Computational Methods
Numerical Linear Algebra

Salvatore Filippone

School of Aerospace, Transport and Manufcturing
`salvatore.filippone@cranfield.ac.uk`

Linear Algebra

The solution of linear algebra problems lies at the heart of most models from mathematical physics:

- Systems of linear equations:

$$Ax = b$$

- Eigenvalue computations:

$$Ax = \lambda x$$

- Linear least squares:

$$\min \|Ax - b\|_2$$

How many different methods do you know?
What is the size of the largest problem you have ever solved?

Linear Systems

If you are given an $N \times N$ linear system

$$Ax = b$$

what is the best method to solve it when

$$\begin{aligned} N &= 2 \ ? \\ N &= 5 \ ? \\ N &= 20 \ ? \\ N &= 100 \ ? \\ N &= 1000 \ ? \end{aligned}$$

$N = 100,000 \ ?$

$N = 10,000,000 \ ?$

$N = 1,000,000,000 \ ?$

Problem: to appreciate what "best" means, we need to review the various methods currently in use. Note: in this lecture we *do not* have enough time to give you full details. We won't even scratch the surface. We'll give a 10000 m view of the surface. . .

Matrix Factorizations

Are there any "special" matrices for which we can solve the above problem(s) easily?

Definition (*Diagonal*)

A matrix D where $D_{ij} = 0$ for all $i \neq j$.

Definition (*lower triangular* (upper))

A matrix T where $T_{ij} = 0$ for all $i < j$ $(i > j)$.

Definition (*orthogonal*)

A matrix Q whose columns are mutually orthogonal vectors with unitary 2-norm.

Diagonal matrices

With a diagonal matrix, a linear system

$$Dx = b,$$

is actually a collection of independent equations

$$d_{ii} x_i = b_i$$

whose solutions are immediately computable by

$$x_i = \frac{b_i}{d_{ii}}, \qquad i = 1, \dots, n$$

and the computational cost is n divisions.

Orthogonal matrices

By definition the columns $Q = (q_1, q_2, \dots, q_n)$ are mutually orthogonal with unitary 2-norm; in formulae

$$\langle q_i, q_j \rangle = q_i^T \cdot q_j = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

since $\|x\|_2^2 = \langle x, x \rangle = x^T x$.
The above formula may be written in a compact way as

$$Q^T Q = I$$

or *the inverse of an orthogonal matrix is its transpose*. Hence

$$Qx = b \quad \Rightarrow \quad Q^T Q x = Q^T b \quad \Rightarrow \quad Ix = Q^T b \quad \Rightarrow \quad x = Q^T b$$

that is,

$$x_i = \sum_{j=1}^{n} q_{ji} b_i$$

with a computational cost of $2n^2$ operations.

Triangular Matrices

If a linear system $Tx = b$ has a lower triangular coefficient matrix L, it can be written as

$$\begin{cases} l_{11}x_1 & = b_1 \\ l_{21}x_1 + l_{22}x_2 & = b_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & = \cdots \\ l_{n1}x_1 + l_{n2}x_2 + \cdots + l_{nn}x_n & = b_n \end{cases}$$

The first equation can be solved by

$$x_1 = \frac{b_1}{l_{11}};$$

having found x_1, we can substitute it in the second equation, which can then be solved:

$$x_2 = \frac{1}{l_{22}}\left(b_2 - l_{21}x_1\right)$$

Triangular Matrices

In general we have:

$$x_i = \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right), \qquad i = 1, \ldots, n$$

or, in Matlab code:

```
n=size(L,1);
for i=1:n
  x(i) = b(i) - L(i,1:i-1)*x(1:i-1);
  x(i) = x(i) / L(i,i);
end
```

Solving a triangular linear system is “easy”. This is *formally equivalent* to performing

$$x = L^{-1} b.$$

Triangular linear systems

What is the operation count?

- At each loop iteration $i = 1 \dots n$, we have a dot product of size $i - 1$ at step 4
- A dot product of size k costs k multiplications and k additions;
- At each loop iteration we have a division (provisionally count all operations as equal).

Therefore:

$$\begin{aligned} \text{opcnt} = \sum_{i=1}^{n} 2(i-1)+1 &= n + 2\sum_{i=1}^{n}(i-1) \\ = n + 2\sum_{i=0}^{n-1} i &= n + 2\frac{(n-1)n}{2} \\ &= n^2 \end{aligned}$$

Matrix Factorizations

General idea: decompose A into the product of "easy" matrices

LU Factorization: $PA = LU$

P permutation, L, U lower and upper triangular;

QR Factorization : $A = QR$

Q orthogonal, R upper trapezoidal (triangular);

Schur decomposition: $AQ = QR$

Q orthogonal, R triangular;

SVD: $A = U\Sigma V^T$

U, V orthogonal, Σ diagonal;

Find a transformation that lands onto an "easy" matrix while maintaining equivalence (same solution as the original problem).

The decomposition cost is generally $O(N^3)$ and *dominant*

Linear Algebra Algorithms

Matrix Factorizations: LU

Search for a decomposition

$$PA = LU$$

where P is a permutation, L, U are lower and upper triangular. The main point is that L^{-1} and U^{-1} are easy to apply; ignoring P for the time being, we have

$$\begin{aligned} Ax = b \quad &\Leftrightarrow \quad LUx = b \\ LUx = b \quad &\Rightarrow \quad x = U^{-1}L^{-1}b \\ Ax = AU^{-1}L^{-1}b \quad &= \quad LUU^{-1}L^{-1}b \\ L(UU^{-1})L^{-1}b = LIL^{-1}b \quad &= \quad (LL^{-1})b = b \end{aligned}$$

Thus $x = U^{-1}L^{-1}b$ is the solution of the original problem.

LU Factorization

We want to factor $A = LU$ so, suppose we have already done it!

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & & \\ l_{21} & l_{22} & \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ & u_{22} & u_{23} \\ & & u_{33} \end{pmatrix}$$

Writing down the products and imposing equality:

$$\begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \end{pmatrix} = \begin{pmatrix} l_{11} \\ l_{21} \\ l_{31} \end{pmatrix} (u_{11}) \qquad \begin{pmatrix} a_{12} & a_{13} \end{pmatrix} = (l_{11}) \begin{pmatrix} u_{12} & u_{13} \end{pmatrix}$$

$$\begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{22} u_{22} & l_{22} u_{23} \\ l_{32} u_{22} & l_{32} u_{23} + l_{33} u_{33} \end{pmatrix} + \begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \begin{pmatrix} u_{12} & u_{13} \end{pmatrix}$$

n^2 equations in $n^2 + n$ unknowns; we need n additional constraints.

LU Factorization: the algorithm

- Factor the diagonal (auxiliary constraint: $l_{ii} = 1$)

$$\text{Compute} \begin{pmatrix} a_{11} \end{pmatrix} \rightarrow \begin{pmatrix} l_{11} \end{pmatrix} (u_{11})$$

- Update the first column:

$$\begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \leftarrow \begin{pmatrix} a_{21} \\ a_{31} \end{pmatrix} (u_{11})^{-1}$$

- Update the first row:

$$\begin{pmatrix} u_{12} & u_{13} \end{pmatrix} \leftarrow (l_{11})^{-1} \begin{pmatrix} a_{12} & a_{13} \end{pmatrix}$$

- Update the lower-right submatrix;

$$\begin{pmatrix} \hat{a}_{22} & \hat{a}_{23} \\ \hat{a}_{32} & \hat{a}_{33} \end{pmatrix} \leftarrow \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - \begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \begin{pmatrix} u_{12} & u_{13} \end{pmatrix}$$

- Apply recursively to lower-right submatrix;

LU Factorization

```
function [L, U]=lufact1(A)
  (nargin() == 1) ||  usage("[L,U] = lufact1(A)");

  m=size(A,1);  n=size(A,2);
  if ((m==0)||(n==0))
    return
  end

  mn=min(m,n);
  for j=1:mn
%     A(j,j+1:n) = (1.0)\A(j,j+1:n);
    A(j+1:m,j) = A(j+1:m,j)/(A(j,j));
    A(j+1:m,j+1:n) = A(j+1:m,j+1:n) - A(j+1:m,j)*A(j,j+1:n);
  end

  if (nargout() < 1)
    ans = A
  elseif (nargout() == 1)
    L=A;
  elseif (nargout() > 1)
    L=tril(A,-1)+eye(mn);
    U=triu(A);
  end

endfunction
```

LU Factorization: computational cost

Assuming $m = n$, at each loop iteration $i = 1, \ldots, n$ we have

- Step 11: commented, since it does nothing.
- Step 12: A scaling of $n - i$ items;
- Step 13: A rank 1 update, $(n-i)^2$ multiplications and $(n-i)^2$ additions.
- If i ranges from 1 to n, $n - i$ ranges from 0 to $n - 1$

Thus, the cost is

$$\text{opcnt} = \sum_{i=0}^{n-1}(2i^2 + i) = \left(\sum_{i=0}^{n-1} i\right) + 2\left(\sum_{i=0}^{n-1} i^2\right) = \frac{(n-1)n}{2} + 2\frac{(n-1)n(2n-1)}{6} = \frac{2}{3}n^3 + O(n^2)$$

As promised, the factorization phase is dominant with respect to the triangular system solve.

LU Factorization vs. Gaussian elimination

Are the two related? Let's see.
Gaussian elimination is based on the systematic application of a series of transformations that *preserve the system solution*:

1. Interchange of any two equations in the set;
2. Multiplication of all terms of an equation by a non-zero constant;
3. Substitution of an equation by its sum (or difference) with another (term by term);

Combining the second and third operation, we can add any multiple of an equation to another; for example, we can take the first equation, multiply it by $m_{21} = a_{21}/a_{11}$ and subtract from the second; but this operation *cancels* the coefficient a_{21}. Repeating for all rows, we *advance towards a triangular system*!!

LU Factorization vs. Gaussian elimination

Intuition tells us this resembles the LU factorization.
Given a matrix $A = (a_{ij})$, define the multipliers $m_{i1} = a_{i1}/a_{11}$.
Then the elimination step that annihilates the coefficients in the first column can be written as the matrix-matrix product

$$\begin{pmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{pmatrix}$$

Similarly, we have

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -m_{32} & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{pmatrix} = U$$

LU Factorization vs. Gaussian elimination

Therefore

$$M_2 M_1 A = U \Rightarrow A = M_1^{-1} M_2^{-1} U.$$

Now M_i triangular implies that M_i^{-1} is triangular as well; the product of triangular matrices is triangular, hence

$$M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1} = L.$$

It is easy to see (multiply the matrices; if an inverse exists, it is unique!):

$$M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & m_{32} & 1 \end{pmatrix}, \qquad M_1^{-1} M_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & m_{32} & 1 \end{pmatrix}.$$

Therefore

The LU factorization is a matrix formulation of the Gaussian elimination procedure.

LU Factorization

Are there any risks/pitfalls? YES!

In forming l_{11}^{-1} and u_{11}^{-1} we are implicitly assuming both exist and are different from zero.

Theorem (Wilkinson, see Golub & Van Loan))

If $A \in \mathbb{R}^{n\times n}$ *is such that* $\det(A(1:k, 1:k)) \neq 0$ *for all* $k = 1 \dots n-1$, *then* A *has an* LU *factorization*

However, this is too restrictive and does not cover:

$$A = \begin{pmatrix} 0 & 1.5 \\ 2 & 0 \end{pmatrix}$$

$Ax = b$ can be solved whenever $\det(A)$ is nonzero (formula for the inverse based on cofactors).

LU Factorization

Our *LU* algorithm does not cover all “reasonable” inputs.

Alternatives:

1. Figure out classes of matrices for which *LU* works “as is” (i.e. restrict the input, and give a different, fancy name to the special version of the algorithm)
2. Modify the *LU* procedure to cope with those matrices for which it fails (i.e. extend the algorithm)

LDM^T Factorization

Under the same conditions as LU factorization, it is possible to extract

$$D = \operatorname{diag}(u_{11}, \ldots, u_{nn})$$

to obtain both L and M unit diagonal

$$A = LDM^T, \quad \text{where } M^T = D^{-1}U.$$

Now consider A symmetric positive definite, i.e.

$$x^T A x > 0 \qquad \text{for all nonzero } x \in \mathbb{R}^n.$$

Easy to prove:

- Any principal submatrix of A is positive definite;
- All eigenvalues are positive real numbers;
- All principal minors are positive;

It follows that $A = LDL^T$, moreover D is strictly positive.

Cholesky Factorization

Theorem (Cholesky decomposition)

When A is symmetric positive definite we have

$$A = LL^T = U^T U.$$

Proof.

Since $A = LDL^T$ with positive D, we can form

$$\sqrt{D} = \mathrm{diag}(\sqrt{d_{ii}}),$$

hence

$$A = L\sqrt{D}\sqrt{D}^T L^T = \bar{L}\bar{L}^T.$$

$\square$

Cholesky Factorization, version 1

- Factor the diagonal (auxiliary constraint: $l_{ii} = u_{ii}$)

$$\text{Compute}\,(a_{11}) \rightarrow (l_{11})\,(l_{11})$$

- Update the first column:

$$\begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \leftarrow \begin{pmatrix} a_{21} \\ a_{31} \end{pmatrix} (l_{11})^{-1}$$

- Update the first row:

$$\begin{pmatrix} l_{21} & l_{31} \end{pmatrix} \leftarrow (l_{11})^{-1} \begin{pmatrix} a_{21} & a_{31} \end{pmatrix}$$

- Update the lower-right submatrix;

$$\begin{pmatrix} \hat{a}_{22} & \hat{a}_{32} \\ \hat{a}_{32} & \hat{a}_{33} \end{pmatrix} \leftarrow \begin{pmatrix} a_{22} & a_{32} \\ a_{32} & a_{33} \end{pmatrix} - \begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \begin{pmatrix} l_{21} & l_{31} \end{pmatrix}$$

- Apply recursively to lower-right submatrix;

Cholesky Factorization, version 2

Since the matrix is symmetric, half the operations are duplicate; work only on L (or U):

- Factor the diagonal
$$l_{11} \leftarrow \sqrt{(a_{11})}$$
- Update the first column:
$$\begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \leftarrow \begin{pmatrix} a_{21} \\ a_{31} \end{pmatrix} (l_{11})^{-1}$$
- Update lower portion of the lower-right submatrix;
$$\begin{pmatrix} \hat{a}_{22} & - \\ \hat{a}_{32} & \hat{a}_{33} \end{pmatrix} \leftarrow \begin{pmatrix} a_{22} & - \\ a_{32} & a_{33} \end{pmatrix} - \begin{pmatrix} l_{21} \\ l_{31} \end{pmatrix} \begin{pmatrix} l_{21} & l_{31} \end{pmatrix}$$
- Apply recursively to lower-right submatrix;

Cholesky Factorization

```
function [L]=chol1(A)
  (nargin() == 1) ||  usage("[L] = chol1(A)");

  m=size(A,1);  n=size(A,2);
  ((m>=0)&&(m==n)) || error("Invalid input: A must be square");
  if (n==0)
    return
  end

  for j=1:n
    (A(j,j) > 0)  || error("Invalid input: A is not SPD");

    A(j,j) = sqrt(A(j,j));
    A(j+1:n,j) = A(j+1:n,j)/(A(j,j));
    A(j+1:n,j+1:n) = A(j+1:n,j+1:n) - tril(A(j+1:n,j)*A(j+1:n,j)');
  end

  if (nargout() < 1)
    tril(A)
  elseif (nargout() >= 1)
    L=(tril(A));
  end

endfunction
```

Thomas Algorithm (TDMA)

Another simplified form of Gaussian elimination, used to solve *tridiagonal and diagonally-dominant* systems of equations.

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

where $a_1 = 0$ and $c_n = 0$.

In matrix format

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & \vdots \\ 0 & a_3 & b_3 & c_3 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & a_{n-2} & b_{n-2} & c_{n-2} & 0 \\ 0 & 0 & \dots & a_{n-2} & b_{n-1} & c_{n-1} \\ 0 & 0 & \dots & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

Thomas Algorithm (TDMA) (cont)

Forward Elimination phase:

```
for $k = 2, \ldots, n$    do
    $m = \frac{a_k}{b_{k-1}}$
    $b_k = b_k - mc_{k-1}$
    $d_k = d_k - md_{k-1}$
```

Backward Substitution phase:

$$x_n = \frac{d_n}{b_n}$$

```
for $k = n-1, \ldots, 1$    do
    $x_k = \frac{d_k - c_k x_{k+1}}{b_k}$
```

Cost is $O(n)$. Compare with $O(n^3)$!

LU Factorization: Pivoting

For non positive definite matrices, overcome the null pivot problem:

> *When an element on the diagonal is zero, search for a nonzero in its column, and swap the relevant rows*

This is equivalent to applying P

$$PA = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1.5 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 1.5 \end{pmatrix}$$

Thus we are computing $PA = LU$ to get at the solution

$$Ax = b \Rightarrow x = U^{-1}L^{-1}Pb$$

Extension: *Always* search for the coefficient with largest absolute value! This improves the numerical accuracy

LU Factorization: Pivoting

```
function [L, U, P]=lupfact1(A)
 ((nargin() ==1)&&(nargout==3)) ||  usage("[L,U,P] = lupfact1(A)");

  m=size(A,1);  n=size(A,2);   mn=min(m,n); lp=eye(m);
  if ((m==0)||(n==0))
    return
  end

  for j=1:mn
    [mx,ix] = max(abs(A(j:m,j)));  ix=ix+j-1;
    tmp(1:n)  = A(j,1:n);   A(j,1:n)  = A(ix,1:n);   A(ix,1:n) = tmp(1:n);
    tmp(1:m)  = lp(j,1:m); lp(j,1:m)  = lp(ix,1:m); lp(ix,1:m) = tmp(1:m);
%     A(j,j+1:n) = (1.0)\A(j,j+1:n);
    A(j+1:m,j) = A(j+1:m,j)/(A(j,j));
    A(j+1:m,j+1:n) = A(j+1:m,j+1:n) - A(j+1:m,j)*A(j,j+1:n);
  end

  L=tril(A,-1); L(1:mn,1:mn)=L(1:mn,1:mn)+eye(mn);
  U=triu(A);
  P=lp;
endfunction
```

Advantages of using LU decomposition

The Gauss elimination process can now be split into two stages:

- LU decomposition, with computational cost $\approx (2/3)n^3$
- Calculation of the actual solution using resulting triangular matrices, with computational cost $\approx 2n^2$

Suppose we need to solve p systems with the identical matrix A but different right-hand sides $\boldsymbol{b}_{(1)}, \boldsymbol{b}_{(2)}, \ldots \boldsymbol{b}_{(p)}$.

Then, we carry out the first stage (LU decomposition) only once, and apply the second stage for each $\boldsymbol{b}_{(k)}$, $k = 1, 2, \ldots p$.

The total computational cost is

$$(2/3)n^3 + 2pn^2$$

rather than

$$p\left[(2/3)n^3 + 2n^2\right]$$

Conditioning of *LU* factorization

What makes a linear system $Ax = b$ *hard to solve*?

Theorem

The LU decomposition (with pivoting) exists if $\det(A) \neq 0$

Does it follow that $\det(A)$ is a measure of the quality of the solution, or closeness to singularity?

Conditioning of *LU* factorization

What makes a linear system $Ax = b$ *hard to solve*?

Theorem

The LU decomposition (with pivoting) exists if $\det(A) \neq 0$

Does it follow that $\det(A)$ is a measure of the quality of the solution, or closeness to singularity?

NO!

Consider the following case:

$$A = \begin{pmatrix} \alpha & 0 & \dots \\ 0 & \alpha & 0 \\ \dots & 0 & \alpha \end{pmatrix}$$

We have

$$\det\left(\frac{1}{2}A\right) = 2^{-n}\det(A)$$

Despite the infinitesimal value of $\det(A/2)$, the result of $\frac{1}{2}Ax = b$ can be computed with as much precision as we can reasonably expect (i.e. just one rounding per element of x)!

Conditioning of LU factorization

Back to the drawing board:

Given A, which perturbation δA makes $(A + \delta A)$ singular?

Conditioning of LU factorization

Back to the drawing board:

Given A, which perturbation δA makes $(A + \delta A)$ singular?

If $(A + \delta A)$ is singular, it follows that there exists x such that:

$$(A + \delta A)x = 0$$

Hence

$$\begin{aligned}
\delta A x &= -Ax \\
A^{-1}\delta A x &= -x \\
\|A^{-1}\delta A x\| &= \|x\| \\
\|A^{-1}\|\|\delta A\|\|x\| &\geq \|x\| \\
\|\delta A\| &\geq \frac{1}{\|A^{-1}\|} \\
\frac{\|\delta A\|}{\|A\|} &\geq \frac{1}{\|A\|\|A^{-1}\|} = \frac{1}{\kappa(A)}
\end{aligned}$$

Conditioning of LU factorization

The quantity

$$\kappa(A) = \|A\|\|A^{-1}\|$$

is the (normwise) *condition number* of A, *the smaller, the better!*
Properties:

- $$1 \le \kappa(A) \le \infty;$$
- If Q is orthogonal, then

$$\kappa_2(Q) = 1$$

Hence, orthogonal transformation are “good”.

Matrix Factorizations: QR

$$A = QR$$

where

- $A \in \mathbb{R}^{m \times n}$ is (in general) rectangular;
- $Q \in \mathbb{R}^{m \times m}$ is orthogonal,
- $R \in \mathbb{R}^{m \times n}$ upper trapezoidal (triangular).

what is it useful for?

Matrix Factorizations: QR

$$A = QR$$

where

- $A \in \mathbb{R}^{m\times n}$ is (in general) rectangular;
- $Q \in \mathbb{R}^{m\times m}$ is orthogonal,
- $R \in \mathbb{R}^{m\times n}$ upper trapezoidal (triangular).

what is it useful for?

1. Alternative way to solve $Ax = b$;
2. Solution of least square problems

$$\min \|Ax - b\|_2;$$

3. Building block in eigenvalue computations.

Matrix Factorizations: QR

Let's see, given the problem

$$\min \|Ax - b\|_2$$

we have

$$\begin{aligned}
\|Q^T(Ax-b)\|_2^2 &= (Q^T(Ax-b))^T(Q^T(Ax-b)) = \\
(Ax-b)^TQQ^T(Ax-b) &= (Ax-b)^T(QQ^T)(Ax-b) = \\
(Ax-b)^TI(Ax-b) &= \|(Ax-b)\|_2^2
\end{aligned}$$

given that Q is orthogonal; now, $Q^TA = Q^TQR$ therefore

$$\min \|Ax-b\|_2 = \min \|Q^T(Ax-b)\|_2 = \min \|Rx - Q^Tb\|_2$$

and this is easy to solve since R is trapezoidal (triangular).
Note that the 2-norm of vectors is invariant under Q:

$$\|Qx\|_2^2 = (Qx)^T(Qx) = x^TQ^TQx = x^TIx = x^Tx = \|x\|_2^2$$

Matrix Factorizations: QR

To compute the QR factorization we need to find transformations P_i such that

1. P is orthogonal, i.e. $P^T P = I$;
2. The (repeated) application to A brings us to a triangular matrix
$$P_n P_{n-1} \cdots P_2 P_1 A = R$$

because then we have

$$A = QR \quad Q = P_1^T P_2^T \cdots P_{n-1}^T P_n^T .$$

Two such tools exist:

1. Householder reflections;
2. Givens rotations.

Householder reflections

Definition (Householder reflection)

(or elementary Householder matrix) is the matrix

$$H = I - 2ww^T, \qquad w \in \mathbb{R}^n, \quad \|w\|_2 = 1,$$

The matrix is

- *Symmetric*

$$(ww^T)^T = (w^T)^T w^T = ww^T, \qquad w^T w = 1$$

- *Orthogonal*

$$\begin{aligned} HH &= (I - 2ww^T)(I - 2ww^T) = \\ &= II - 2Iww^T - 2ww^T I + 4ww^T ww^T \\ &= I - 4ww^T + 4w(w^T w)w^T \\ &= I - 4ww^T + 4ww^T = I. \end{aligned}$$

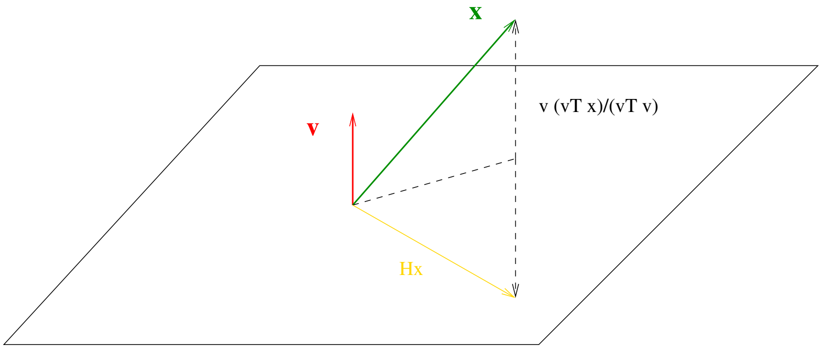
Householder reflections

From a geometric point of view, computing $y = Hx$ is the same as applying a *reflection* with respect to the hyperplane orthogonal to vector w; note that matrix $P = I - ww^T$ applies a projection onto the same hyperplane.

Starting from a non-unitary vector v we can update the definition as

$$H = I - 2\frac{vv^T}{v^Tv}.$$

Householder reflections



Householder reflections

All previous properties are important, but the practical usefulness of Householder reflections shines when we ask the question:

> *Is it possible to find w such that the application of H introduces zeros?*

Let's be a bit more specific: let's suppose we want to zero all entries in a vector after the first, i.e.

$$Hx = ke_1,$$

where e_1 is the first vector in the canonical basis for $\mathbb{R}^n$.
Since H is orthogonal we must have

$$|k| = \|ke_1\| = \|Hx\| = \|x\|;$$

therefore

$$k = \pm\sigma, \qquad \sigma = \|x\|.$$

Householder reflections

Expanding the product we have

$$ke_1 = Hx = x - 2ww^Tx = x - 2(w^Tx)w$$

hence

$$w = \frac{x - ke_1}{2w^Tx} = \frac{x - ke_1}{\|x - ke_1\|},$$

where the last equality must be true since we are searching for a w with unit norm.
Since the first component of x is $x_1 = x^Te_1$, we have

$$x^THx = x^Tx - 2(w^Tx)^2 = kx_1 \Rightarrow (w^Tx)^2 = \frac{\sigma^2 - kx_1}{2},$$

thus

$$\|x - ke_1\| = 2w^Tx = \sqrt{2(\sigma^2 - kx_1)},$$

which allows us to properly normalize w.

Householder reflections

We are free to choose the sign of σ, it's best to make sure that $-kx_1 > 0$, or

$$k = -sign(x_1)\sigma, \qquad \|x - ke_1\| = \sqrt{2\sigma(\sigma + |x_1|)}.$$

A Matlab code implementing this derivation:

```
n=length(x);
e1=eye(n,1);
sigma=norm(x);
k=-sign(x(1))*sigma;
lambda=sqrt(2*sigma*(sigma+abs(x(1))));
w=(x-k.*e1)./lambda;
H=eye(n)-2*w*w';
```

Householder reflections

The previous code is correct but it is usually modified to avoid the square root and the normalization:

$$2ww^T = \frac{(x - ke_1)(x - ke_1)^T}{\sigma(\sigma + |x_1|)} = \frac{1}{\beta} vv^T$$

Moreover, it is not necessary to build explicitly H; when we apply to vector z we have

$$Hz = (I - \frac{1}{\beta} vv^T)z = z - \delta v, \quad \delta = \frac{1}{\beta} v^T z,$$

therefore matrix H can be effectively stored using just one vector v and a scalar β.

Example

Let $\boldsymbol{x} = (1, 7, 2, 3, -1)^T$. Find P.

- $||\boldsymbol{x}||_2 = 8$, $\boldsymbol{v} = (9, 7, 2, 3, -1)^T$
- the reflection matrix is

$$
\begin{aligned}
P &= I - 2\frac{vv^T}{v^Tv} = I - \frac{1}{72}\begin{pmatrix} 81 & 63 & 18 & 27 & -9 \\ 63 & 49 & 14 & 21 & -7 \\ 18 & 14 & 4 & 6 & -2 \\ 27 & 21 & 6 & 9 & -3 \\ -9 & -7 & -2 & -3 & 1 \end{pmatrix} \\
&= \begin{pmatrix} -0.1250 & -0.8750 & -0.2500 & -0.3750 & 0.1250 \\ -0.8750 & 0.3194 & -0.1944 & -0.2917 & 0.0972 \\ -0.2500 & -0.1944 & 0.9444 & -0.0833 & 0.0278 \\ -0.3750 & -0.2916 & -0.0833 & 0.8750 & 0.0417 \\ 0.1250 & 0.0972 & 0.0278 & 0.0417 & 0.9861 \end{pmatrix}
\end{aligned}
$$

- the result is $P\boldsymbol{x} = (-8, 0, 0, 0, 0)^T$

Householder QR

Consider the first step of QR factorization: we need to zero the elements of the first column of A below th main diagonal with an orthogonal transformation, obtaining

$$A^{(2)} = H_1 A^{(1)}, \qquad A^{(2)} = \begin{pmatrix} k_1 & a_{12}^{(2)} \\ 0 & A_{22}^{(2)} \end{pmatrix},$$

where $A_{22}^{(2)}$ is the submatrix $A(2:m, 2:n)$. At this point we can build the matrix $\hat{H}_2$ that zeroes the first column of $A_{22}^{(2)}$; if we then *border* it with the first row/column of the identity we obtain

$$H_2 = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & \hat{H}_2 & \\ 0 & & & \end{pmatrix}, \qquad A^{(3)} = H_2 A^{(2)} = \begin{pmatrix} k_1 & v_1^T \\ 0 & \hat{H}_2 A_{22}^{(2)} \end{pmatrix}$$

Householder QR

Expanding $A^{(3)}$

$$A^{(3)} = H_2 A^{(2)} = \begin{pmatrix} k_1 & * & \dots & \dots & * \\ 0 & k_2 & * & \dots & * \\ 0 & 0 & & & \\ \vdots & \vdots & & \hat{A}^{(3)} & \\ 0 & 0 & & & \end{pmatrix},$$

and, in general, at each step i we need zero the first column of a submatrix of size $(n-i+1) \times (n-i+1)$ with a Householder transformation $\hat{H}_i$ which is then bordered o obtain

$$H_i = \begin{pmatrix} I_{i-1} & 0 \\ 0 & \hat{H}_i \end{pmatrix}$$

Householder QR

Finally

$$R = A^{(n)} = H_{n-1}A^{(n-1)} = H_{n-1}H_{n-2}\cdots H_2H_1A = Q^TA$$

where Q^T is orthogonal, being the productf orthogonal matrices.
We have then computed

$$A = QR.$$

This is the algorithm used by the Matlab `qr()` function; it costs $4/3n^3$ arithmetic operations.
When $m > n$ it is easy to verify that with n reflections the algorithm produces a trapezoidal matrix

$$R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$$

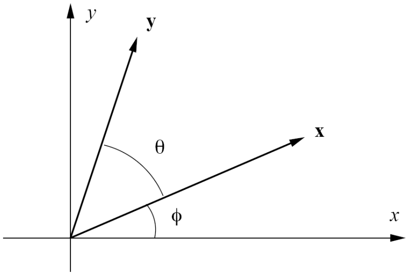
where R_1 is upper triangular, at a cost of $2n^2(m - n/3)$ operations.

Givens rotations

In $\mathbb{R}^2$ a rotation by an angle θ is represented by the orthogonal matrix

$$G(\theta) = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

meaning that $y = Qx$ is a vector with the same length as x rotated by θ radians.



Givens rotations

In $\mathbb{R}^n$ an *elementary Givens matrix* applies a rotation in the plane identified by vectors e_i and e_j:

$$G_{ij} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & \dots & s & & \\ & & \vdots & & \vdots & & \\ & & -s & \dots & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

where $c^2 + s^2 = 1$.

Givens rotations

Can we zero entries in a vector with a Givens rotation?

$$y = G_{ij}x \Rightarrow y_k = \begin{cases} cx_i + sx_j, & k = i \\ -sx_i + cx_j, & k = j \\ x_k & \text{otherwise} \end{cases}$$

We have two parameters c and s linked by one constraint; we can therefore impose one additional constraint

$$y_j = 0;$$

to achieve this we need $c = sx_i/x_j$,hence

$$\left(\frac{x_i^2}{x_j^2} + 1\right) s^2 = 1 \Rightarrow s^2 = \frac{x_j^2}{x_i^2 + x_j^2}, \quad c^2 = \frac{x_i^2}{x_i^2 + x_j^2}$$

Givens rotations

A numerically stable implementation is:

```
function [c,s] = givrot(xi,xj)
  if (xj == 0)
    c=1; s=0;
    return
  elseif (abs(xj)>abs(xi))
    t=xi/xj;
    z=sqrt(1+t^2);
    s=1/z; c=ts;
  else
    t=xj/xi;
    z=sqrt(1+t^2);
    c=1/z; s=tc;
  end
end
```

Givens QR

It is therefore possible to reduce a matrix to triangular form with a sequence of rotations, each zeroing one entry:

$$G_{n-1,n}G_{n-2,n}G_{n-2,n-1}\cdots G_{2,4}G_{2,3}G_{1,n}\cdots G_{1,3}G_{1,2}A = R$$

We arrive at a result equivalent to the Householder QR, but the computational cost is $3n^2(m - n/3)$ operations, higher than the cost of the Householder version. The main difference is that Givens rotations can be used to zero entries in a very selective way:

If matrix A already contains zeroes below the main diagonal, we can skip the corresponding transformation(s), thereby reducing the computational cost

With Householder reflections we cannot save arithmetic by exploiting pre-existing zeros. Givens rotations are the method of choice when matrix A is already “close” to triangular form R.

Linear Algebra: Iterative Solvers

Iterative Solvers

Solve $Ax = b$ building a series $x_1, x_2, \dots, x_k$ converging to the solution $\hat{x}$.
Ingredients:

- Initial guess x_0;
- Advancement rule $x_{i+1} = f(x_0, x_1, \dots, x_i)$;
- Stopping criterion.

Final goal: solve an $N \times N$ system in less than $O(N^3)$ operations (even better, in less than $O(N^2)$ operations).
If sequence stops at $j < N$ and each iteration costs $O(N^2)$ then we have a total cost

$$jN^2 < N^3$$

Stopping criteria for iterative solvers

Criteria usually based on the *residual*

$$r = b - Ax,$$

or, a measure of how closely our solution reproduces the data. Most common criteria:

- $$\|r\| \leq \epsilon;$$
- $$\|r\| \leq \epsilon\|b\|;$$
- $$\|r\| \leq \epsilon(\|A\|\|x\| + \|b\|)$$

The second is probably the most commonly used, but it can be misleading for ill-conditioned systems.

Iterative solvers

First idea: split the matrix

$$A = M + (A - M)$$

Hence

$$Mx = (M - A)x + b$$

which can be transformed into:

$$\begin{aligned} Mx_{k+1} &= (M - A)x_k + b \Rightarrow \\ x_{k+1} &= M^{-1}(M - A)x_k + M^{-1}b \end{aligned}$$

Iterative solvers

Defining:

$$B = M^{-1}(M - A)$$

we have

$$x_{k+1} = Bx_k + M^{-1}b$$

Note that we also have

$$e_{k+1} = Be_k = B^k e_0$$

Iterative solvers

We hope to achieve

$$\lim_{k\to\infty} \|e_k\| = 0$$

But we have

$$\|e_k\| = \|B^k e_0\| \le \|B^k\| \|e_0\| \le \|B\|^k \|e_0\|$$

Therefore: convergence condition:

$$\rho(B) < 1$$

- Sufficient condition (in theory);
- But in practice it may not enough!

In fact, *non-normal* matrices may exhibit *large humps* in the convergence history.

Iterative solvers

Total iteration time:

$$T_{\text{tot}} = T_{\text{setup}} + N_{\text{it}} \times T_{\text{it}}$$

Where do the various terms come from? Let's look again:

$$B = M^{-1}(M - A)$$

and we want $\rho(B)$ small.
Therefore we need:

- M easy (fast) to determine; T_{setup}
- M^{-1} easily applied; T_{it}
- $M \approx A \Rightarrow B \approx 0$; N_{it}

These requisites are contradictory! So, we need to find good tradeoffs.

Jacobi method

$$M = diag(A)$$

```
m  = diag(diag(a));
mi = m;
mi(mi ~= 0) = 1.0 ./ mi(mi ~= 0) ;
n= (m-a);
bn2=norm(b);
for i=1:itmax
    x=mi*(n*x+b);
    rn2 = norm(b-a*x);
    if (rn2 < eps*rb)
        break;
    end
end
```

Gauss-Seidel method

$$A = L + D + U \Rightarrow M = (L + D)$$

```
m =  tril(a);
n = -triu(a,1);
bn2=norm(b);
for i=1:itmax
    x=m\(n*x+b);
    rn2 = norm(b-a*x);
    if (rn2 < eps*rb)
        break;
    end
end
```

Iterative solvers

Iterative solvers:

- Are difficult to tune;
- Are not universally robust;
- Need good preconditioners (defined later);

Then, why bother???

Final goal: solve an $n \times n$ system in (much) less than $O(n^2)$ operations!!

Sparse Matrices

In most applications A is sparse:

Definition

A matrix $A \in \mathbb{R}^{n \times n}$ is sparse if $nnz(A) \ll O(n^2)$

In practice:

> *A matrix is sparse when the percentage of its entries that are zero is sufficiently large that it pays off to devise a scheme to avoid their explicit memorization in the computer (Wilkinson)*

When discretizing a differential equation, most coefficients are “structurally” zero because differential operators are local ! (Note: there may exist coefficients that are only “numerically” zero). Nonzero coefficients are $nz = k \cdot n$ with k depending on the discretization technique and local topology, *not* on n.

Sparse Matrices: finite differences

Approximate derivatives (Taylor expansion) for a discrete set of points $x = nh,\ n \in \mathbb{N}$:

$$\frac{du}{dx}(x) = \frac{u(x+h) - u(x)}{h} - \frac{h}{2}\frac{d^2u(x)}{dx^2} + O(h^2)$$

Using the same formula with $-h$, summing and applying the mean value theorem we derive

$$\frac{d^2u(x)}{dx^2} = \frac{u(x+h) + u(x-h) - 2u(x)}{h^2} - \frac{h^2}{12}\frac{d^4u(\xi)}{dx^4}.$$

If we now do the same in both x and y directions we get

$$\Delta u(x) \approx \frac{1}{h^2} \quad [\quad u(x_1+h, x_2) + u(x_1-h, x_2) \\ +u(x_1, x_2+h) + u(x_1, x_2-h) - 4u(x_1, x_2)],$$

a 5-point approximation stencil (i.e.: $k = 5$).

Sparse Matrices: solution methods

First idea: let's use factorizations:

"Direct methods for sparse matrices".

We have a major problem here:

The LU (or QR) factors of a sparse matrix ARE NO LONGER sparse, but dense.

Equally, the inverse A^{-1} is dense (in general)
Therefore the storage requirements grow explosively. Ex: Poisson Equation

$$x_{0,1} + x_{1,0} + x_{0,-1} + x_{-1,0} - 4x_{0,0} = b_{0,0}$$

a square domain with 8 grid points on each side

Sparse Matrices

Iterative solvers to the rescue. Why?
The sparse matrix A is only ever used to perform matrix-vector products

$$y = Ax,$$

hence *its structure is not altered*.

Sparse Matrix-Vector product: only perform the arithmetic for the nonzeros. Hence the cost of an SpMV operation is:

$$2 \cdot nnz = 2 \cdot kn = O(n)$$

Therefore the total cost of an iterative method that converges in j iterations will be

$$O(jkn) \ll O(n^3),$$

and often

$$jkn < n^2.$$

Iterative solvers: Krylov methods

Today we (almost) always use *Krylov Subspace Projection Methods*:

CG	: Conjugate Gradients, for SPD matrices;
GMRES	: Generalized Minimum Residual, for nonsymmetric matrices;
BiCGSTAB	: BiConjugate Gradients Stabilized, for nonsymmetric systems;

CGS, BiCG, TFQMR,. . .

Their derivation and a detailed analysis of their properties would bring us too far, however:

- They are *orders of magnitude* more efficient than stationary iterations;
- The rate of convergence depends on the spectrum of the matrices, in sometimes very complicated ways;

Iterative solvers: preconditioning

Apply a transformation to the problem

$$Ax = b \Leftrightarrow M^{-1}Ax = M^{-1}b$$

which preserves the solution; however $M^{-1}A$ has a *different spectrum*

We want:

1. M is easy to compute;
2. M^{-1} is easy to apply;
3. $M^{-1}A \approx I$;

These three requisites are contradictory!

But a good preconditioner guarantees convergence in $j \ll n$ steps.

E.g.: BiCGSTAB with Algebraic Multigrid, on a system from a thermal diffusion *PDE* of size 3×10^6 can converge in as few as 12 iterations!!!