Solving systems of equations

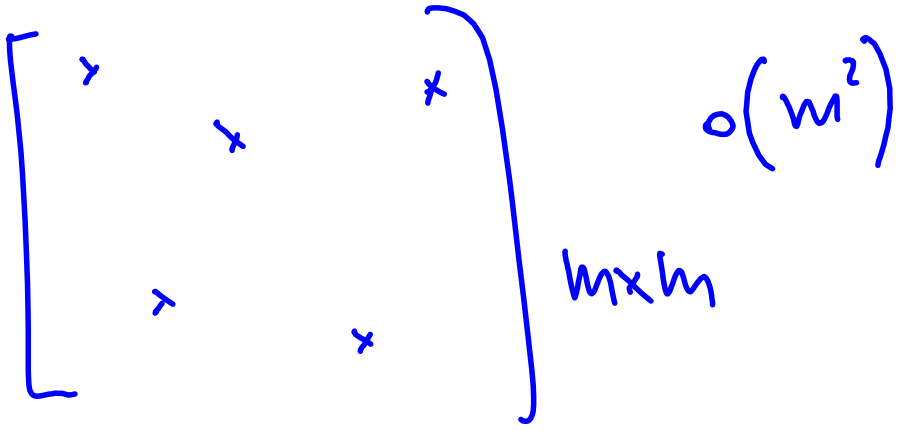
This concludes our journey through linear least squares problems.

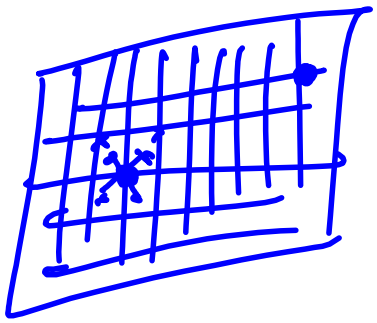
Next topic: solving systems of equations, $Ax = b$, $A \in \mathbb{R}^{m \times m}$.

. . . again? Yes, but this time we focus on large, sparse matrices.

Storing a $m \times m$ matrix with $m = 100\,000$ requires $\approx 80GB$. Applying an algorithm with complexity $O(m^3)$ to it will be prohibitive.

Luckily, many real-world matrices are sparse (loads of zeros). This includes matrices from graph/networks, KKT systems, discretizations of many 'local' phenomena. . .

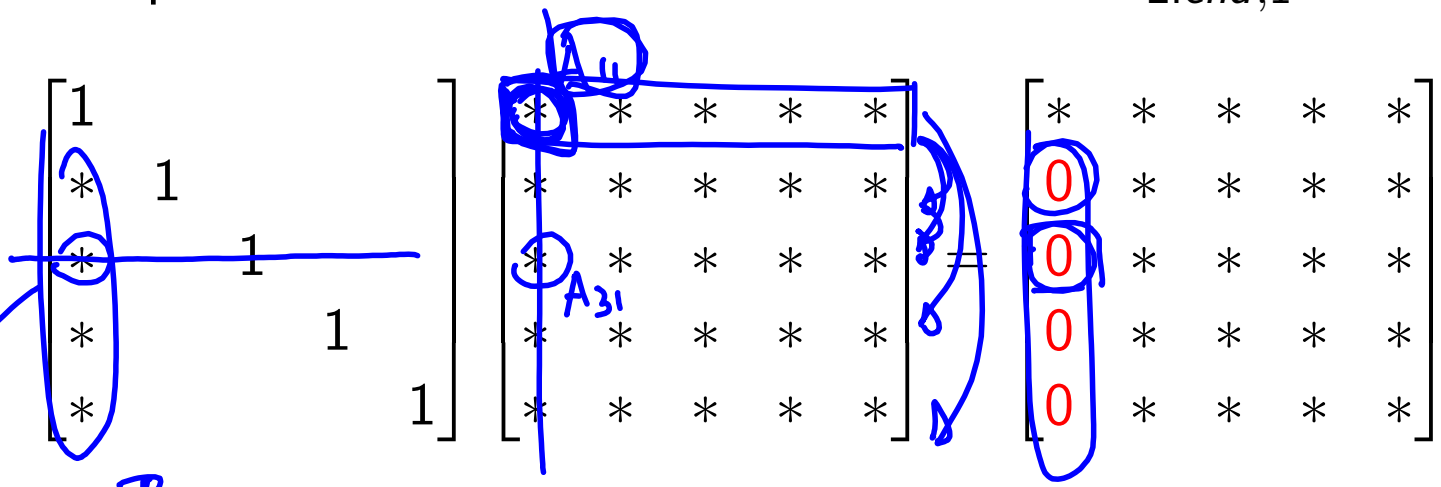




Review of LU / Gaussian elimination

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

Add multiples of row 1 to rows 2...n to kill off $A_{2:end,1}$:



$$L_1 A = A_1.$$

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

LU factorization

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

$$L_1 A_1 = A_2,$$

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

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)

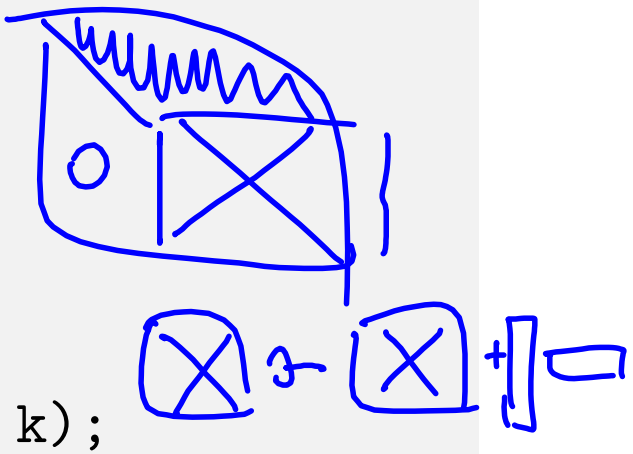
The product of the L_i's can be computed for free:

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

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

LU factorization — code

```
function [L, U] = lu_factorization(A)
m = size(A, 1);
L = eye(m);
U = A;
for k = 1 : m - 1
  % 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: $\frac{2}{3}m^3 + O(m^2)$: half as much as QR factorization.

$2(m-1)^2 + 2(m-2)^2 + \cdots + 2\cdot 2^2 + 2\cdot 1^2 \approx \frac{2}{3}m^3$

Use to solve linear systems

$Ax=b \quad x=A^{-1}b=(LU)^{-1}b=$

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

$O(n^3)$ $O(n^2)$ $O(n^2)$ back-subst.

$=U^{-1}L^{-1}b$

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

- upper/lower triangular systems: back-substitution ($O(n^2)$).
- 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, though, that `Q \ b` doesn't do the right thing for an orthogonal Q (why)?

$A \backslash b \quad O(n^3) \qquad A \backslash c \quad O(n^3)$

$a_{11}x_1 = b_1$ ⟶ get x_1

$a_{21}x_1 + a_{22}x_2 = b_2$ ⟶ x_2 $O(n^2)$

$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$ ⟶ x_3

Stability

Is LU factorization numerically stable? Absolutely not.

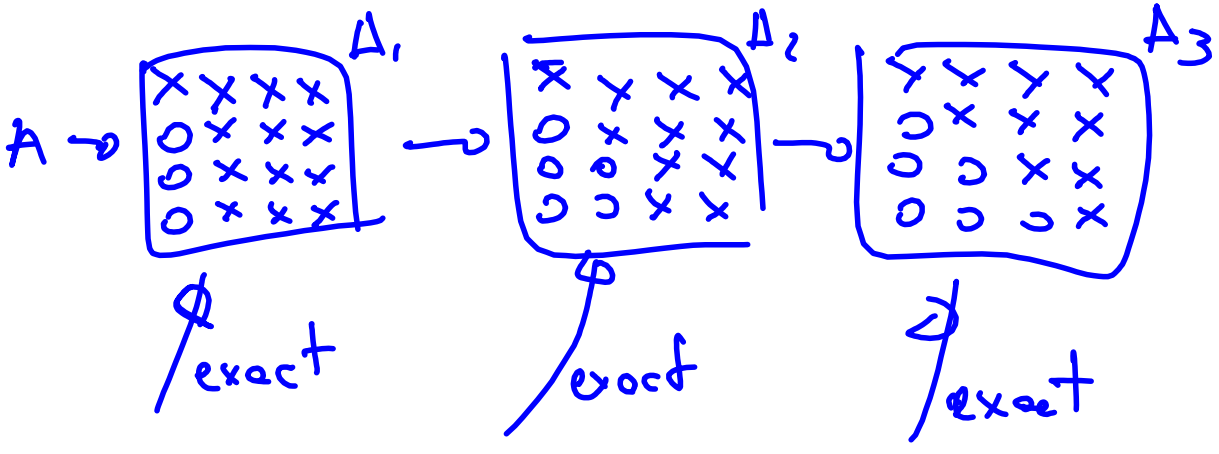
Main issue: it may produce L, U with norm much larger than $\|A\|$:

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

A_{21}/A_{11}

-10^{30} $1-10^{30}$

Errors in each step can be bounded with $\|L\|$ and/or $\|U\|$ (compare with our earlier argument for QR, where they were bounded by $\|R\| = \|A\|$ and/or $\|Q\| = 1$).



QR: $\|\Delta_i A\| = O(\epsilon_{\text{machine}})\|A_i\|$ if $\|A\|$

$A+\Delta_1 A$

$A+\Delta_1 A+\Delta_2 A$

$A+\Delta_1 A+\Delta_2 A+\Delta_3 A$

Pivoting

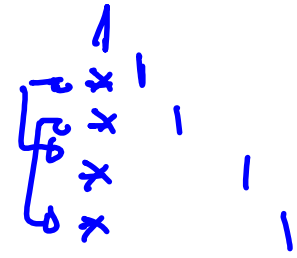
Typical fix: column (or partial) pivoting. At each step, for instance

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

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:

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

Pivoting



Carrying out the same process gives

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

Luckily, we can reorder those factors: it turns out that

$$L_{m-1}P_{m-1}\ldots L_2P_2L_1P_1 = \widehat{L}_{m-1}\widehat{L}_{m-2}\ldots\widehat{L}_1P_{m-1}P_{m-2}\ldots P_1$$

where $\widehat{L}_i$ have the same structure as the L_i.

We get

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

where P is a permutation matrix.

Matlab's `lu(A)` returns either `[L, U]` with L 'permuted triangular', or `[L, U, P]`.

Theorem

Any matrix $A \in \mathbb{R}^{m \times m}$ admits a factorization $A = PLU$, 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.

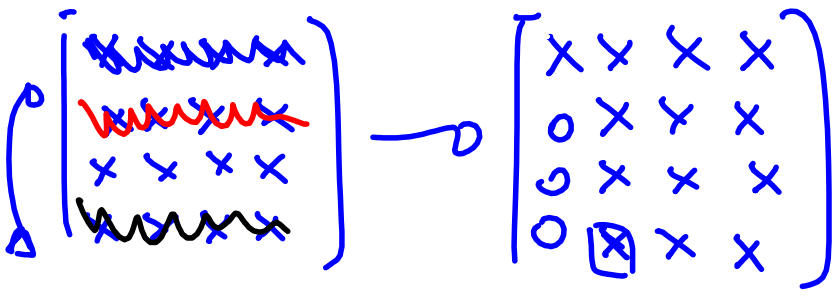
This is what Matlab's `x = A \ b` does for a general, dense square A: computes PLU, solves the system, throws away the factors.

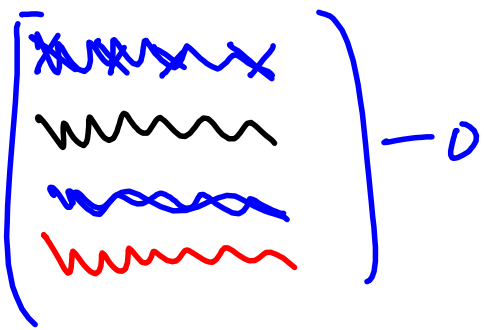
$|L_{ij}| \leq 1$

LU with partial pivoting — coding considerations

Different way to see it: if we 'magically' knew the correct permutation of the rows of A (to form a matrix $P^{-1}A = P^T A$), then LU would finish without need for pivoting.

Swapping rows of $U \leftrightarrow$ changing initial permutation.





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
  % determine pivot position
  [unused, p] = max(abs(U(k:end, k)));
  % convert index into k:end into index in 1:end
  p = k-1 + p;
  U([k,p], 1:end) = U([p,k], 1:end);
  perm([k, p]) = perm([p, k]);
  % 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
```

Stability of LU with partial pivoting

Pivoting ensures that $|L_{ij}| \leq 1$. Is LU stable now? Still not.

Worst case $\|U\|/\|A\|$ may grow as $\approx 2^m$ — see the exercises for an example.
Average case In practice, this never happens.

"Real world" matrices basically always have small $\|U\|/\|A\|$.

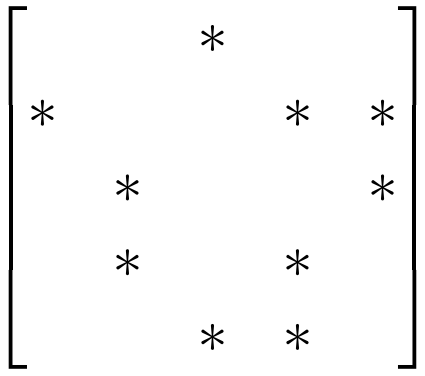
You can safely use LU + a quick check on entry size rather than QR — it saves a factor 2 on the cost.

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

Sparse matrices

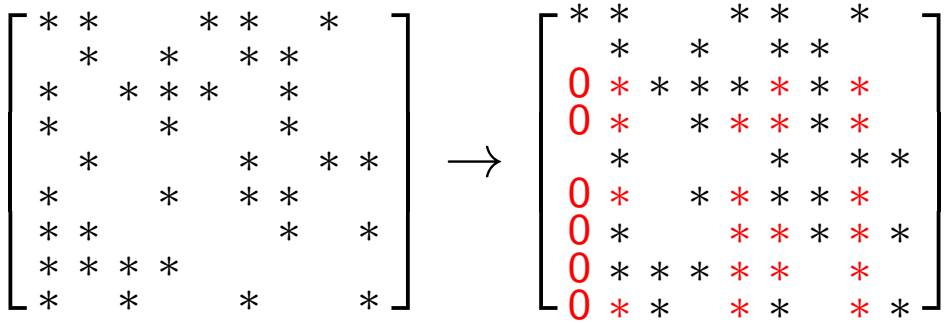
A sparse matrix is one with 'a lot of zeros'. Stored (more or less) as a list (i, j, A_{ij}):

```
>> 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)
```



LU of sparse matrices

LU / Gaussian elimination causes fill-in:



(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
```

Remark cost of `[L,U] = lu(A); c = L \ b; x = U \ c`: $O(nnz(L) + nnz(U))$.

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'.

. . . but we already wanted to choose the pivot row to ensure stability. We will need a tradeoff between these two criteria.

Sparse LU

We won't see an implementation.

- The heuristics are more complex
- It's a tight for loop, so not something you'd want to implement anyway in (interpreted) Matlab, Python, etc.
- 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.

Avoiding fill-in

. . . Sometimes, you just can't.

```
>> 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.
- On sparse matrices, too, at least until fill-in becomes catastrophic (and that happens).
- When your L, U factors are too large for your cache / RAM, it's time to look for other algorithms (coming in the next lectures).
- In real life, use a library (but it's always good to have an idea of what it does). Blocking, compiled code, and 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 of $m \times m$ 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 $\|U\| \geq 2^{m-1}$. Can you see how this example extends to larger dimension?