Chapter 2: **Linear** system

Given n linear equations with n variables:

$$\begin{aligned}
a_{11}\,x_1 + a_{12}\,x_2 + \cdots + a_{1n}\,x_n &= b_1 \\
a_{21}\,x_1 + a_{22}\,x_2 + \cdots + a_{2n}\,x_n &= b_2 \\
\vdots \quad & \quad \vdots \\
a_{n1}\,x_1 + a_{n2}\,x_2 + \cdots + a_{nn}\,x_n &= b_n
\end{aligned}$$

Matrix-form: solve $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$ with

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Agenda

If A is invertible, then it is easy to solve $Ax = b$ in Python with

```
x = np.linalg.solve(A,b)
```

but we would like to:

- explain what is really happening,
- investigate computational complexity, and
- study the sensitivity of the solution to errors in the data.

Concerning efficiency (not in the book)

Based on"Cramer's rule", the ith element in x:

$$x_i = \frac{\det(\boldsymbol{A}^{[i]})}{\det(\boldsymbol{A})}\ , \qquad i = 1, 2, \ldots, n$$

where

$$\boldsymbol{A}^{[i]} \equiv \Big[\, A(:, 1 : i{-}1)\, ,\ b\, ,\ A(:, i{+}1 : n)\,\Big].$$

The computational complexity of calculating a determinant

$$\approx n! \text{ multiplications}$$

Gaussian elimination requires

$$\approx n^3 \text{ multiplications}$$

Example: $n = 10 \Rightarrow (n+1)n! = 39\,916\,800$ and $n^3 = 1\,000$.

Back substitution (page 91–92)

The matrix $\boldsymbol{A}$ is *upper triangular* if

$$a_{ij} = 0, \qquad i > j.$$

Example: The following upper triangular system

$$\begin{bmatrix} 5 & -5 & 10 \\ 0 & 2 & 4 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -25 \\ 16 \\ -2 \end{bmatrix}.$$

can be simply solved by:

$$\begin{aligned} x_3 &= (-2)/(-1) = 2 \\ x_2 &= (16 - 4\,x_3)/2 = (16 - 4 \cdot 2)/2 = 8/2 = 4 \\ x_1 &= (-25 - (-5)\,x_2 - 10\,x_3)/5 = (-25 + 5 \cdot 4 - 10 \cdot 2)/5 = -5. \end{aligned}$$

Back substitution (page 92)

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

<u>Algorithm:</u>

$x_n = b_n/a_{nn}$
for $i = n-1 : -1 : 1$
    sum $= b_i$
    for $j = i+1 : n$
        sum $=$ sum $- a_{ij} \cdot x_j$
    end
    $x_i =$ sum$/a_{ii}$
end

Computational complexity of back substitution (page 93)

A "LOp" = "Long Operation" = a multiplication or a division.

The number of LOps in back substitution is:

$$
\begin{aligned}
\text{LOps} &= 1 + 2 + \cdots + (n-1) + n \\
&= {}^1\!/\!_2\, n\,(n+1) \\
&= {}^1\!/\!_2\, n^2 + {}^1\!/\!_2\, n \;\approx\; {}^1\!/\!_2\, n^2 \,.
\end{aligned}
$$

The computational complexity is dominated by the term of n with the highest power. So the dominant computational complexity is total ${}^1\!/\!_2 n^2$ LOps.

n	${}^1\!/\!_2\, n^2$	${}^1\!/\!_2\, n^2 + {}^1\!/\!_2\, n$
10	50	55
100	5 000	5 050
1000	500 000	500 500

General systems: Gaussian elimination (page 72–77)

Example:

$$\begin{bmatrix} 5 & -5 & 10 \\ 2 & 0 & 8 \\ 1 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -25 \\ 6 \\ 9 \end{bmatrix} \qquad \begin{matrix} (1) \\ (2) \\ (3) \end{matrix}$$

$$\begin{bmatrix} 5 & -5 & 10 \\ 0 & 2 & 4 \\ 0 & 2 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -25 \\ 16 \\ 14 \end{bmatrix} \qquad \begin{matrix} (1) \\ (2') = (2) - \frac{2}{5} \cdot (1) \\ (3') = (3) - \frac{1}{5} \cdot (1) \end{matrix}$$

$$\begin{bmatrix} 5 & -5 & 10 \\ 0 & 2 & 4 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -25 \\ 16 \\ -2 \end{bmatrix} \qquad \begin{matrix} (1) \\ (2') \\ (3'') = (3') - 1 \cdot (2') \end{matrix}$$

Back substitution gives:

$$\boldsymbol{x} = \begin{bmatrix} -5 \\ 4 \\ 2 \end{bmatrix} .$$

Gaussian elim. creates 0's below the main diagonal (page 72–77)

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & \otimes & \times & \times \\ & & \otimes & \times & \times \end{bmatrix} \begin{array}{l} \leftarrow \text{ the row is finished} \\ \leftarrow \text{ the row is finished} \\ \leftarrow \text{ pivot row} \\ \leftarrow \text{ create a 0 at the element } \otimes \\ \leftarrow \text{ create a 0 at the element } \otimes \end{array}$$

<u>Algorithm:</u>

for $k = 1 : n - 1$
    for $i = k + 1 : n$
        xmult $= a_{ik}/a_{kk}$
        for $j = k + 1 : n$
            $a_{ij} = a_{ij} - \text{xmult} \cdot a_{kj}$
        end
        $b_i = b_i - \text{xmult} \cdot b_k$
    end
end

Pivoting – mathematical reason

Example:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

multiplier:

$$\text{xmult} = a_{21}/a_{11} = 1/0 = \infty,$$

Gaussian elimination is broken down 

But $\boldsymbol{A}$ is nonsingular – interchange row 1 and 2:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$x_2 = 1/1 = 1 \ , \qquad x_1 = (2 - 1 \cdot 1)/1 = 1 \ .$$

Pivoting – numerical reason

Solve by Gaussian elimination, but with different number of significant digits:

$$\begin{bmatrix} 0.0003 & 3.0000 \\ 1.0000 & 1.0000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2.0001 \\ 1.0000 \end{bmatrix}$$

digits	x_1	x_2	rel. error in x_1
3	-3.00	0.667	-11
4	0.0000	0.6667	1
5	0.30000	0.66667	0.1
6	0.330000	0.666667	0.01
7	0.3330000	0.6666667	0.001
∞	0.333333...	0.6666666...	

Quite large error, although we have many significant digits.

Now with pivoting!

$$\begin{bmatrix} 1.0000 & 1.0000 \\ 0.0003 & 3.0000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.0000 \\ 2.0001 \end{bmatrix}$$

digits	x_1	x_2	rel. error in x_1
3	0.333	0.667	0.001
4	0.3333	0.6667	0.0001
5	0.33333	0.66667	0.00001
6	0.333333	0.666667	0.000001
7	0.3333333	0.6666667	0.0000001
∞	0.333333...	0.6666666...	

Much smaller error – even with few significant digits!

$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$: risk for *loss of significance* without pivoting

Let $\epsilon \ll 1$ be a small number (e.g. $\epsilon = 0.0003$):

$$\begin{bmatrix} \epsilon & 3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2.0001 \\ 1 \end{bmatrix}, \qquad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 2/3 \end{bmatrix}.$$

Gaussian elimination without pivoting:

$$\begin{aligned} \text{xmult} &= \frac{a_{21}}{a_{11}} = \frac{1}{\epsilon} \gg 1 \\ a'_{22} &= a_{22} - \text{xmult} \cdot a_{12} = 1 - \frac{1}{\epsilon} \cdot 3 \approx -\frac{3}{\epsilon} \\ b'_2 &= b_2 - \text{xmult} \cdot b_1 = 1 - \frac{1}{\epsilon} \cdot 2.0001 \approx -\frac{2}{\epsilon}\ . \end{aligned}$$

Without pivoting: error

Applying Gaussian elimination without pivoting, we get

$$\boldsymbol{A}' = \begin{bmatrix} a_{11} & a_{12} \\ 0 & a'_{22} \end{bmatrix} = \begin{bmatrix} \epsilon & 3 \\ 0 & 1 - {}^{3}/_{\epsilon} \end{bmatrix}, \qquad \boldsymbol{b}' = \begin{bmatrix} 2.0001 \\ 1 - {}^{2.0001}/_{\epsilon} \end{bmatrix}.$$

With very small ϵ, the computer actually calculates:

$$\widehat{\boldsymbol{A}}' = \begin{bmatrix} a_{11} & a_{12} \\ 0 & \widehat{a}'_{22} \end{bmatrix} = \begin{bmatrix} \epsilon & 3 \\ 0 & -{}^{3}/_{\epsilon} \end{bmatrix}, \qquad \widehat{\boldsymbol{b}}' = \begin{bmatrix} 2.0001 \\ -{}^{2.0001}/_{\epsilon} \end{bmatrix}.$$

Then, the absolute error in calculation is:

$$\boldsymbol{A}' - \widehat{\boldsymbol{A}}' = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \qquad \boldsymbol{b}' - \widehat{\boldsymbol{b}}' = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

NB: Errors appearing in $\boldsymbol{A}$ and $\boldsymbol{b}$ are in the same order!

Without pivoting: What went wrong?

Calculate the solution:

$$
\begin{aligned}
\widehat{x}_2 &= \widehat{b}'_2 / \widehat{a}'_{22} \\
&= (-{}^{2.0001}/_{\epsilon}) / (-{}^{3}/_{\epsilon}) = {}^{2.0001}/_{3} \approx {}^{2}/_{3}
\end{aligned}
$$

$$
\begin{aligned}
\widehat{x}_1 &= (b_1 - a_{12}\,\widehat{x}_2)/a_{11} \\
&= (2.0001 - 3 \cdot {}^{2.0001}/_{3})/\epsilon = 0
\end{aligned}
$$

The result of $\widehat{x}_2$ is OK, but $\widehat{x}_1$ is *completely wrong!*

The problem is due to **loss of significance in subtraction** in the computation. For example: with $\epsilon = 0.0003$:

$$a'_{22} = a_{22} - f \cdot a_{12} = 1 - {}^{1}/_{\epsilon} \cdot 3 = 1 - 10^4 \to -10^4.$$

Note: Avoid elements in the matrix increasing in the Gaussian elimination.

$$\boxed{\text{Keep the multiplier } |\text{xmult}| \text{ small!}}$$

Now – with pivoting:

The pivot system:

$$\begin{bmatrix} 1 & 1 \\ \epsilon & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2.0001 \end{bmatrix}, \qquad \text{xmult} = \frac{a_{21}}{a_{11}} = \epsilon \quad \text{(It is small!)}$$

Apply Gaussian elimination:

$$a'_{22} = 3 - \epsilon \cdot 1 \to \widehat{a}'_{22} = 3, \qquad b'_2 = 2.0001 - \epsilon \cdot 1 \to \widehat{b}'_2 = 2.0001 \ .$$

The errors $\boldsymbol{A}' - \widehat{\boldsymbol{A}}' = \begin{bmatrix} 0 & 0 \\ 0 & \epsilon \end{bmatrix}$ and $\boldsymbol{b}' - \hat{\boldsymbol{b}}' = \begin{bmatrix} 0 \\ \epsilon \end{bmatrix}$ are small now!

$$\begin{aligned} \widehat{x}_2 &= \hat{b}'_2 / \hat{a}'_{22} = 2.0001/3 = 0.6667 \approx {}^2/_3 \\ \widehat{x}_1 &= (b_1 - a_{12}\,\widehat{x}_2)/a_{11} = (1 - 1 \cdot {}^{2.0001}/_3)/1 = 0.3333 \approx {}^1/_3 \ . \end{aligned}$$

It is much better now: $\boldsymbol{x} - \widehat{\boldsymbol{x}} \approx \begin{bmatrix} 3.33 \cdot 10^{-5} \\ -3.33 \cdot 10^{-5} \end{bmatrix}$

Gaussian elimination with partial pivoting

$$\begin{array}{cl} \begin{bmatrix} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \color{red}{\times} & \times & \times & \times \\ & & \color{red}{\times} & \times & \times & \times \\ & & \color{red}{\times} & \times & \times & \times \\ & & \color{red}{\times} & \times & \times & \times \end{bmatrix} & \begin{array}{l} \\ \\ \leftarrow \text{ row } k \\ \\ \leftarrow \text{ row } \ell \\ \\ \end{array} \end{array}$$

Algorithm: for $k = 1 : n-1$

find index ℓ such that $|a_{\ell k}| \geq |\color{red}{a_{ik}}|$, $i = k, \ldots, n$

interchange row k and ℓ in $\boldsymbol{A}$ as well as b_k and b_ℓ

for $i = k+1 : n$

xmult $= a_{ik}/a_{kk}$

for $j = k+1 : n$

$a_{ij} = a_{ij} - \text{xmult} \cdot a_{kj}$

end

$b_i = b_i - \text{xmult} \cdot b_k$

end

end

Gaussian elimination with *scaled* partial pivoting (page 85–89)

There are many strategies for pivoting. Here we introduce another one:

At the beginning of Gaussian eliminationen, we compute a *scale factor* $\boldsymbol{s}$ for each equation in the system as

$$s_i = \max_{1 \le j \le n} |a_{ij}| = \texttt{np.max(np.abs(A[i-1,:]))}, \qquad i = 1, 2, \ldots, n \; .$$

In Step k, we use the equation whose ratio $|a_{lk}/s_l|$ is the largest as the pivot equation, i.e,

$$\frac{|a_{\ell k}|}{s_\ell} \ge \frac{|a_{ik}|}{s_i}, \qquad i = k, \ldots, n \; .$$

See next slide $\rightarrow$

Gaussian elimination with *scaled* partial pivoting – code

$$\left[\begin{array}{cccccc} \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & \times & \times & \times & \times \end{array}\right] \begin{array}{l} \\ \\ \leftarrow \text{ row } k \\ \\ \leftarrow \text{ row } \ell \\ \\ \end{array}$$

Algorithm:

$$\begin{array}{l}
\text{Compute scale factor } \boldsymbol{s} \\
\text{for } k = 1 : n-1 \\
\quad \text{find index } \ell \text{ such that } |a_{\ell k}|/s_\ell \geq |a_{ik}|/s_i,\ i = k, \ldots, n \\
\quad \text{swap row } k \text{ and } \ell \text{ in } \boldsymbol{A} \text{ same as } b_k \text{ and } b_\ell \\
\quad \text{for } i = k+1 : n \\
\quad\quad \text{xmult} = a_{ik}/a_{kk} \\
\quad\quad \text{for } j = k+1 : n \\
\quad\quad\quad a_{ij} = a_{ij} - \text{xmult} \cdot a_{kj} \\
\quad\quad \text{end} \\
\quad\quad b_i = b_i - \text{xmult} \cdot b_k \\
\quad \text{end} \\
\text{end}
\end{array}$$

Computational complexity of Gaussian elimination (page 92–93)

The *dominant* number of long operations in Gaussian elimination with scaled partial pivoting is:

$$\begin{aligned} \text{LOps} &= n^2 + (n-1)^2 + (n-2)^2 + \cdots + 4^2 + 3^2 + 2^2 \\ &= {}^{n}\!/\!_{6}\,(n+1)\,(2n+1) - 1 \\ &\approx {}^{1}\!/\!_{3}\,n^3 \;. \end{aligned}$$

The corresponding number of long operation in "forward processing" of the right-hand side is:

$$\text{LOps} = 1 + 2 + 3 + \cdots + (n-1) = {}^{1}\!/\!_{2}\,n\,(n-1) \approx {}^{1}\!/\!_{2}\,n^2 \;.$$

The total number for solving $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$ is:

$$\text{LOps} \approx {}^{1}\!/\!_{3}\,n^3 + {}^{1}\!/\!_{2}\,n^2 + {}^{1}\!/\!_{2}\,n^2 \approx {}^{1}\!/\!_{3}\,n^3 \;.$$

Simple ("naive") Gaussian elimination (page 358–359)

The algorithm overwrites the current elements in $\boldsymbol{A}$ and $\boldsymbol{b}$:

for $k = 1 : n - 1$
    for $i = k + 1 : n$
        $f_{ik} = a_{ik}/a_{kk} \quad \leftarrow$ the multiplier
        for $j = k + 1 : n$
            $a_{ij} = a_{ij} - f_{ik} \cdot a_{kj}$
        end
        $b_i = b_i - f_{ik} \cdot b_k$
    end
end

Simple ("naive") LU-factorization

Another formulation of Gaussian elimination: we can alway have

$$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{U},$$

where

- $\boldsymbol{U}$ is an upper triangular *after* Gaussian elimination, and
- $\boldsymbol{L}$ is a unit lower triangular including elimination coefficients:

$$\ell_{ik} = f_{ik}, \quad k < i \qquad \text{and} \qquad \ell_{kk} = 1.$$

Unit triangular means that the diagonal elements of $\boldsymbol{L}$ are all 1's.

These are very useful **notations**.

Numerical example (page 359–362)

$$\boldsymbol{M}_1\,\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1/2 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \\ 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \end{bmatrix} = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \end{bmatrix}$$

NB: we have $f_{21} = 2$, $f_{31} = 1/2$ and $f_{41} = -1$.

$$\boldsymbol{M}_2\,\boldsymbol{M}_1\,\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -3 & 1 & 0 \\ 0 & 1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \end{bmatrix} = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 4 & -13 \end{bmatrix}$$

$$\boldsymbol{M}_3\,\boldsymbol{M}_2\,\boldsymbol{M}_1\,\boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -2 & 1 \end{bmatrix} \begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 4 & -13 \end{bmatrix} = \underbrace{\begin{bmatrix} 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \\ 0 & 0 & 2 & -5 \\ 0 & 0 & 0 & -3 \end{bmatrix}}_{\boldsymbol{U}}$$

Numerical example

$$
\begin{aligned}
\boldsymbol{M}_3\,\boldsymbol{M}_2\,\boldsymbol{M}_1\,\boldsymbol{A} &= \boldsymbol{U} \\
\boldsymbol{M}_2\,\boldsymbol{M}_1\,\boldsymbol{A} &= \boldsymbol{M}_3^{-1}\boldsymbol{U} \\
\boldsymbol{M}_1\,\boldsymbol{A} &= \boldsymbol{M}_2^{-1}\boldsymbol{M}_3^{-1}\boldsymbol{U} \\
\boldsymbol{A} &= \boldsymbol{M}_1^{-1}\boldsymbol{M}_2^{-1}\boldsymbol{M}_3^{-1}\boldsymbol{U} = \boldsymbol{L}\,\boldsymbol{U}
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{L} = \boldsymbol{M}_1^{-1}\boldsymbol{M}_2^{-1}\boldsymbol{M}_3^{-1} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1/2 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & -1/2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1/2 & 3 & 1 & 0 \\ -1 & -1/2 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ f_{21} & 1 & 0 & 0 \\ f_{31} & f_{32} & 1 & 0 \\ f_{41} & f_{42} & f_{43} & 1 \end{bmatrix}
\end{aligned}
$$

The proof in page 363-364 is skipped.

Gaussian elimination with pivoting

Here: We only consider partial pivoting (same principle for scaled partial pivoting):

for $k = 1 : n-1$
    interchange the rows such that $|a_{kk}| = \max_{k \leq i \leq n} |a_{ik}|$ $\leftarrow$ pivoting
    make the same interchange in b
    for $i = k+1 : n$
        $f_{ik} = a_{ik}/a_{kk}$ $\leftarrow$ the multiplier
        for $j = k+1 : n$
            $a_{ij} = a_{ij} - f_{ik} \cdot a_{kj}$
        end
        $b_i = b_i - f_{ik} \cdot b_k$
    end
end

LU factorization with pivoting

Same idea as before: we can always have

$$\boldsymbol{P}\,\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{U},$$

where

- $\boldsymbol{P}$ is a permutation matrix,
- $\boldsymbol{U}$ is an upper triangular *after* Gaussian eliminationen, and
- $\boldsymbol{L}$ is a unit lower triangular including elimination coefficients:

$$\ell_{ik} = f_{ik}, \quad k < i \qquad \text{and} \qquad \ell_{kk} = 1.$$

The only difference is the matrix $\boldsymbol{P}$.

Permutation matrix

A permutation matrix is a square binary matrix that has exactly one entry of 1 in each row and each column and 0s elsewhere.

$$\boldsymbol{P} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

It is used to interchange elements:

$$\boldsymbol{P}\,\boldsymbol{b} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} = \begin{bmatrix} b_3 \\ b_4 \\ b_1 \\ b_5 \\ b_2 \end{bmatrix}$$

Solving linear system using LU factorization (≈ page 365)

First, we compute the factorization of $\boldsymbol{A}$ in the linear system $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$, and get

$$\boldsymbol{P}\,\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{P}\,\boldsymbol{b} \qquad \Leftrightarrow \qquad \boldsymbol{L}\,\boldsymbol{U}\,\boldsymbol{x} = \boldsymbol{P}\,\boldsymbol{b}\ .$$

Then we can solve the system by the following three steps:

1. Permute the right-hand side: $\hat{\boldsymbol{b}} = \boldsymbol{P}\,\boldsymbol{b}$.
2. Solve $\boldsymbol{L}\,\boldsymbol{y} = \hat{\boldsymbol{b}} \;\Leftrightarrow\; \boldsymbol{y} = \boldsymbol{L}^{-1}\hat{\boldsymbol{b}}$ (forward substitution).
3. Solve $\boldsymbol{U}\,\boldsymbol{x} = \boldsymbol{y} \;\Leftrightarrow\; \boldsymbol{x} = \boldsymbol{U}^{-1}\boldsymbol{y}$ (back substitution).

The result: $\boldsymbol{x} = \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{P}\,\boldsymbol{b}$.

$$\boldsymbol{P}\,\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{U} \Leftrightarrow \boldsymbol{A} = \boldsymbol{P}^{-1}\boldsymbol{L}\,\boldsymbol{U} \Leftrightarrow \boldsymbol{A}^{-1} = \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{P}$$

When we have $\boldsymbol{L}$ and $\boldsymbol{U}$, it only costs n^2 LOps to solve $\boldsymbol{A}\,\boldsymbol{x}_{\text{ny}} = \boldsymbol{b}_{\text{ny}}$.

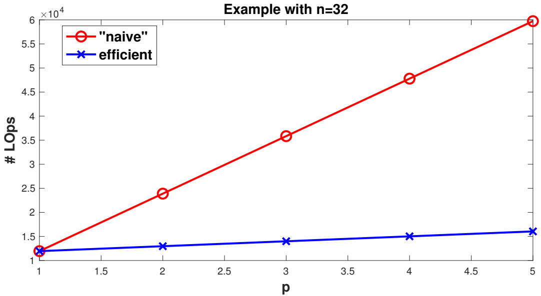
It is cheap to solve the *new right-hand side.*

Example with multiple right-hand sides (page 371)

An example to show the advantage of using LU-factorization for multiple right-hand sides:

$$\boldsymbol{A}\,\boldsymbol{x}^{(1)} = \boldsymbol{b}^{(1)}, \quad \boldsymbol{A}\,\boldsymbol{x}^{(2)} = \boldsymbol{b}^{(2)}, \quad \ldots, \quad \boldsymbol{A}\,\boldsymbol{x}^{(p)} = \boldsymbol{b}^{(p)}.$$

- "Naive" implementation: p Gaussian elimination and p back substitutions. It costs totally $p(n^3/3 + n^2)$ LOps.
- Efficient implementation: One LU-factorization with p forward and back substitutions, costs $n^3/3 + p\,n^2$ LOps.



LU factorization in Python with `scipy.linalg.lu`

In Python we can calculate LU factorization by using the function `lu` from scipy.linalg:

```
>>> import numpy as np
>>> from scipy.linalg import lu

>>> A = np.array([ [0.1, 7, -0.3],
                   [0.3, -0.2, 10],
                   [3, -0.1, -0.2] ])

>>> P, L, U = lu(A)
```

LU factorization in Python with `scipy.linalg.lu`

```
>>> print(f'P =\n {P}\n L =\n {L}\n U =\n {U}\n')
P =
0     0     1
1     0     0
0     1     0
L =
1.0000        0         0
0.0333   1.0000         0
0.1000  -0.0271    1.0000
U =
3.0000  -0.1000   -0.2000
0        7.0033   -0.2933
0        0        10.0120

>>> tmp = scipy.linalg.solve_triangular(L, P@b, lower=True)
>>> x = scipy.linalg.solve_triangular(U, tmp, lower=False)
```

Summary

- **Newton's method.**
  It can be generalized to solve m equations with m variables.
  In every iteration, it **requires to solve a linear system**.
- **Computational complexity**.
  Computational complexity of solving $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$ is $1/3 n^3$ LOps (multiplications and divisions). That is, if n is multiplied by 10, then it costs 1000 times more LOps and 1000 times longer computing time.
- **Reason for pivoting**.
  Mathematically: avoid dividing by 0.
  Numerically: avoid large multipliers in row eliminations, which can lead to loss of significance.
- **Partial pivoting**.
  Find the numerical largest element in the current column, and interchange the rows such that this element stays in the diagonal.