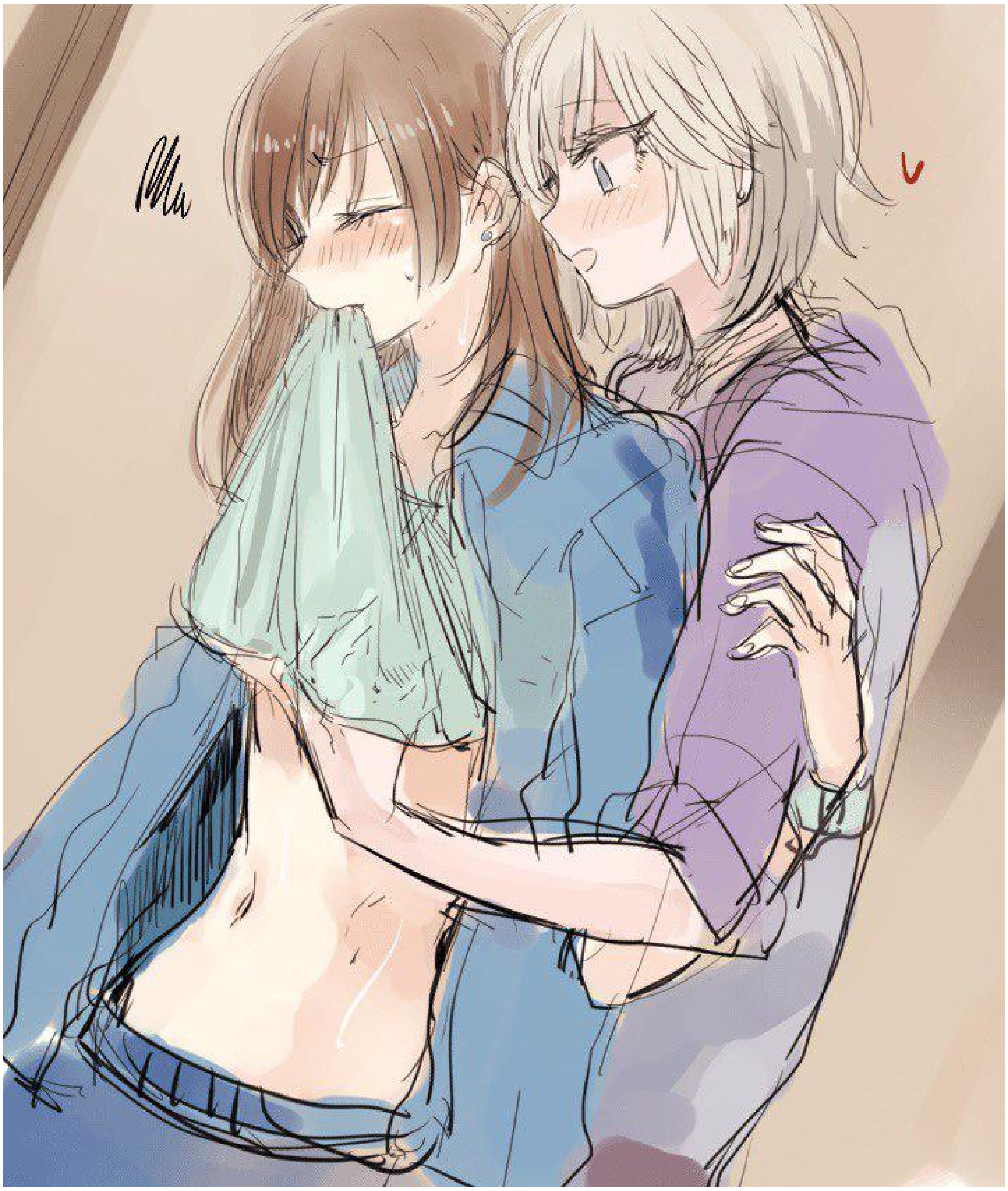
Random questions for NLA exam

9-10 December 2019



1. Representation of numbers in computer. Fixed and floating point arithmetics. Loss of significance.

Information in computer is stored in bits. Each bit saves either value of zero or one. There are several ways to represent a real number in computer, we will talk about *fixed-point* and *floating-point* arithmetics.

- *Fixed-point representation* Fixed-point representation, otherwise called **Qm.n** format, takes the fixed amount of bits for magnitude (**integer**) and **fractional** parts, to be more precise, m and n bits in each case (that is where the name comes from).

  One bit in the beginning is reserved for sign of a number, so in total we need *m + n + 1* bits to represent number this way. Hence, the range of numbers we can represent this way is $[-(2^m), 2^m - 2^{-n}]$ with *resolution* (fractional part accuracy) of 2^{-n}.

  Because of its fixed range and resolution, this kind of representation possesses **absolute** accuracy (you can represent any number in given range and resolution). Also, you should be aware of overflow when sum of two numbers is out of given range.

  Example: _Fract in C/C++

- *Floating-point representation* This format uses the representation of number in following way:

$$num = sign \times significand \times base^{exponent}$$

  Sign is either zero or one (as in fixed-point case), significand and base are positive integers, and exponent is either positive or negative depending on whether the number had fractional part.

  Note that in this case the accuracy is *relative* (varies depending on the scale of a number) due to its exponential nature. This can cause the problem called *loss of significance*: the components that does not fit the significand bits are truncated. Be aware while substracting numbers with large fractinal part.

  Example: *float32* in C: 1 bit for sign, 8 bits for exponent, 23 bits for significand.

2. Vector norms. Forward and backward stability. Disks in different norms. First norm and compressed sensing.

Vector norm is a nonnegative-valued scalar function from vector space V $V \to [0, +\infty)$ which satisfies the following properties:

1. $\|\alpha x\| = |\alpha| \, \|x\|$ (*absolutely scalable*)
2. $\|x + y\| \leq \|x\| + \|y\|$ (*triangle inequality*)
3. $\|x\| = 0 \Leftrightarrow x = 0$ (*positive semidefinite*)

Interpretation: *norm is qualitive measure of smallness of a vector*
The distance between two vectors can be given in terms of a vector norm:

$$d(x, y) = \|x - y\|$$

Examples of vector norms

p-norm: $\|x\|_p = (\sum_{i=1}^{n} |x_i|^p)^{1/p}$

Its important subclasses:

Manhattan distance (L_1): $\|x\|_1 = \sum_{i=1}^{n} |x_i|$

Euclidean norm (L_2): $\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$

Chebyshev norm (L_∞): $\|x\|_\infty = \max_i |x_i|$

All (vector) norms are equivalent in a sense that

$$C_1 \|x\|_* \leq \|x\|_{**} \leq C_2 \|x\|_*$$

Unit disk of *-norm is a set of points such that

$$\|x\|_* \leq 1$$

Here are examples of unit disks for different norms:

- L_1-norm: rhombus
- L_2-norm: circle
- L_∞-norm: square

0.1 Compressed sensing.

Consider we want to minimize the norm of solution of underdetermined linear system:

$$\|x\| \to \min_x$$
$$Ax = f$$

In case of euclidean norm, this is typical least squares problem. If we minimize this, nearly no coefficients will go to zero. But using L_1-norm sparsifies the solution, as many absolute values will go to zero. This is called **compressed sensing**.

0.2 Stability of algorithms.

Suppose we want to find **numerical algorithm** $a(x)$ that approximates given function $f(x)$.

- This algorithm is called **forward stable** if

$$\|a(x) - f(x)\| \leq \epsilon, \epsilon \to 0$$

- The algorithm is called **backward stable** if for any x there exits vector $x + \delta x$ such that:

$$a(x) = f(x + \delta x), \|\delta x\| \to 0$$

4. Matrix norms. Example of a norm that is not a matrix norm (does not satisfy submultiplicative property). Scalar product. Cauchy-Schwarz-Bunyakovsky inequality.

Matrix norm is a vector norm of the vector space of $n \times m$ matrices that satisfies following conditions:

1. $\|\alpha A\| = |\alpha| \, \|A\|$
2. $\|A + B\| \leq \|A\| + \|B\|$
3. $\|A\| \geq 0, \|A\| = 0 \Leftrightarrow A = 0$

Additionaly matrix norm can be **submultiplicative**

$$\|AB\| \leq \|A\| \, \|B\|$$

Example of non-submultiplicative norm: Chebyshev norm

$$\|A\|_C = \max_{i,j} |a_{ij}|$$

$$\max_{i,j} |\sum_k^n a_{ik} b_{kj}| \geq \max_{i,j} |a_{ij}| \max_{i,j} |b_{ij}|$$

General examples of norms:

Frobenius norm: $\|A\|_F = \sqrt{\sum_i^n \sum_j^m |a_{ij}|^2}$

Operator norms: $\|A\|_{*,**} = sup_{x \neq 0} \frac{\|Ax\|_*}{\|x\|_{**}}$

- $\|A\|_{1,1} = \max_j \sum_i^n |a_{ij}|$
- $\|A\|_{2,2} = \sqrt{\lambda_{max}(A^* A)}$ (*spectral norm*)
- $\|A\|_{\infty,\infty} = \max_i \sum_j^m |a_{ij}|$

0.3 Scalar product. Cauchy-Schwarz-Bunyakovski inequality

- Scalar product for vectors:

$$(x, y) = x^* y = \sum_i^n \overline{x}_i y_i$$

It is true that:

$$\|x\|_2 = \sqrt{(x, x)}$$

(Euclidean norm is induced by scalar product)

- (Frobenius) Scalar product for matrices:

$$(A, B)_F = \sum_i^n \sum_j^m \overline{a_{ij}} b_{ij} = trace(A^* B)$$

$$\|A\|_F = \sqrt{(A, A)_F}$$

The **angle** between vectors (and matrices in the same fashion) can be defined as

$$cos(x, y) = \frac{(x, y)}{\|x\|_2 \|y\|_2}$$

Cosinus naturally lies in boundaries of [-1, 1], to guarantee that the important **Cauchy-Schwarz-Bunyakovski inequality** can be used

$$|(x, y)| \leq \|x\|_2 \|y\|_2$$

5. Unitary matrices and their properties. Examples: Fourier matrix, permutation matrix, Householder reflections, Givens rotations.

Square matrix $U \in \mathbf{C}^{n \times n}$ is called unitary if

$$U^* U = U U^* = I_n$$

Or equivalently, columns and rows of the matrix form orthonormal basis in $\mathbf{C}^n$
In case of real matrices we can call them **orthogonal**.

Some properties of Unitary matrices:

1. Product of two unitary matrices is a unitary matrix:

$$(UV)^* UV = V^* (U^* U) V = V^* V = I$$

2. Unitary matrices preserve spectral and Frobenius norm:

$$\|UAV\|_2 = \|A\|_2$$

$$\|UAV\|_F = \|A\|_F$$

0.4 Examples of unitary matrices

1. Permutation matrix P whose rows (columns) are permutation of rows (columns) of the identity matrix

$$P^* = P^T$$

$$P^T P = P P^T = P' = \begin{cases} p'_{ij} = 1, i = j, \\ p'_{ij} = 0, i \neq j \end{cases} = I$$

2. Discrete Fourier Transform (DFT) matrix

$$F_n = \frac{1}{\sqrt{n}} e^{-i \frac{2\pi kl}{n}} {}_{k,l=0}^{n-1}$$

3. Householder matrix

$$H(v) = I - 2vv^*$$

$$H^*H = (I-2vv^*)^*(I-2vv^*) = (I-2v^*v)(I-2vv^*) = I - 4vv^* + 4vv^*vv^* = I - 4vv^* + 4vv^* = I = HH^*$$

4. Givens rotation

$$G = \begin{pmatrix} \frac{x_i}{\sqrt{x_i^2+x_j^2}} & \frac{x_j}{\sqrt{x_i^2+x_j^2}} \\ -\frac{x_j}{\sqrt{x_i^2+x_j^2}} & \frac{x_i}{\sqrt{x_i^2+x_j^2}} \end{pmatrix}$$

19. Divide and conquer algorithm for symmetric eigenvalue problems (with derivation)

We know that SVD exists for any matrix. Thinking about it, we made our matrix diagonal using two-sided unitary transofmations. It is true that using two-sided Householder transormations we can in the same fashion acquire **bi-diagonal form B**.

Now we can acquire tridiagonal T from B:

$$T = B^*B$$

This is how SVD is transformed into **symmetric eigenvalue problem**. Our task is to find eigenvalues of T.

Our tridiagonal matrix can be presented in block form:

$$T = \begin{bmatrix} T_1' & \beta \\ \beta^T & T_2' \end{bmatrix}$$

$$\beta = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \beta_m & 0 & \cdots & 0 \end{pmatrix}$$

It can be rewritten as

$$T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \beta_m vv^*$$

$$v = (0, \cdots 1, 1 \cdots, 0)^T$$

Where v has nonzeros only on $n/2$ and $n/2+1$; $rank(vv^*) = 1$. Why is that? I will try to explain according to maestro (`https://youtu.be/L-VF7u0W0Ys?t=1400`): we extract for the matrix the part on the edge of blocks which looks like

$$\begin{pmatrix} \beta_m & \beta_m \\ \beta_m & \beta_m \end{pmatrix}$$

This obviously has rank of one. If we write what we have in vector form, we will have exactly $\beta_m vv^*$.

Then if we found eigen decomposition for T_1 and T_2

$$T_1 = Q_1\Lambda_1 Q_1^*$$

$$T_1 = Q_2\Lambda_1 Q_2^*$$

In block form we get the following

$$\begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \beta_m vv^* = \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} (D + \rho uu^*) \begin{bmatrix} Q_1^* & 0 \\ 0 & Q_2^* \end{bmatrix}$$

$$\begin{bmatrix} Q_1^* & 0 \\ 0 & Q_2^* \end{bmatrix} T \begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} = D + \rho uu^*$$

$$D = \begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix}$$

Why is that? Because unitarity does not mess up eigenvalues, so we preserve D plus matrix of rank one ρuu^*. Our task is exactly to find eigenvalues of $D + \rho uu^*$.

If λ is an eigenvalue, we have

$$(D + \rho uu^*)x = \lambda x$$
$$(D - \lambda I)x + \rho uu^* x = 0$$
$$x + \rho(D - \lambda I)^{-1} uu^* x = 0$$
$$u^* x + \rho u^*(D - \lambda I)^{-1} uu^* x = 0$$

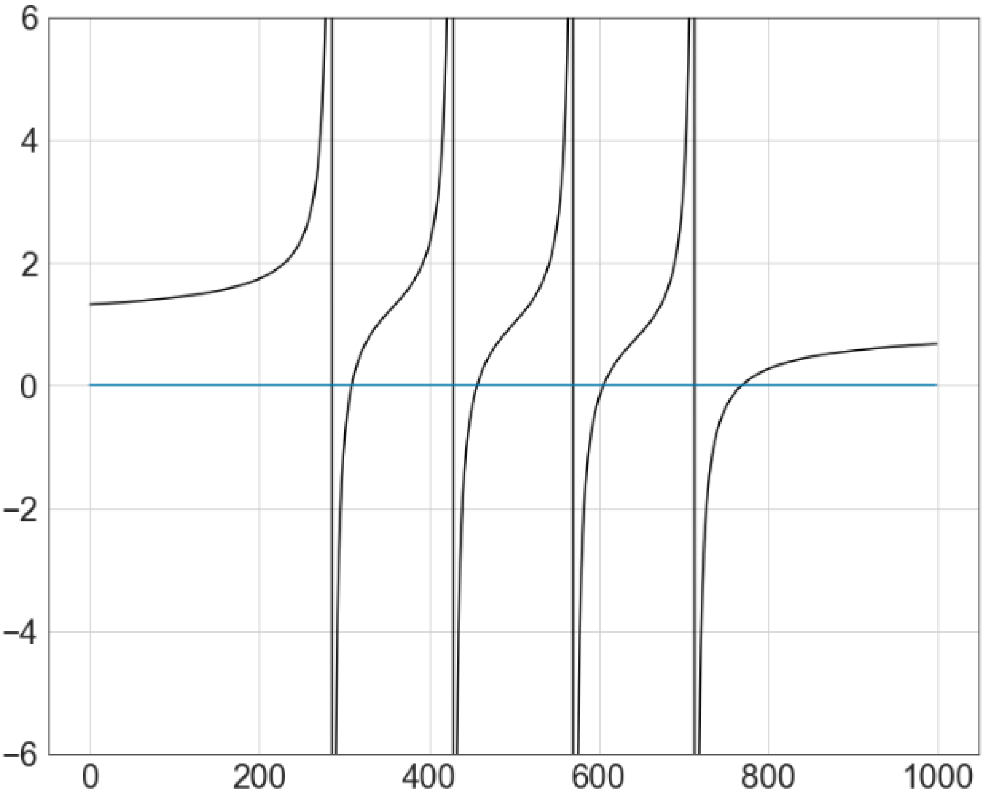
Note that $x \neq 0$ is nonzero by def. of eigenvector, $u \neq 0$ because this is how we defined it. Suppose $u^* x = 0$. But if that was the case, x is the eigvector of our matrix, D is diagonal, so x would contain only one nonzero value. Thus we contradict ourselves and scalar product with one-element-nonzero vector cannot be zero. Hooray, we can divide both parts by $u^* x$. we get

$$1 + \rho u^*(D - \lambda I)^{-1} u = 0$$
$$1 + \rho(u, (D - \lambda I)^{-1} u) = 0$$

Then in vector form we have (characteristic) equation

$$1 + \rho \sum_i^n \frac{|u_i|^2}{d_i - \lambda} = 0$$

How to solve it numerically? Approximate by hyperbola



$$f(\lambda) = c_0 + \frac{c_1}{d_i - \lambda} + \frac{c_2}{d_{i+1} - \lambda}$$

For each pair of ds $[d_i, d_{i+1}$

20. Jacobi method for eigenvalue problem. Its convergence (with proof).

The idea of Jacobi method is to use Givens rotation succesfully

$$G = \begin{pmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{pmatrix}$$

to minimize sum of squares of diagonal elements

$$\Gamma(A) = \text{off}(U^* A U), \quad \text{off}^2(X) = \sum_{i \neq j} |X_{ij}|^2 = \|X\|_F^2 - \sum_{i=1}^n x_{ii}^2$$

Let us proof the convergence. First, show that

$$\text{off}(B) < \text{off}(A)$$

Where $B = U^* A U$.

$$\Gamma^2(A) = \text{off}^2(B) = \|B\|_F^2 - \sum_{i=1}^{n} b_{ii}^2 =$$

Because unitary matrices preserve Frobenius norm, $\|B\|_F = \|A\|_F$. Suppose we performed Givens rotation on indices p, q.

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a_{pp} & a_{pq} \\ a_{pq} & a_{qq} \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} = \begin{pmatrix} b_{pp} & 0 \\ 0 & b_{qq} \end{pmatrix}$$

Then because only rows and columns p and q of A are modified in B, we have

$$\|A\|_F^2 - \sum_{i \neq p,q} b_{ii}^2 - (b_{pp}^2 + b_{qq}^2) = \|A\|_F^2 - \sum_{i \neq p,q} a_{ii}^2 - (a_{pp}^2 + 2a_{pq}^2 + a_{qq}^2)$$

Now because a_{pp} and a_{qq} are the ones under rotation, we get

$$= \|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2 - 2a_{pq}^2 = \text{off}^2(A) - 2a_{pq}^2 < \text{off}^2(A)$$

If we select γ as the largest off-diagonal element

$$|a_{ij}| \leq \gamma$$

Let $N = n(n-1)/2$ be the number of non diagonal elements. From the sum above beacuse γ is the largest off-diagonal entry we get

$$\text{off}(A)^2 \leq 2N\gamma^2$$

By the lemma above

$$\text{off}(B)^2 = \text{off}(A)^2 - 2\gamma^2$$

So we get

$$\Gamma(A) = \text{off}(B)^2 \leq \sqrt{(1 - \frac{1}{N})\text{off}(A)}$$

Now suppose we had large number of steps k in Jacobi algorithm. From Calculus 101 we remember that if $k \to \infty$

$$\Gamma(A^{(K)}) \leq (1 - \frac{1}{N})^{k/2}\text{off}(A^{(k)} = e^{-1/2}\text{off}(A^{(k)}$$

21. Sparse matrix arithmetics: COO, LIL, CSR, CSC formats.

The simpliest format is to use **coordinate format (COO)** to represent the sparse matrix as positions and values of non-zero elements.

- i, j are array of integers
- val is the real array of nonzero elements

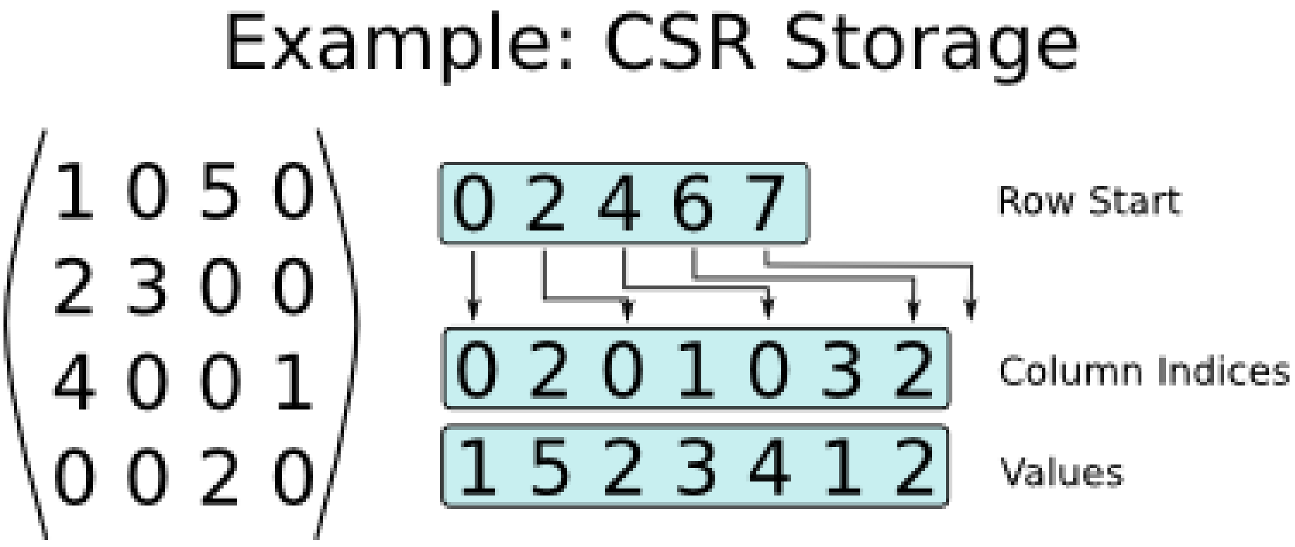
It stores $3 \cdot nnz$ elements.

- It is not optimal in storage
- It is not optimal for matrix-by-vector product
- It is not optimal for removing elements as you must make nnz operations to find one element

In the CSR (compressed sparse row) format, we use three different arrays to store matrix

1. ia stores row start and is an integer array of length $n + 1$
2. ja stores column indices and is an integer array of length nnz (non-zero elements)

3. *sa* stores the exact values and is an integer array of length nnz (non-zero elements)



In total, we need $2 * nnz + n + 1$ elements to store in memory.

- Sparse matrices give complexity reduction.
- But they are not very good for parallel/GPU implementation.
- They do not give maximal efficiency due to random data access.
- Typically, peak efficiency of $10\% - 15\%$ is considered good.

CSC (compressed sparse column) is just similar with **CSR**, we store column start and row indices.

LIL (list of lists) format stores one list per row, with each entry containing the column index and the value. Typically, these entries are kept sorted by column index for faster lookup.

22. Sparse LU decomposition, connection with graphs. Dependence of fill-in on ordering of graph nodes (with example). Nested dissection and spectral bisection algorithms.

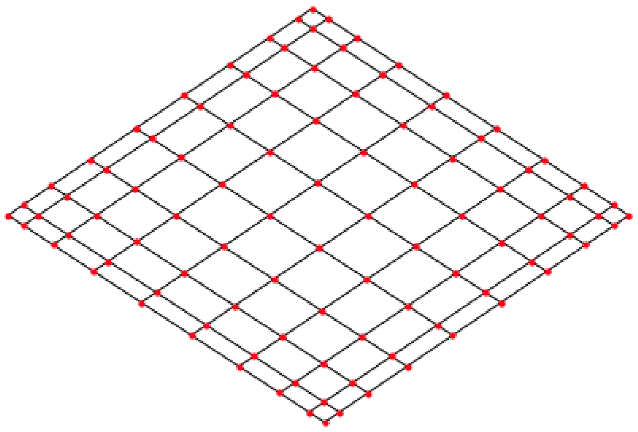
Recall LU decomposition

$$A = LU$$

In this case, the factors L and U can be sparse, for example, in case of tridiagonal matrix L and U are bidiagonal forms (recall algorithm from home task 2).

The number of nonzeros in LU decomposition has a deep connection to the graph theory. If we have sparse matrix, we can build graph such that if matrix element a_{ij} is nonzero, there exists an edge between i and j. Vice versa, every adjacency graph has (sparse) matrix (recall algo course). Illustration below.

```
import networkx as nx
n = 10
ex = np.ones(n);
lp1 = sp.sparse.spdiags(np.vstack((ex,  -2*ex, ex)), [-1, 0, 1], n, n, 'cs
r');
e = sp.sparse.eye(n)
A = sp.sparse.kron(lp1, e) + sp.sparse.kron(e, lp1)
A = csc_matrix(A)
G = nx.Graph(A)
nx.draw(G, pos=nx.spectral_layout(G), node_size=10)
```



Nested dissection. Split the graph into two with minimal number of vertices on the separator (set of vertices removed after we separate the graph into two distinct connected graphs). Complexity of the algorithm depends on the size of the graph separator.

23. Richardson iteration. Optimal choice of parameter. Convergence estimate.

Suppose we want to solve sparse linear system

$$Ax = b$$

The simpliest idea is to simple iterate using some parameter τ. This is called *Richardson iteration.*

$$\tau(Ax - b) = 0 \Rightarrow$$

$$x - \tau(Ax - b) = x \Rightarrow$$

1. Choose arbitary (nonzero!) x_0
2. $x_{k+1} = x_k - \tau(Ax_k - b)$

The optimal parameter τ is such that minimizes $\|I - \tau A\|$, so iteration converges as fast as possible. In case $A = A^*$ and positive definite:

$$\tau^* = \frac{2}{\lambda_{min} + \lambda_{max}}$$

30. Preconditioning concept. Right and left preconditioners. Jacobi, Gauss-Seidel and SOR preconditioners.

Here we introduce the concept of precondtitioning.
Suppose we have a linear system

$$Ax = f$$

Definition. Preconditioners For matrices P_R and P_L:

- if $P_L^{-1}A$ has better (smaller) *condition number* than A, it is called **left preconditioner**
- if AP_R^{-1} has smaller *condition number* than A, it is called **right preconditioner**

- if $P_L^{-1}AP_R^{-1}$, then A has both left and right preconditioner

or AP_L^{-1}

Then we can solve system easier

$$P_L^{-1}Ax = P_L^{-1}f$$
$$AP_R^{-1}y = f \Rightarrow P_Rx = y$$
$$P_L^{-1}AP_R^{-1}y = P_L^{-1}f \Rightarrow P_Rx = y$$

Now to the basic preconditioners

Jacobi method

The idea is that we express the diagonal element by sum of non-diagonals

$$a_{ii}x_i = -\sum_{i\neq j} a_{ij}x_j + f_i$$

Then perform *Richardson iteration* with $\tau = 1$, deriving x from above:

$$x_i^{(k+1)} = -\frac{1}{a_{ii}}(\sum_{i\neq j} a_{ij}x_j^{(k)} + f_i)$$

In the matrix form

$$x^{(k+1)} = D^{-1}((D - A)x^{(k)} + f)$$
$$x^{(k+1)} = x^{(k)} - D^{-1}(Ax^{(k)} - f)$$

We get D = diag(A) as **left preconditioner** (called **Jacobi predconditioner**).

Gauss-Seidel method

Modify Jacobi method by using last updates of previous components in addition to previous step

$$x_i^{(k+1)} = -\frac{1}{a_{ii}}(\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + \sum_{i\neq j} a_{ij}x_j^{(k)} - f_i)$$

In matrix form:

$$A = L + D + L^*$$

$$x^{(k+1)} = x^{(k)} - (L + D)^{-1}(Ax^{(k)} - f)$$

So we get **Gauss-Seidel preconditioner** $P = L + D$.

SOR preconditioner

Modify Gauss-Seidel preconditioner by adding parameter ω:

$$P = \frac{1}{\omega}(D + \omega L)$$

If the Jacobi method converges, then optimal ω is

$$\omega^* = \frac{2}{1 + \sqrt{1 - \rho_j^2}}$$

Otherwise there is no explicit formula.

31. Incomplete LU for preconditioning, ILU(τ), ILU(k), second-order ILU (ILU2)

Recall we used Gaussian elimination to count LU

$$A = P_1 L U P_2^T$$

Where P_1 and P_2 are pivots. We want to use sparse L, U but *fill-ins* will appear. The idea is to approximate LU by throwing away (ignore) new fill-ins

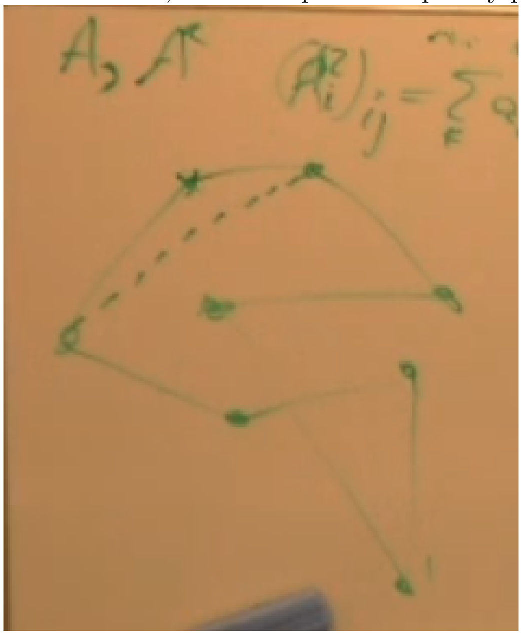
```
L = np.zeros((n, n))
    U = np.zeros((n, n))
    for k in range(n): #Eliminate one row
        L[k, k] = 1
        for i in range(k+1, n):
            L[i, k] = a[i, k] / a[k, k]
            for j in range(k+1, n):
                a[i, j] = a[i, j] - L[i, k] * a[k, j]  #New fill-ins appear
here
        for j in range(k, n):
            U[k, j] = a[k, j]
```

This is what called Incomplte LU (or, to be specific, ILU(0)).

You can also modify this method by not throwing all fill-ins, but only the ones which are smaller than the predefined threshold τ, or, as a variation, control the amount of stored nonzero elements. This is called **ILUT(τ)**.

Now let us describe the idea of **ILU(k)** method. It is true that for every $(n \times n)$ sparse matrix we can devise corresponding adjacency graph based on its value.

Now imagine we count LU decomposition with elimination. When we eliminate the variable, we get the system with $(n-1 \times n-1)$ variables. Look what is happening in graph. One vertex is out, and new edge can occur between vertexes which have common neighbours; in graph terminology, they are *second-order neighbours*. In a matrix sense, this corresponds to sparsity patterns of matrix A^2.



The purpose of **ILU(k)** is to leave only the values which correspond to neighbours of k-th order or lower; the most common and effective is to leave only first and second-order neighbours (**ILU(2)**).

The other effective solution is **second-order LU** for symmetric matrix A

$$A \approx U_2 U_2^T + U_2^T R_2 + R_2^T U_2$$

U_2 are upper triangular sparse matrixes, and R_2 is small with respect to some tolerance parameter

32. Fast Fourier Transform (FFT). Cooley-Tukey algorithm (with derivation).

The idea of Fast Fourier Transorm is to permute DFT matrix and simplify, so we get multiplication by smaller blocks and diagonal matrices.

First let us recall the usual DFT matrix.

$$F_n = \frac{1}{\sqrt{n}}[e^{-i\frac{2\pi}{n}kl}]_{k,l=0}^{n=1}$$

$$F_n = \frac{1}{\sqrt{n}} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ 1 & \omega^3 & \omega^6 & \cdots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

We now will describe **Cooley-Tukey Algorithm** in matrix form. Select such n for DFT that it is a power of 2. Suppose we have permutation matrix P_n such that

$$P_n F_n = \begin{pmatrix} \omega_n^{2kl} & \omega_n^{2k(n/2+l)} \\ \omega_n^{(2k+1)l} & \omega_n^{(2k+1)(n/2+l)} \end{pmatrix}$$

The first one is just DFT matrix for $n/2$. The other can be simplified by factoring the powers of the exponential in a fashion like this:

$$\omega_n^{2kl} = e^{-2kl\frac{2\pi i}{n}} = e^{-kl\frac{2\pi i}{n/2}} = \omega_{n/2}^{kl}$$

$$\begin{pmatrix} \omega_n^{2kl} & \omega_n^{2k(n/2+l)} \\ \omega_n^{(2k+1)l} & \omega_n^{(2k+1)(n/2+l)} \end{pmatrix} = \begin{pmatrix} F_{n/2} & F_{n/2} \\ F_{n/2}\Omega_{n/2} & -F_{n/2}\Omega_{n/2} \end{pmatrix}$$

Where

$$\Omega_{n/2} = diag(1, \omega_n, \omega_n^2, \cdots, \omega_n^{n/2-1})$$

We can write it as product

$$\begin{pmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{pmatrix} \begin{pmatrix} I_{n/2} & I_{n/2} \\ \Omega_{n/2} & -\Omega_{n/2} \end{pmatrix}$$

Here are the identity and diagonal matrices, for which storage and matmul is much cheaper.

33. Conituous and discrete convolution. Eigendecomposition for circulant matrices (with proof).

Definition. (Continuous) convolution Let $x(t)$ and $y(t)$ be two given functions on $t \in T$. The convolution of these functions is defined as

$$(x * y)(t) = \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau$$

Definition. Discrete convolution We can approximate the integral above by a quadrature sum. In that case the discrete summation is left

$$z_i = \sum_{j=0}^{n-1} x_j y_{i-j}$$

This is called **discrete convolution**.

Let us investigate how to eigendecompose that special matrix called circulant

$$\begin{pmatrix} c_n & c_{n-1} & c_{n-2}\cdots c_1 & & \\ c_1 & c_0 & c_{n-1}\cdots c_2 & & \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n-1} & c_{n-2} & c_{n-3} & \cdots & c_n \end{pmatrix}$$

Theorem. *On diagonalization (eigendecomposition) of circulant matrices. Let C be a circulant matrix of size $(n \times n)$ and c be its first column, then*

$$C = \frac{1}{n} F_n^* diag(F_n c) F_n$$

Proof. Consider some (complex) n-th root of 1 ω. Introduce λ

$$\lambda = c_n + \omega c_1 + \cdots + \omega^{n-1} c_{n-1}$$

But notice that by multiplying this equation by powers of ω, we get

$$\begin{aligned} \lambda\omega &= c_{n-1} + \omega c_n + \cdots + \omega^{n-1} c_{n-2} \\ \lambda\omega^2 &= c_{n-2} + \omega c_{n-1} + \cdots + \omega^{n-1} c_{n-3} \\ \lambda\omega^{n-1} &= c_1 + \omega c_2 + \cdots + \omega^{n-1} c_n \end{aligned}$$

But holy smokes, this is just rows of C in circulation! Hence if we put omegas in a vector, we get

$$\lambda(1, \omega, \cdots, \omega^{n-1}) = (1, \omega, \cdots, \omega^{n-1}) \cdot C$$

Note that by definition of eigenvectors vector of omegas is eigenvector of C, and λ is its eigenvalue.

Selecting and stacking possible omegas to the power of n ($\omega = 1, \omega_n, \cdots, \omega_n^{n-1}$) we get DFT matrix

$$\Lambda F_n = F_n C$$

Then

$$C = \frac{1}{n} F_n^* \Lambda F_n$$

□

34. Product of Toeplitz matrix by vector via FFT. BTTB matrix-by-vector product via 2D FFT.

Suppose we want to count matvec $Tx = y$ for Toeplitz matrix T of size $(m \times m)$ $(t_0, t_{-1}, \cdots, t_{-n}, t_1, \cdots, t_n)$, where $t_{(-)k}$ is element k steps below (above) main diagonal. We can "pad" T and x using FFT as follows:

$$y = (y_1, \cdots, y_m) = iFFT(FFT(t_0, t_1, \cdots, t_n, t_{-1}, \cdots, t_{-n})) \cdot FFT(x_1, x_2, \cdots, x_m, 0, \cdots, 0)$$

On multilevel Toeplitz matrices.
Suppose we have *2-dimensional discrete convolution*

$$y_{i_1 i_2} = \sum_{j_1, j_2}^{n} t_{i_1 - j_1, i_2 - j_2} x_{j_1 j_2}$$

Traversing it back into 2d reality, we have *BTTB (block Topelitz with Toeplitz blocks)* matrix:

$$T = \begin{pmatrix} T_0 & T_{-1} & T_{-2} \cdots T_{1-n} & \\ T_1 & T_0 & T_{-1} \cdots T_{2-n} & \\ T_2 & T_1 & T_0 \cdots T_{3-n} & \\ \vdots & \vdots & \vdots & \ddots \; \vdots \\ T_{n-1} & T_{n-2} & T_{n-3} \cdots T_0 & \end{pmatrix}$$

Where $T_k = t_{k, i_2 - j_2}$ are Toeplitz matrices.

The matvec of these matrices is just modification of 1-d case

$$Y = \begin{pmatrix} y_{11} & \cdots & y_{1m} \\ y_{21} & \cdots & y_{2m} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mm} \end{pmatrix} = iFFT2d(FFT2d \begin{pmatrix} t_{0,0} & t_{1,0} & \cdots & t_{n,0} & t_{-n,0} & \cdots & t_{-1,0} \\ t_{0,1} & t_{1,1} & \cdots & t_{n,1} & t_{-n,1} & \cdots & t_{-1,1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ t_{0,-1} & t_{1,-1} & \cdots & t_{n,-1} & t_{-n,-1} & \cdots & t_{-1,-1} \end{pmatrix})\cdot$$

$$\begin{pmatrix} x_{11} & \cdots & x_{1m} \\ x_{21} & \cdots & x_{2m} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mm} \end{pmatrix}$$

FFT2d is just sequence of your usual *FFT* transforms applied first to rows and then to columns of matrix.

35. Matrix functions. Matrix exponential and its applications. Problem with evaluating matrix exponential and how to avoid it. Schur-Parlett algorithm. Matrix functions via Pade approximation.

Definition. Matrix polynomial Matrix polynomial $P(A)$ for matrix A is given by

$$P(A) = \sum_{k=0}^{n} c_k A^k$$

Definition. Matrix function in terms of Taylor series of a matrix

$$f(A) = \sum_{k=0}^{\infty} c_k A^k$$

Definition. Matrix function in terms of Cauchy integral representation

$$f(A) = \int_{\Gamma} f(z)(zI - A)^{-1} dz$$

Where Γ is closed contour that encloses the spectrum of A.

Definition. Matrix exponentional

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!} = I + A + \frac{A^2}{2} + \frac{A^3}{3!} + \cdots$$

The problem: for large x Taylor series converges very slowly to e^x, thus for A with big norm matrix exponential also converges slowly.

Solution one: compute approximation in Krylov subspace using Arnoldi orthogonalization method

$$f(A) \approx f(QHQ^*)v = Qf(H)Q^*v$$

H is upper Hessenberg. Now we can easily reduce it to triangualar and compute functional for triangular matrix (Schur-Parlett alogorithm, discuss later).

Solution two: **Pade approximation**

$$e^x = \frac{p(x)}{q(x)}$$

Solution three: **Scaling & squaring algorithm**

1. Scale matrix $B = \frac{A}{2^k}$. It will have lesser norm (less than 1).
2. Compute $C = e^B$ via *Pade approximation.*
3. $e^A \approx C^{2k}$

Definition. Schur-Parlett algorithm

1. Schur decomposition of A: $A = UTU^*$
2. $F(A) = UF(T)U^*$
3. Compute functional of triangular matrix: first, diagonals

$$f_{ii} = F(t_i i)$$

Then, other elements are counted one by one:

$$f_{ij} = t_{ij} \frac{f_{ii} - f_{jj}}{t_{ii} - t_{jj}} + \sum_{k=i+1}^{j-1} \frac{f_{ik} t_{kj} - t_{ki} f_{kj}}{t_{ii} - t_{jj}}$$

36. Sylvester and Lyapunov equations, their applications. Solving Sylvester equation using Bartels-Stewart method.

Definition. Sylvester equation. For given matrices $A \in \mathbf{C}^{n\times n}, B \in \mathbf{C}^{m\times m}, C \in \mathbf{C}^{n\times m}$ **Sylvester matrix equation** with respect to $X \in \mathbf{C}^{n\times m}$ is defined as:

$$AX + (-)XB = C$$

Select applications: displacement rank of a matrix.

$$Z_e T - T Z_f = GH^T, G \in (n\times r), H \in (n \times r)$$

Now let us rewrite it using vector product

$$vec(AX + XB) = vec(C)$$

$$(I \otimes A + B^T \otimes I) vec(X) = vec(C)$$

System is linear with its matrix $(I \otimes A + B^T \otimes I)$.

Now we introduce **Bartels-Stewart** method: it uses Schur decomposition of A and B

$$A = Q_1 T_1 Q_1^*$$

$$B^T = Q_2 T_2 Q_2^*$$

Then

$$(I \otimes A + B^T \otimes I) = (I \otimes (Q_1 T_1 Q_1^*) + (Q_2 T_2 Q_2^*) \otimes I) = (Q_2 \otimes Q_1)(I \otimes T_1 + T_2 \otimes I)(Q_2^* \otimes Q_1^*) =$$

$$= (Q_2 \otimes Q_1)(I \otimes T_1 + T_2 \otimes I)(Q_2 \otimes Q_1)^{-1}$$

Thus we need only to solve

$$(I \otimes T_1 + T_2 \otimes I)x = c$$

In case A and B are *Hermitian*, T_1 and T_2 are diagonal by Schur.

Definition. Lyapunov equation. **continious Lyapunov matrix equation** with respect to X is defined as:

$$AX + XA^T = C$$

discrete Lyapunov matrix equation with respect to X is defined as:

$$AXA^* - X = C$$

Select applications:

1. Stability of systems: it can be shown that system is stable iff $\forall Q = Q^* > 0$ there exists a unique positive definite solution P:

$$AP + PA^* = Q$$

2. Model order reduction

$$\frac{d\hat{x}}{dt} = \hat{A}\hat{x} + \hat{B}u, y = \hat{C}\hat{x}$$

$\hat{A}, \hat{B}, \hat{C}$ are found through Lyapunov equations.