Lecture 2

Solving the linear systems of algebraic equations with direct methods

outline

- remainder of some basic properties of matrices
- norms of vectors & matrices
- Gauss elimination method
- LU factorization of square matrix
- LDL^T & LL^T factorizations for symmetric matrices
- LU factorization of tridiagonal matrix
- Gram-Schmidt orthogonalization & QR factorization
- overdetermined systems of linear equations

- examples:
    - solution of Poisson equation with finite difference method
    - shaping polynomial function

- short survey over linear algebra numerical packages: BLAS & LAPACK

Basic properties of matrices

Matrix - it is an ordered set of real or complex number placed in 2D array,
each element has two indices (row & column) allowing for its localization in array

$$A = [a_{i,j}] \in C^{m \times n}, \quad i = 1, \ldots, m, \quad j = 1, \ldots, n$$

$$A_{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}$$

$$m = n \implies \text{square matrix}$$

$$m > n \implies \text{rectangular matrix}$$

$$m < n \implies \text{rectangular matrix but useless for our purposes}$$

rank(A) – the number of linearly independent columns,
full rank square matrix is nonsingular and invertible

$$rank(A) = m = n \implies Det(A) \neq 0 \implies A^{-1} \text{ exists}$$

Transposition of matrix – we change the order of indices of matrix elements, (for square matrix it is rotation about its diagonal)

$$A = [a_{i,j}] \quad \Longrightarrow \quad A^T = [a_{j,i}]$$

properties:

$$(A^T)^T = A$$

$$(\alpha A)^T = \alpha A^T$$

$$(A + B)^T = A^T + B^T$$

$$(AB)^T = B^T A^T$$

$$(A\vec{x})^T = \vec{x}^T A^T$$

Nonsingular / invertible matrix – nonsingular matrix is invertible

$$Det(A) \neq 0 \qquad A^{-1}\text{exists}$$

properties:

$$AA^{-1} = A^{-1}A = I$$

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

Symmetry of square matrix – knowledge of symmetry allows to use dedicated factorization/diagonalization methods which can work faster, be more accurate and stable

$$A \in R^{n\times n}: \quad A^T = A \quad \text{- symmetric matrix}$$

$$A \in C^{n\times n}: \quad A^H = \left(A^T\right)^* = A \quad \text{- symmetric hermitian matrix}$$

$$A \in C^{n\times n}: \quad A^T = A \quad \text{- complex symmetric matrix}$$

Orthogonal matrix - column vectors or row vectors are mutually orthogonal (euclidean norm), matrix can be rectangular or square

$$Q \in C^{m\times n}: \quad Q^T Q = I \qquad \leftarrow \text{if columns/rows are normalized}$$

$$Q^{-1} Q = I$$

$$Q^{-1} = Q^T$$

example: rotation matrix in 2D & 3D

$$O = \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} \quad \Longrightarrow \quad O^{-1} = O^T = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix}$$

Vector norms

- norms are used for quantitative descriptions of vectors and matrices
- vector (matrix) norm is a function which assigns single nonnegative real number for each vector defined in vector space C^n (C^{mxn})

For any two vectors and a number such as

$$\vec{x}, \vec{y} \in C^n, \quad \alpha \in C$$

vector norm must fullfill the following axioms

nonnegative: $$||\vec{x}|| \geq 0, \quad ||\vec{x}|| = 0 \iff \vec{x} = \vec{0}$$

multiply by number: $$||\alpha\vec{x}|| = |\alpha| \cdot ||\vec{x}||$$

triangle inequality: $$||\vec{x} + \vec{y}|| \leq ||\vec{x}|| + ||\vec{y}||$$

For n-dimensional vectors we use family of **p-norms**

$$\vec{x} = [x_1, x_2, \ldots, x_n]^T$$

$$||\vec{x}||_p = (|x_1|^p + |x_2|^p + \ldots |x_n|^p)^{\frac{1}{p}}$$

example norms:

$$p = 1, \quad ||\vec{x}||_1 = |x_1| + |x_2| + \ldots + |x_n|$$

$$p = 2, \quad ||\vec{x}||_2 = (x_1^2 + x_2^2 + \ldots + x_n^2)^{1/2}$$

$$p = \infty, \quad ||\vec{x}||_\infty = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$$

Remark: in numeriacal algorithms in nearly all cases we use p=2 (euclidean) norm, other norms are exploited in more theoretical applications

Among all vector norms, the 2-norm (Euclidean) is the most often used. It is closely related to **vector scalar (inner) product** which can be symbolically written in few ways (and worth to remember) depending on the scientific field

- mathematicians notation

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$$

- computer science notation (the „rule” derived from matrix-vector multiplication)

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{bmatrix}$$

$$\vec{a}^T = [a_1, a_2, \dots, a_n]$$

$$\vec{a}^T \vec{b} = \boldsymbol{a}^T \boldsymbol{b} = \begin{bmatrix} a_1 & a_2 & \dots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} a_i b_i$$

- physicists (Dirac) notation

$$\langle \vec{a} | \vec{b} \rangle = \sum_{i=1}^{n} a_i b_i$$

Matrix norms

Matrix norms must obey the followig axioms

$$
\begin{aligned}
||A|| &\geq 0, \quad ||A|| = 0 \iff A = 0 \\
||\alpha A| &= |\alpha| \cdot ||A|| \\
||A + B|| &\leq ||A|| + ||B|| \\
||AB|| &\leq ||A|| \cdot ||B|| \\
||A\vec{x}|| &\leq ||A|| \cdot ||\vec{x}|| \quad \text{(compatible vector \& matrix norms)}
\end{aligned}
$$

Compatible norms – matrix norm is generated by vector norm as follows

$$
\|A\|_{pq} = \frac{\|A\vec{x}\|_q}{\|\vec{x}\|_p}
$$

shorthand notation: $p = q \quad \Longrightarrow \quad \|A\|_{pq} = \|A\|_{pp} = \|A\|_p$

examples of matrix norms:

$$\|A\|_1 = \max_{j=1,2,\ldots,n} \sum_{i=1}^{m} |a_{ij}|$$ - the largest sum of elements' modulus in columns

$$\|A\|_2 = \left(\max_{i=1,\ldots,n} \lambda_i(AA^T) \right)^{1/2}$$ - spectral norm

$$\|A\|_\infty = \max_{i=1,2,\ldots,n} \sum_{j=1}^{n} |a_{ij}|$$ -the largest sum of elements' modulus in rows

$$\|A\|_{1\infty} = \max_{i,j} |a_{ij}|$$ -the largest modulus of single element

Gauss elimination method

Our aim is to find the solution of system of linear equations (SLE).
SLE expressed in square form can not be solved directly, it must be first transformed to the triangle form which then allows to find elements of solution vector sequentially one by one.

original SLE in full square form:

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

STEP 1
Gauss elimination method

transformed SLE in triangular form:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ldots\ldots\ldots\ldots &\ldots \quad \ldots \\ a_{nn}x_n &= b_n \end{aligned}$$

STEP 2
backward substitution

we start from x_n, then calculate x_{n-1}, x_{n-2},...,x_1

$$\begin{aligned} x_n &= \frac{b_n}{a_{nn}} \\ x_i &= \frac{b_i - a_{ii+1}x_{i+1} - \ldots - a_{in}x_n}{a_{ii}} \end{aligned}$$

solution

$$\vec{x} = [x_1, x_2, \ldots, x_n]$$

how to get triangular form of SLE?

- we must eliminate all matrix elements below the diagonal
- elimination is done iteratively, only elements in single column are cancelled per single iteration
- iterative process starts at the first column and end in n-1 column

Here we use SLE in explicit form

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

but for our numerical purposes it will be expressed rather in matrix form

$$A\vec{x} = \vec{b}$$

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

another shorthand notation concerns parts of matrix A

- k-th row of matrix $A_{k,*}$
- m-th column of matrix $A_{*,m}$

Elimination process

original SLE

$$\begin{aligned} a_{11}^{(1)}x_1 + a_{12}^{(1)}x_2 + \ldots + a_{1n}^{(1)}x_n &= b_1^{(1)} \\ a_{21}^{(1)}x_1 + a_{22}^{(1)}x_2 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots &= \ldots \\ a_{n1}^{(1)}x_1 + a_{n2}^{(1)}x_2 + \ldots + a_{nn}^{(1)}x_n &= b_n^{(1)} \end{aligned}$$

$$l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \qquad \text{column: 1, rows: i=2,...,n}$$

$$[A_{i,*}, b_i]^{(1)} - l_{i1}\,[A_{1,*}, b_1]^{(1)} = [A_{i,*}, b_i]^{(2)}$$

we get SLE with different (changed) coefficients $a_{ij}^{(2)}$ and $b_i^{(2)}$

$$\begin{aligned} a_{11}^{(2)}x_1 + a_{12}^{(2)}x_2 + \ldots + a_{1n}^{(2)}x_n &= b_1^{(2)} \\ a_{22}^{(2)}x_2 + \ldots + a_{2n}^{(2)}x_n &= b_2^{(2)} \\ \ldots\ldots\ldots\ldots\ldots\ldots &= \ldots \\ a_{n2}^{(2)}x_2 + \ldots + a_{nn}^{(2)}x_n &= b_n^{(2)} \end{aligned}$$

$$l_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}} \qquad \text{column: 2, rows: i=3,...,n}$$

$$[A_{i,*}, b_i]^{(2)} - l_{i1}\,[A_{2,*}, b_2]^{(2)} = [A_{i,*}, b_i]^{(3)}$$

we get SLE with coefficients $a_{ij}^{(3)}$ and $b_i^{(3)}$

$$\begin{aligned} a_{11}^{(3)}x_1 + a_{12}^{(3)}x_2 + \ldots + a_{1n}^{(3)}x_n &= b_1^{(3)} \\ a_{22}^{(3)}x_2 + \ldots + a_{2n}^{(3)}x_n &= b_3^{(2)} \\ \ldots\ldots\ldots\ldots &= \ldots \\ \ldots + a_{nn}^{(3)}x_n &= b_n^{(3)} \end{aligned}$$

after n-1 iterations we get the trangular SLE with different coeficients a_{ij} and b_i

$$\begin{aligned} a_{11}^{(n-1)}x_1 + a_{12}^{(n-1)}x_2 + \ldots + a_{1n}^{(n-1)}x_n &= b_1^{(n-1)} \\ a_{22}^{(n-1)}x_2 + \ldots + a_{2n}^{(n-1)}x_n &= b_2^{(n-1)} \\ \ldots\ldots\ldots\ldots &= \ldots \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned}$$

Remark: to avoid division by 0 (1) rows or (2) rows & columns can be permuted – partial/full selection of main element

LU factorization

The Gauss elimination method is an iterative process which transforms SLE,
if we write SLE in matrix form we will see that elimination transforms the matrix of SLE

$$A^{(1)}\vec{x} = \vec{b}^{(1)}$$
$$A^{(2)}\vec{x} = \vec{b}^{(2)}$$
$$\vdots$$
$$A^{(n-1)}\vec{x} = \vec{b}^{(n-1)} \quad \Rightarrow \quad U\vec{x} = \vec{b}^{(n-1)}$$

upper triangular matrix

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

Process of transformation of the matrix is called factorization and any kind of factorization
may be useful as it makes the problem of solving SLE much easier.

The Gauss eliminiation is very popular method utilized to find LU factorization,
namely, we wish to express the matrix A as product of two matrices: L-lower triangle and U-upper triangle

$$A = LU$$

$$\mathrm{L} = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & 0 & 0 \\ \dots & \dots & \dots & 1 & 0 \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{bmatrix} \qquad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

Applications of LU factorization

- **solving single SLE**

forward substitution

backward substitution

$$A\vec{x} = \vec{b} \quad \rightarrow \quad L\underbrace{(U\vec{x})}_{\vec{y}} = \vec{b} \quad \rightarrow \quad L\vec{y} = \vec{b} \quad \rightarrow \quad U\vec{x} = \vec{y}$$

- **finding the value of determinant of matrix A** (e.g. to check the invertibility of matrix)

$$\det(\mathrm{A}) = \det(\mathrm{LU}) = \underbrace{det(L)}_{=1} det(U) = det(U) = \prod_{i=1}^{n} u_{i,i}$$

- **finding the inverse of matrix A** (we should avoid it if possible)

lets define the right side of SLE as Euclidean basis vector e_i

$$\vec{e}_i = [0, 0, \ldots, 1, \ldots, 0]^T$$

we may solve such sinlge SLE using LU

$$LU\vec{x}_i = \vec{e}_i, \quad i = 1, 2, \ldots, n$$

as well as a set of n such SLEs

$$LU\underbrace{[\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n]}_{X} = \underbrace{[\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n]}_{I} \quad \rightarrow \quad LUX = I \rightarrow X = A^{-1}$$

LU factorization of tridiagonal matrix

Tridiagonal matrices very often appear in 1D differential problems with derivatives approximated with finite differences. We can exploit the spectial structure of a matrix i.e. operations can be performed for non-zero elements only

tridiagonal matrix:

$$T = \begin{bmatrix} d_1 & c_1 & & & & \\ a_2 & d_2 & c_2 & & & \\ & a_3 & d_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & d_n \end{bmatrix}$$

$$T = LU$$

$$L = \begin{bmatrix} 1 & & & & & \\ l_2 & 1 & & & 0 & \\ & l_3 & 1 & & & \\ & & \ddots & \ddots & & \\ & 0 & & \ddots & \ddots & \\ & & & & l_n & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} u_1 & c_1 & & & & \\ & u_2 & c_2 & & 0 & \\ & & u_3 & c_3 & & \\ & & & \ddots & \ddots & \\ & 0 & & & \ddots & c_{n-1} \\ & & & & & u_n \end{bmatrix}$$

All we need to do is to find the two diagonal vectors of elements l_i and u_i (c_i are the same as in T).

The elements l_i and u_i are computed sequentially

$$u_1 = d_1$$

$$l_i = \frac{a_i}{u_{i-1}}$$

$$u_i = d_i - l_i c_{i-1}, \quad i = 2, 3, \ldots, n$$

Having LU we fast find the solution of SLE by performing forward and backward substitutions

$$T\vec{x} = LU\vec{x} = L\vec{y} = \vec{b}$$

$$L\vec{y} = \vec{b} \quad \Rightarrow \quad y_1 = b_1$$

$$y_i = b_i - l_i y_{i-1}, \quad i = 2, \ldots, n$$

$$U\vec{x} = \vec{y} \quad \Rightarrow \quad x_n = \frac{y_n}{u_n}$$

$$x_i = \frac{y_i - c_i x_{i+1}}{u_i}, \quad i = n-1, n-2, \ldots, 1$$

LU factorization is general and can be applied to any square matrix A.

However, for symmetric matrices we have other dedicated factorizations which:

- work 2 times faster
- need only half the space (upper or lower matrix A is needed by numerical procdeure)

- **LL^T (Cholesky factorization)** - used for symmetric/hermitean, positively defined matrix, (L is lower triangular matrix)

positively defined matrix: $\forall \vec{x} \in C^n : \quad \vec{x}^T A \vec{x} > 0$

factorization: $A = LL^T$

solution of SLE: $A\vec{x} = L\underbrace{(L^T\vec{x})}_{\vec{y}} = \vec{b} \implies L\vec{y} = \vec{b} \implies L^T\vec{x} = \vec{y}$

- **LDL^T** – used for symmetric/hermitean if we don't know if the matrix is positively defined, (L and D are the lower triangular and the diagonal matrix, respectively)

factorization: $A = LDL^T$

solution of SLE: $A\vec{x} = \vec{b}$

$$LD\underbrace{L^T\vec{x}}_{\vec{y}} = L\underbrace{D\vec{y}}_{\vec{z}} = L\vec{z} = \vec{b}$$

we solve 3 SLEs:

$$L\vec{z} = \vec{b}$$
$$D\vec{y} = \vec{z}$$
$$L^T\vec{x} = \vec{y}$$

Correctness of the SLE solution → residual vector

We must keep in mind that transformation/factorization of original matrix and then forward & backward substitutions are made within in-exact arithmetic (real numbers are represented by the floating-point ones → rounding errors). Rounding errors may perturb to some extent the solution vector.

Routinely we shall check the level of perturbation of the solution vector calculating **the residual vector**

- exact solution

$$A\vec{x} = \vec{b} \quad \Longrightarrow \quad \vec{r} = \vec{b} - A\vec{x} = \vec{0} \quad \text{- residual vector}$$

- numerical/perturbed solution

$$A\vec{x}_p = \vec{b} \quad \Longleftrightarrow \quad \vec{x}_p = \vec{x} + \delta\vec{x}$$

$$A\vec{x} + A\delta\vec{x} = \vec{b} \quad \Longrightarrow \quad \vec{r} = \underbrace{\vec{b} - A\vec{x}}_{=0} - A\delta\vec{x} \neq \vec{0}$$

Residual vector can be used to improve the solution, we shall just solve one more SEL

$$-A\delta\vec{x} = \vec{r}$$

$$\vec{x} \approx \vec{x}_p - \delta\vec{x}$$

→ solution is not exact because solution of second SLE is also perturbed

Remark: solution shall be only one or two times improved and no more,
rounding errors do not allow for fixing „largely broken solution”,
usually iterative improvement is implicitly done by numerical package we have used to solve the SEL

Conditioning of the SLE matrix

Conditioning means how the rounding errors influence on the solution of SLE, namely we can define perturbation of solution as dependent on:

- perturbation of the matrix element

$$\frac{||\delta\vec{x}||}{||\vec{x}+\delta\vec{x}||} \leq \kappa(A)\frac{||\delta A||}{||A||}$$

- perturbation of the right hand side vector elements

$$\frac{||\delta\vec{x}||}{||\vec{x}||} \leq \kappa(A)\frac{||\delta\vec{b}||}{||\vec{b}||}$$

perturbations δA and δb stand for rounding errors which occur during factorization or forward/backward substitution

Both expressions are linked by the scaling factor known as **condition number**

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

Remark 1: condition number is an inherent property of matrix

Remark 2: large condition number (e.g. $> 10^6$-10^8) may lead to largely perturbed (inaccurate) solutions which can not be accepted, on the other hand, the small number increases our chance for finding good solution i.e. with small residual vector

Remark 3: even for large condition number we still can solve SLE, we can minimize perturbations δA and δb by exploting stronger floating-point arithmetic

Example 1: Solution of Poisson equation with Finte Difference Method (FDM)

- general form of Poisson equation: $\nabla V(\vec{r}) = -\rho(\vec{r})$

V – potential (electric/gravitational)
ρ – source of field
(electric charge/ graviatational mass)

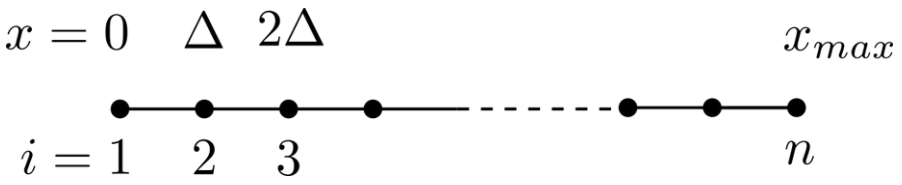
- we consider one-dimensional case with Dirichlet boundary conditions

$$\frac{d^2V}{dx^2} = -\rho(x), \quad x \in [0, x_{max}]$$

$$V(0) = V_L, \quad V(x_{max}) = V_R$$

Because FDM will be used, which defines derivative for a set of distinct points homogenously located in 1D space, we define the mesh of points

$$x = \Delta \cdot (i-1) = x_i, \quad \Delta = \frac{x_{max}}{n-1}, \quad i = 1, \ldots, n$$



Now we may discretize Poisson equation by rewriting it only for the mesh points approximating the derivative with finite difference formula, in the following we use abbreviations

$$x = x_i, \qquad V(x) = V(x_i) = V_i, \qquad \rho(x) = \rho(x_i) = \rho_i$$

$$\frac{d^2V}{dx^2} = \frac{V_{i+1} - 2V_i + V_{i-1}}{\Delta^2} \quad (+O(\Delta^2))$$

$$\frac{d^2V}{dx^2} = -\rho(x) \quad \Rightarrow \quad \frac{V_{i-1} - 2V_i + V_{i+1}}{\Delta^2} = -\rho_i$$

Lets multiply last expresson by Δ^2 and write it for each mesh point, starting from the 1-st

$$\begin{array}{rrrrrcl}
\cancel{V_0} & -2V_1 & +V_2 & & & = & -\rho_1\Delta^2 \\
V_1 & -2V_2 & +V_3 & & & = & -\rho_2\Delta^2 \\
 & V_2 & -2V_3 & +V_4 & & = & -\rho_2\Delta^2 \\
 & & \ddots & \ddots & \ddots & = & \vdots \\
 & & V_{n-1} & -2V_n & +\cancel{V_{n+1}} & = & -\rho_n\Delta^2
\end{array}$$

The points V_0 and V_{n+1} do not belong to mesh points and must be skipped.

We get SLE which comfortably may be defined in matrix form

$$\begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & \cdots & \cdots & \cdots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ \vdots \\ V_{n-1} \\ V_n \end{bmatrix} = \begin{bmatrix} -\rho_1\Delta^2 \\ -\rho_2\Delta^2 \\ -\rho_3\Delta^2 \\ \vdots \\ -\rho_{n-1}\Delta^2 \\ -\rho_n\Delta^2 \end{bmatrix}$$

Finally we need the boundary condition (BC) to be embedded into SLE, these are given for V_1 and V_n i.e. these rows (equations) shall be changed in such a way to fullfill BC. This can be simpy done in firstly, canceling all elements in these rows, secondly make diagonal elements equal one, and third, put in the right vector required values of potentials V_1 and V_n.

$$\begin{bmatrix} 1 & 0 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & \cdots & \cdots & \cdots & \\ & & 1 & -2 & 1 \\ & & & 0 & 1 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ \vdots \\ V_{n-1} \\ V_n \end{bmatrix} = \begin{bmatrix} V_L \\ -\rho_2\Delta^2 \\ -\rho_3\Delta^2 \\ \vdots \\ -\rho_{n-1}\Delta^2 \\ V_R \end{bmatrix}$$

The final SLE matrix has tridiagonal nonsymmetric form.
Muliplying V-vector by 1-st or n-th row we get V_L or V_R boundary values.

Example 2: use of SLE to find the coefficients of polynomial shaping function

Low-degree shaping polyniomials are used in interpolation of curves, surfaces or to solve ODE/PDE with the finite element methods. This popularity results from severals factors, they are easily differentated, integrated and last but not least shape of polynomial can be modelled according to some local requirements.
Lets consider the last property.

First, we introduce set of collocation points at which required properties will be defined

$$\begin{cases} x_0: & f(x_0), & f'(x_0) \\ x_1: & f(x_1), & \\ x_2: & f(x_2), & \\ x_3: & f(x_3), & f'(x_3) \end{cases}$$

$$f'(x_k) = \left.\frac{df}{dx}\right|_{x=x_k}$$

There are 4 points but 6 conditions. The unique polynomial which fullfills these condition is 5-th order because it has six free parameters

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

hence we are looking for coefficients vector

$$\vec{a} = [a_0, a_1, a_2, a_3, a_4, a_5] = ?$$

for the monomial basis functions

$$\{x^i : i = 0, \ldots, 5\} = \{1, x, x^2, x^3, x^4, x^5\}$$

Lets assume following conditions:

$$
\begin{aligned}
x_0 = 0: &\quad f(x_0) = 0, \quad f'(x_0) = 1 \\
x_1 = 1: &\quad f(x_1) = 1 \\
x_2 = 2: &\quad f(x_2) = 0 \\
x_3 = 3: &\quad f(x_3) = -1, \quad f'(x_3) = 0
\end{aligned}
$$

How to construct SLE? And how it looks like?

One may notice that expression defining the polynomial is linear with respect to its coefficients.
By writing down set of conditions for this general form of polynomial we get the system of linear equations

$$
\begin{aligned}
1 \cdot a_0 + \quad & x_0 \cdot a_1 + \quad x_0^2 \cdot a_2 + \quad x_0^3 \cdot a_3 + \quad x_0^4 \cdot a_4 + \quad x_0^5 \cdot a_5 = \quad 0 \quad (\text{value}) \\
0 \cdot a_0 + \quad & 1 \cdot a_1 + \quad 2x_0^1 \cdot a_2 + \quad 3x_0^2 \cdot a_3 + \quad 4x_0^3 \cdot a_4 + \quad 5x_0^4 \cdot a_5 = \quad 1 \quad (\text{derivative}) \\
1 \cdot a_0 + \quad & x_1 \cdot a_1 + \quad x_1^2 \cdot a_2 + \quad x_1^3 \cdot a_3 + \quad x_1^4 \cdot a_4 + \quad x_1^5 \cdot a_5 = \quad 1 \quad (\text{value}) \\
1 \cdot a_0 + \quad & x_2 \cdot a_1 + \quad x_2^2 \cdot a_2 + \quad x_2^3 \cdot a_3 + \quad x_2^4 \cdot a_4 + \quad x_2^5 \cdot a_5 = \quad 0 \quad (\text{value}) \\
1 \cdot a_0 + \quad & x_3 \cdot a_1 + \quad x_3^2 \cdot a_2 + \quad x_3^3 \cdot a_3 + \quad x_3^4 \cdot a_4 + \quad x_3^5 \cdot a_5 = \quad -1 \quad (\text{value}) \\
0 \cdot a_0 + \quad & 1 \cdot a_1 + \quad 2x_3^1 \cdot a_2 + \quad 3x_3^2 \cdot a_3 + \quad 4x_3^3 \cdot a_4 + \quad 5x_3^4 \cdot a_5 = \quad 0 \quad (\text{derivative})
\end{aligned}
$$

SLE in matrix form: $A\vec{a} = \vec{b}$

$$
\begin{bmatrix}
1 & x_0^1 & x_0^2 & x_0^3 & x_0^4 & x_0^5 \\
0 & x_0 & 2x_0^1 & 3x_0^2 & 4x_0^3 & 5x_0^4 \\
1 & x_1^1 & x_1^2 & x_1^3 & x_1^4 & x_1^5 \\
1 & x_2^1 & x_2^2 & x_2^3 & x_2^4 & x_2^5 \\
1 & x_3^1 & x_3^2 & x_3^3 & x_3^4 & x_3^5 \\
0 & x_3 & 2x_3^1 & 3x_3^2 & 4x_3^3 & 5x_3^4
\end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix}
=
\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ -1 \\ 0 \end{bmatrix}
$$

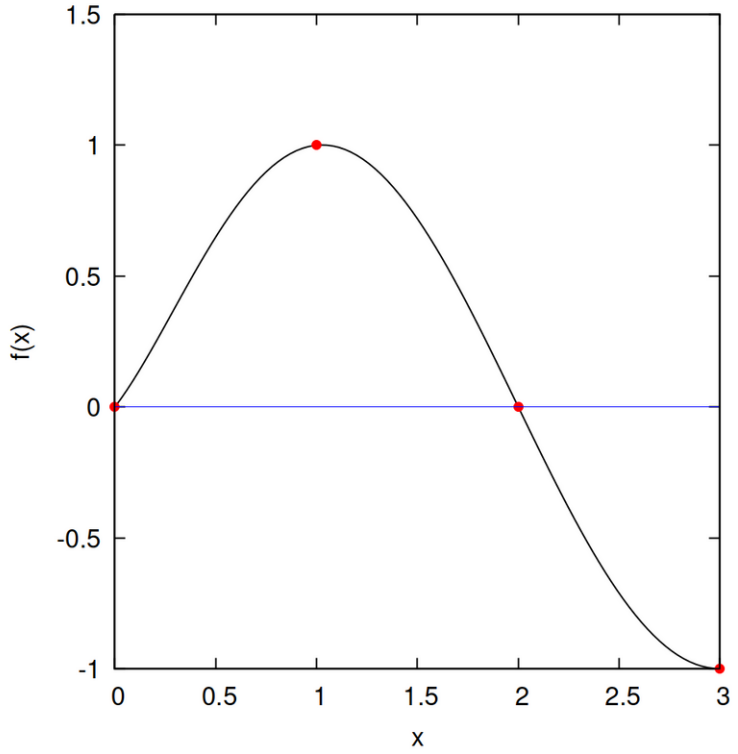
In order to solve this problem the LU factorization must be used because the matirx is not symmetric.

assumed conditions

$$
\begin{aligned}
x_0 = 0: &\quad f(x_0) = 0, \quad f'(x_0) = 1 \\
x_0 = 1: &\quad f(x_1) = 1 \\
x_0 = 2: &\quad f(x_2) = 0 \\
x_0 = 3: &\quad f(x_3) = -1, \quad f'(x_3) = 0
\end{aligned}
$$

solution

$$f(x) = 1\,x + 1.5\,x^2 - 2.1481\,x^3 + 0.7222\,x^4 - 0.0741\,x^5$$



at collocation points, polynomial **exactly** fullfills conditions,
but between them it smoothly adapts

Remark: coefficient $a_0=0$ vanishes due to first condition $f(x_0)=0$,
it means that monomial x^0 is not needed in monomial basis

Gram-Schmidt orthogonalization

Our task:
transform linearly independent vector basis into
the basis containing orthogonal and normalized vectors

$$\{\vec{u}_1,\, \vec{u}_2\} \Longrightarrow \{\vec{q}_1,\, \vec{q}_2\}$$

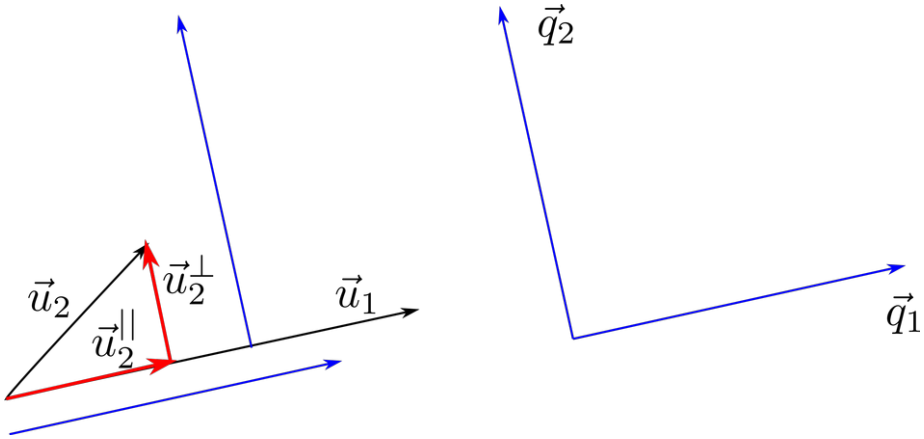
$$\vec{u}_i^{\,T}\vec{u}_j \neq 0 \qquad \vec{q}_i^{\,T}\vec{q}_j = \delta_{i,j}$$

($\delta_{i,j}$-Kroenecker delta)

- first, we must set the direction of
  the first vector q_1, usually is the same as u_1
  (additionally q_1 is normalized)

$$\vec{q}_1 = \frac{\vec{u}_1}{\|\vec{u}_1\|}$$

Graphical interpretation of **basis transformation**
of two-vector basis



- from figure we infer the following relation

$$\vec{u}_2 = \vec{u}_2^{\perp} + \vec{u}_2^{\parallel}$$

$$\vec{u}_2 = \vec{u}_2^{\perp} + r\vec{q}_1, \quad (r \in R) \qquad \vec{q}_1^{\,T} \cdot /$$

$$\vec{q}_1^{\,T}\vec{u}_2 = \underbrace{\vec{q}_1^{\,T}\vec{u}_2^{\perp}}_{=0} + r\,\underbrace{\vec{q}_1^{\,T}\vec{q}_1}_{=1} \quad \Longrightarrow \quad r = \vec{q}_1^{\,T}\vec{u}_2$$

$$\vec{u}_2^{\perp} = \vec{u}_2 - \vec{q}_1\left(\vec{q}_1^{\,T}\vec{u}_2\right)$$

$$\vec{q}_2 = \frac{\vec{u}_2^{\perp}}{\|\vec{u}_2^{\perp}\|}$$

QR factorization

Any matrix $A \in C^{m \times n}$, $m \geq n$ with linearly independent columns spans n-dimensional subspace, i.e. the set of column-vectors forms the basis which can be orthogonalized with Gram-Schmidt method, by retaining also the information about the scalar products we get the new so called **QR factorization**

remark: QR factorization applies to both, square and rectangular matrices

$$A = QR$$

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} = \left[\begin{array}{c|c|c|c} q_{11} & q_{12} & \dots & q_{1n} \\ q_{21} & q_{22} & \dots & q_{2n} \\ q_{31} & q_{32} & \dots & q_{3n} \\ \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \dots & \vdots \\ q_{m1} & q_{m2} & \dots & q_{mn} \end{array}\right] \begin{bmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ 0 & r_{22} & \dots & r_{2n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & r_{nn} \end{bmatrix}$$

- Q is an orthogonal matrix while R is an upper triangular matrix retaining the scalar products

$$\vec{q}_k = \vec{a}_k - \sum_{i=1}^{k-1} \vec{q}_i \, r_{ik}$$

$$r_{ik} = \frac{\vec{q}_i^{\,T} \vec{a}_k}{d_i}, \quad i = 1, 2, \dots, k-1$$

$$d_i = \vec{q}_i^{\,T} \vec{q}_i$$

$$r_{ii} = 1$$

q-vectors are not normalized

$$Q^T Q = D = diag(d_1, d_2, \dots, d_n) \neq I$$

Least square problem

We know how to construct usual SLE i.e. defined by square matrix → solution SLE: LU, LL^T, LDL^T factorization

However, if we add to standard SLE a set of additional conditions in form of equations,
we get SLE with number of equations greater than number of unknows.
It has ractangular matrix and is called **overdetermined SLE**

$$m > n$$

$$A \in R^{m \times n}$$

$$\vec{x} \in R^n$$

$$\vec{b} \in R^m$$

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \dots \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ \dots \\ \dots \\ b_m \end{bmatrix}$$

Immediately we may ask two important questions

Q1: does any solution of such SLE exists?

Q2: how to find the solution?

A1: In most cases there is no exact solution of SLE, all we can do is looking for „the best approximated solution”

A2: we can find the approximated solution by applying the QR factorization

If we find **any** solution of overdetermined SLE (by any means) we can always check it correctness by calculating the residual vector and its Euclidean (p=2) norm

$$\vec{r} = \vec{b} - A\vec{x}$$

$$||\vec{r}||_2 = (\vec{r}^T\vec{r})^{1/2}$$

Obviously we are looking for the solution which minimizes the norm of residual vector, because this norm is non-negative we may analyze its **square** as well

$$\text{best } \vec{x} \quad \Longrightarrow \quad \vec{r}^T\vec{r} = \min$$

This kind of numerical task is called **the least square problem**.

Because we seek the minimum of square of residual vector which depends on all elements of x vector, we shall calculate the gradient of the square and from equaling it to zero-valued vector get conditions for the solution.

- first, lets define n-dimensional quadratic function for the least square problem

$$F(\vec{x}) = \vec{r}^{\,T}\vec{r}$$

- second, find its explicit dependence on the solution x

$$\begin{aligned} F(\vec{x}) &= (\vec{b} - A\vec{x})^{\,T}(\vec{b} - A\vec{x}) \\ &= (\vec{b}^{\,T} - \vec{x}^{\,T}A^{T})(\vec{b} - A\vec{x}) \\ &= \vec{b}^{\,T}\vec{b} - \vec{b}^{\,T}A\vec{x} - \vec{x}^{\,T}A^{\,T}\vec{b} + \vec{x}^{\,T}A^{\,T}A\vec{x} \end{aligned}$$

- third, calculate the gradient of F to localize the minimum

$$\nabla_{\vec{x}} F = \frac{\partial F}{\partial \vec{x}} = \vec{0}$$

$$\frac{\partial F}{\partial \vec{x}} = -\vec{b}^{\,T}A - A^{T}\vec{b} + A^{T}A\vec{x} + \vec{x}^{\,T}A^{T}A = \vec{0}$$

$$A^{T}A\vec{x} - A^{T}\vec{b} = 0$$

$$A^{T}A\vec{x} = A^{T}\vec{b} \qquad (A^{T}A)^{-1} \cdot /$$

$$\vec{x} = (A^{T}A)^{-1}A^{T}\vec{b}$$

the task defined as least square problem has solution and we know how to find it

$$\begin{aligned} \left(\boldsymbol{b}^{T}A\right)_j &= \sum_i b_i\, a_{i,j} \\ &= \sum_i a_{j,i}\, b_i \\ &= \left(A^{T}\boldsymbol{b}\right)_j \end{aligned}$$

QR factorization to solve least square problem

- solution of LSP can be found in standard way i.e. by first, multiplying both sides SLE by A^T and then solving transformed SLE by means LL^T factorization (**A^TA is square symmetric/hermitean matrix**),
- however, multiplication of two matrices would introduce additional rounding errors to the solution due to much larger condition number and for this reason should be avoided
- for rectangular matrix we still can use **QR factorization** and exploit properties of Q and R matrices

$$A\vec{x} = \vec{b}$$

get rid of Q → $QR\vec{x} = \vec{b} \qquad Q^T \cdot / \qquad (Q^TQ = D)$

get rid of D → $DR\vec{x} = Q^T\vec{b} \qquad D^{-1} \cdot /$

$$D = diag\{d_1, d_2, \ldots, d_n\}$$

$$R\vec{x} = D^{-1}Q^T\vec{b} = \vec{y}$$

$$D^{-1} = \left\{\frac{1}{d_1}, \frac{1}{d_2}, \ldots, \frac{1}{d_n}\right\}$$

backward substitution → $R\vec{x} = \vec{y}$

Remark: (i) QR factorization is very stable due to orthogonalization,

(ii) QR factorization we can use for solving standard SLE with square matrix but such attempt has drawbacks:

- QR needs more space reserved in memory for both matrices
- finding QR factorization requires more time than other methods (not economical if not needed)

Linear algebra numerical packages

BLAS - **B**asic **L**inear **A**lgebra **S**ubprograms, C/C++/Fortran and other, freely-available highly optimized collection of 3-level subroutines for basic

- level 1: scalar, vector, vector-vector operations
- level 2: matrix-vector operations
- level 3: matrix-matrix operations

LAPACK – **L**inear **A**lgebra **PAC**Kage, C/C++/Fortran and other, freely-available highly-optimized (based on BLAS subroutines) software for advanced tasks like e.g.

- solving systems of linear equations
- solving least-squares problem
- solving eigenvalue problems

- matrix factorizations: LU, LL^T, LDL^T, QR, SVD, Schur

- both are available at: www.netib.org
- each Linux-based systems can install them (and already has) from its own package-repository since they are required by many applications to work
- numerical routines are provided for single/double precision & real/complex numbers
- BLAS and LAPACK are provided with other freely-available packages for e.g. **Math Kernel Library** (Intel), **GNU Scientific Library**
  (these are used at computer classes)

Example CBLAS-1 routines (Intel MKL)

- ordinary dot-product of real vectors (double) $c = \vec{x}^T \vec{y}$

*double cblas_ddot (const MKL_INT n, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy);*

- dot-product in Hibert space (double complex) $c = \vec{x}^H \vec{y} = \left(\vec{x}^T\right)^* \vec{y}$

*void cblas_ddotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotc);*

Layout – indicate on row-wise storage or column-wise storage of matrix elements

Example CBLAS-2 routines (Intel MKL)

- general matrix-vector multiplication (double) $\vec{y} = \alpha A \vec{x} + \beta \vec{y}, \qquad \alpha, \beta \in R$

*void cblas_dgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);*

Example CBLAS-3 routines (Intel MKL)

- matrix-matrix- product with general (non-symmetric) matrices (double)

$$C = \alpha\, op(A)\, op(B) + \beta C, \qquad \alpha, \beta \in R$$
$$op(A) = A, \quad op(A) = A^T, \quad op(A) = A^H$$

*void cblas_dgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);*

Example of LAPACK routines (taken from Intel MKL)

- LU factorization (double) $A = LU$

```
lapack_int LAPACKE_dgetrf (int matrix_layout , lapack_int m , lapack_int n ,
                           lapack_complex_double * a , lapack_int lda , lapack_int * ipiv );
```

- Cholesky factorization (double) $A = LL^T$

```
lapack_int LAPACKE_dpotrf (int matrix_layout , char uplo , lapack_int n ,
                           lapack_complex_double * a , lapack_int lda );
```

- solving SLE using LU factorization (zegtrf has to be called before) for many right-hand sides (double) $AX = B$

```
lapack_int LAPACKE_dgetrs (int matrix_layout , char trans , lapack_int n , lapack_int nrhs ,
    const double * a ,lapack_int lda , const lapack_int * ipiv , double * b , lapack_int ldb );
```

- solving SLE using Cholesky factorization (dpotrf has to be called before) for many right-hand sides (double) $AX = B$

```
lapack_int LAPACKE_dpotrs (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs ,
                           const double * a ,lapack_int lda , double * b , lapack_int ldb );
```

- matrix inversion (dgetrf has to be called before) (double) $AA^{-1} = I$

```
lapack_int LAPACKE_dgetri (int matrix_layout , lapack_int n , double * a , lapack_int lda ,
                           const lapack_int * ipiv );
```

- QR factorization (double) – matrix Q is not formed explicitly

$$A = QR$$

```
lapack_int LAPACKE_dgeqrf (int matrix_layout, lapack_int m, lapack_int n, double* a,
                           lapack_int lda, double* tau);
```

- decoding explicit form of Q matrix passed from dgeqrf (double)

```
lapack_int LAPACKE_dorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);
```

- solving overdetermined SLE – linear least-squares problem for many right-hand sides (double)
  it is a DRIVER – it calls QR routine itself

$$AX = B$$

```
lapack_int LAPACKE_dgels (int matrix_layout, char trans, lapack_int m, lapack_int n,
                          lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb);
```