[L, U] = lu_factorization(A);
c = L \ b;
x = U \ c;
```

Useful point to remark: in Matlab, \ checks matrix structure and does the right thing:

- upper/lower triangular systems: back-substitution ($O(m^2)$).
- non-triangular linear systems: LU (then throw away the factors).
- symmetric and/or sparse systems: uses appropriate LU variants (will see in the following).

Note: `Q \ b` doesn't expand to $Q^T b$ for an orthogonal Q. Why?

Note: Python does not have an 'automagic' equivalent. Scipy fits most needs (e.g., `scipy.linalg.solve`, `scipy.linalg.solve_triangular`), but be sure to check which algorithm you are using!

Stability

Is LU factorization numerically stable? Absolutely not.
Main issue: small pivots. E.g.,

$$\begin{bmatrix} 10^{-30} & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & & \\ 10^{30} & 1 & \\ 10^{30} & & 1 \end{bmatrix} \begin{bmatrix} 10^{-30} & 1 & 1 \\ 0 & 1-10^{30} & 1-10^{30} \\ 0 & 1-10^{30} & -1-10^{30} \end{bmatrix}.$$

In floating point arithmetic, $\pm 1 - 10^{30}$ gets approximated with $10^{30} \implies$ failure (division by 0). However, that matrix is very far from singular.

In an analysis similar to the one we did for QR, one can prove that $\tilde{L}\tilde{U} = \hat{A}$ with

$$\|\hat{A} - A\| \leq \|L\|\|U\| O(\mathrm{u}),$$

however, $\|L\|\|U\|$ can get much larger than $\|A\|$, as seen above.

Pivoting

Typical fix: column (or partial) pivoting. At each step, for instance

$$\begin{bmatrix} * & * & * & * & * \\ 0 & b_{22} & * & * & * \\ 0 & b_{32} & * & * & * \\ 0 & b_{42} & * & * & * \\ 0 & b_{52} & * & * & * \end{bmatrix},$$

instead of using row 2 as a 'pivot row', swap rows so that $\max(|b_{22}|, |b_{32}|, |b_{42}|, |b_{52}|)$ occurs in the pivot position: for instance, swap row 2 and 5:

$$L_2 \begin{bmatrix} 1 & & & & \\ & & & & 1 \\ & & 1 & & \\ & & & 1 & \\ & 1 & & & \end{bmatrix} \begin{bmatrix} * & * & * & * & * \\ 0 & b_{22} & * & * & * \\ 0 & b_{32} & * & * & * \\ 0 & b_{42} & * & * & * \\ 0 & b_{52} & * & * & * \end{bmatrix} = \begin{bmatrix} * & * & * & * & * \\ 0 & b_{52} & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}.$$

Pivoting

Carrying out the same process gives

$$L_{m-1}P_{m-1}\dots L_2P_2L_1P_1A = U.$$

Another stroke of luck: we can rearrange the factor to get

$$\underbrace{P_{m-1}P_{m-2}\dots P_1}_{P}A = \underbrace{\hat{L}_1^{-1}\hat{L}_2^{-1}\dots\hat{L}_{m-1}^{-1}}_{L}U,$$

where P is a permutation matrix (orthogonal), and the $\hat{L}_k$ are obtained by applying the same row exchanges to the non-identity part of L_k.

Matlab's `lu(A)` returns `[L, U, P]` such that $PA = LU$ (or, when called with 2 outputs, `[L, U]` with L 'permuted triangular' s.t. $A = LU$).

Solving linear systems

Theorem

Any matrix $A \in \mathbb{R}^{m \times m}$ admits a factorization $A = P^T LU$, where P is a permutation matrix, L is lower triangular with ones on its diagonal, and U is upper triangular.

Systems with P, L, U can all be solved in $O(m^2)$ or less.

In practice, one can store P as a permutation of $[1, \ldots, m]$ (check `help lu`).

This is what Matlab's `x = A \ b` and `scipy.linalg.solve` do for a general, dense square A: compute PLU, solve the system, throw away the factors.

Note: overhead of partial pivoting: $O(m^2)$ instructions (though, arguably, zero floating point operations).

LU with partial pivoting — code

```
function [L, U, perm] = lu_factorization(A)
m = size(A, 1); L = eye(m); U = A; perm = 1:m;
for k = 1 : m - 1 % loop invariant: L*U == A(perm,:)
    % determine pivot position
    [val, pos] = max(abs(U(k:end, k)));
    % convert index into k:end into index in 1:end
    p = pos + k-1;
    % swap rows
    U([k,p], 1:end) = U([p,k], 1:end);
    L([k,p], 1:k-1) = L([p,k], 1:k-1);
    perm([k, p]) = perm([p, k]);
    % proceed with LU factorization step
    L(k+1:end, k) = U(k+1:end, k) / U(k, k);
    U(k+1:end, k) = 0;
    U(k+1:end, k+1:end) = U(k+1:end, k+1:end) ...
        - L(k+1:end, k) * U(k, k+1:end);
end
```

Stability of LU with partial pivoting

Pivoting ensures that $|L_{ij}| \leq 1$, hence $\|L\|$ stays small.
Is LU stable now? Still no, in the worst case.

Worst case: $\|U\|/\|A\|$ may grow as $\approx 2^m$ — see the exercises for an example.
Average case: In practice, "real world" matrices basically always have small $\|U\|/\|A\|$.

You can safely use LU + residual test rather than QR, to solve square linear systems: it saves a factor 2 on the cost.

Note: we cannot use LU to solve least-squares problems: we needed the norm-invariance properties of the orthogonal Q there.

Sparse matrices

```
>> A = sprandn(5,5, 0.6)
A =
   (2,1) -2.4372e-01
   (3,2) -1.1480e+00
   (4,2) 7.2225e-01
[...]
   (4,4) 2.5855e+00
   (2,5) -1.1658e+00
   (3,5) 1.0487e-01
>> spy(A)
```

$$\begin{bmatrix} & & * & & \\ * & & & * & * \\ & * & & & * \\ & * & & * & \\ & & * & * & \end{bmatrix}$$

LU of sparse matrices

LU / Gaussian elimination causes some fill-in:

$$\begin{bmatrix} * & * & & & * & * & & * & \\ & * & & * & & * & * & & \\ * & & * & * & * & & * & & \\ * & & & * & & & * & & \\ & * & & & & * & & * & * \\ * & & & * & & * & * & & \\ * & * & & & & & * & & * \\ * & * & * & * & & & & & \\ * & & * & & & * & & & * \end{bmatrix} \to \begin{bmatrix} * & * & & & * & * & & * & \\ & * & & * & & * & * & & \\ \color{red}{0} & \color{red}{*} & * & * & * & \color{red}{*} & * & \color{red}{*} & \\ \color{red}{0} & \color{red}{*} & & * & \color{red}{*} & \color{red}{*} & * & \color{red}{*} & \\ & * & & & & * & & * & * \\ \color{red}{0} & \color{red}{*} & & * & \color{red}{*} & * & * & \color{red}{*} & \\ \color{red}{0} & * & & & \color{red}{*} & \color{red}{*} & * & \color{red}{*} & * \\ \color{red}{0} & * & * & * & \color{red}{*} & \color{red}{*} & & \color{red}{*} & \\ \color{red}{0} & \color{red}{*} & * & & \color{red}{*} & * & & \color{red}{*} & * \end{bmatrix}$$

(and, in case you were thinking about it, QR causes even more fill-in).

Example: fill-in

```
>> A = bucky(); %sample sparse matrix
>> spy(A)
>> [L,U] =lu(bucky, 0);
>> nnz(A), nnz(L), nnz(U) %no. of nonzeros
ans =
   180
ans =
   541
ans =
   539
```

Avoiding fill-in

Fill-in depends a lot on the sparsity pattern of each specific matrix; some have more, some have less.
How can one (try to) avoid it?

Idea Choose the pivot row to be as sparse as possible at each step.
Better idea Try to predict sparsity pattern after one or several steps, instead of being ‘greedy’.

Some but’s
. . . but finding the optimal sparsity pattern is NP-complete.
. . . but we already wanted to choose the pivot row to ensure stability. We need a tradeoff between these two criteria.

Sparse LU

We will not see a sparse LU implementation here.

- The tradeoff heuristics may be complex.
- It is a tight for loop, so not something you'd want to write in (interpreted) Matlab, Python, etc. anyway.
- One needs to deal with the sparse representation, allocate memory for these lists. . .
- Another detail we have glossed over: blocking. In practice, one matrix-matrix multiplication is faster than n matrix-vector multiplications (cache reuse, vectorization. . .). So it's better to lump operations into blocks — especially for dense LU.
- And we didn't even start speaking about multi-threaded code.

Use libraries: `[L,U] = lu(A)`, `scipy.sparse.linalg.splu`.
Or directly `A b`, `scipy.sparse.linalg.spsolve`.
Remark the cost of `[L,U] = lu(A); c = L b; x = U c` (using sparse matrix storage) is $O(m \cdot (nnz(L) + nnz(U)))$.

Avoiding fill-in

. . . Sometimes, you just can't avoid it.

```
>> A = sprandn(2000, 2000, 0.005);
>> [L,U] = lu(A);
>> nnz(A) % number of nonzeros
ans =
      19955
>> nnz(L)
ans =
    1272931
>> nnz(U)
ans =
    1289056
```

Wrap-up

- LU factorization is the go-to algorithm to solve linear equations. Costs half as much as QR; stable in practice.
- It works on sparse matrices, too, with a suitable implementation, at least until fill-in becomes problematic (and that happens).
- When your L, U factors are too large for your memory, it is time to look for other algorithms (coming in the next lectures).
- In real life, use a library for this (but it's always good to have an idea of what it does). Blocking, storage format, compiled code, and other low-level optimizations give a substantial speedup.

Exercises

1. Check the 'stroke of luck' identity (for $m = 5$ is enough) by computing the matrix products.
2. Make some numerical experiments: take many $m \times m$ matrices, for a fixed m; what is (as a function of m) the maximum ratio $\|L\|\|U\|/\|A\|$ that you can obtain with unpivoted Gaussian elimination? With pivoted GE?
3. Compute (using Gaussian elimination) the U factor of the LU factorization of

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

and check that $\|U\| \geq 2^4$. Can you see how this example extends to larger dimension?

Book reference: Trefethen–Bau, Lectures 20–22.