

# Numerical Linear Algebra for Computational Science and Information Engineering

## Lecture 05

### Sparse Data Structures and Basic Linear Algebra Subprograms

Nicolas Venkovic  
[nicolas.venkovic@tum.de](mailto:nicolas.venkovic@tum.de)

Group of Computational Mathematics  
School of Computation, Information and Technology  
Technical University of Munich

Winter 2025-26



# Outline I

1	Basic linear algebra subprograms (BLAS)	1
2	Sparse matrix data structures Section 9.1 in Darve & Wootters (2021)	10
3	Sparse BLAS Section 9.1 in Darve & Wootters (2021)	28
4	Sparse matrices and graphs Section 9.2 in Darve & Wootters (2021)	29
5	Homework problem	32
6	Practice session	33

# Basic linear algebra subprograms (BLAS)

# Basic linear algebra subprograms (BLAS)

- ▶ What is BLAS?
  - Originated in the 1970s, as a set of **low-level routines** for common **linear algebra operations**, first written in Fortran.
  - Became a **standard** for the specification of linear algebra subroutines.
- ▶ Why use BLAS?
  - **Performance:** algorithmic optimizations, multi-threading, vectorization, loop unrolling, cache and register blocking, instruction pipelining, ...
  - **Portability:** Consistent interface across different platforms.
- ▶ Over time, **different BLAS libraries** have been developed, in **different languages**, for **different hardware**:
  - Intel oneAPI MKL: Proprietary, highly optimized for Intel architectures, GPU support through SYCL, comprehensive.
  - OpenBLAS: Open source, multi-architecture support, some GPU support, derived from GotoBLAS, community-driven.
  - BLIS: Open source, research-oriented (UT Austin).
  - ATLAS: Open source, empirical auto-tuning during build.
  - GPU only: Nvidia cuBLAS, AMD rocBLAS, ...

# Common BLAS subroutines

BLAS routines are **organized into levels**, and follow a **naming convention** for most standard operations.

► **Level 1** (**vector** operations, typically  $O(n)$  ops.):

- Dot product (**DDOT**, **SDOT**, ...):  $x^T y$
- Vector addition (**DAXPY**, **SAXPY**, ...):  $y \leftarrow \alpha x + y$
- Vector norms (**DNRM2**, **SNRM2**, ...):  $\|x\|_2$

► **Level 2** (**matrix-vector** operations, typically  $O(n^2)$  ops.):

- Matrix-vector multiply (**DGEMV**, **SGEMV**):  $y \leftarrow \alpha Ax + \beta y$
- Rank-1 update (**DGER**, **SGER**):  $A \leftarrow \alpha xy^T + A$
- Triangular solve (**DTRSV**, **STRSV**):  $x \leftarrow T^{-1}x$

► **Level 3** (**matrix** operations, typically  $O(n^3)$  ops.):

- Matrix-matrix multiply (**DGEMM**, **SGEMM**, ...):  $C \leftarrow \alpha AB + \beta C$
- Rank- $k$  update (**DSYRK**, **SSYRK**, ...):  $C \leftarrow \alpha AA^T + \beta C$

The first letter in the name of a subroutine represents the data type:

**D**: double precision real

**S**: single precision real

**C**: single precision complex

**Z**: double precision complex

# Common BLAS subroutines, cont'd

## Level 1 BLAS

	dim	scalar	vector	scalars	5-element array		prefixes
SUBROUTINE	xROTRG	(		A, B, C, S )	D1, D2, A, B, C, S )	Generate plane rotation	S, D
SUBROUTINE	xROTMG	(			PARAM )	Generate modified plane rotation	S, D
SUBROUTINE	xROT	( N,	X, INCX, Y, INCY,			Apply plane rotation	S, D
SUBROUTINE	xROT	( N,	X, INCX, Y, INCY,		PARAM )	Apply modified plane rotation	S, D
SUBROUTINE	xSWAP	( N,	X, INCX, Y, INCY )			$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE	xSCAL	( N,	X, ALPHA, Y, INCX )			$x \leftarrow \alpha x + y$	S, D, C, Z, CS, ZD
SUBROUTINE	xCOPY	( N,	X, INCX, Y, INCY )			$y \leftarrow x$	S, D, C, Z
SUBROUTINE	xAXPY	( N,	ALPHA, X, INCX, Y, INCY )			$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION	xDOT	( N,	X, INCX, Y, INCY )			$\text{dot} \leftarrow x^T y$	S, D, DS
FUNCTION	xDOTU	( N,	X, INCX, Y, INCY )			$\text{dot} \leftarrow x^T y$	C, Z
FUNCTION	xDOTC	( N,	X, INCX, Y, INCY )			$\text{dot} \leftarrow x^H y$	C, Z
FUNCTION	xDOT	( N,	X, INCX, Y, INCY )			$\text{dot} \leftarrow \alpha + x^T y$	SDS
FUNCTION	xNRM2	( N,	X, INCX )			$\text{nrm2} \leftarrow  x _2$	S, D, SC, DZ
FUNCTION	xASUM	( N,	X, INCX )			$\text{asum} \leftarrow  re(x) _1 +  im(x) _1$	S, D, SC, DZ
FUNCTION	IxAMAX	( N,	X, INCX )			$\text{amax} \leftarrow 1^N k \ni re(x_k) + im(x_k)$	S, D, C, Z
						$= \max(re(x_i)) + im(x_i)$	

## Level 2 BLAS

	options	dim	b-width	scalar	matrix	vector	scalar	vector	
xGEMV	( TRANS,	M, N,	ALPHA, A, LDA, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$		S, D, C, Z
xGEMV	( TRANS,	M, N, KL, KU,	ALPHA, A, LDA, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$		S, D, C, Z
xHEMV	( UPLO,	M, N,	ALPHA, A, LDA, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y$		C, Z
xHEMV	( UPLO,	M, N,	ALPHA, A, LDA, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y$		C, Z
xHPMV	( UPLO,	M, N,	ALPHA, AP, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y$		C, Z
xSPMV	( UPLO,	M, N,	ALPHA, A, LDA, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y$		S, D
xSPMV	( UPLO,	M, N,	ALPHA, AP, X, INCX, BETA, Y, INCY )				$y \leftarrow \alpha Ax + \beta y$		S, D
xTPMV	( UPLO, TRANS, DIAG,	N,	A, LDA, X, INCX )				$y \leftarrow \alpha Ax + \beta y$		S, D
xTPMV	( UPLO, TRANS, DIAG,	N, K,	A, LDA, X, INCX )				$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$		S, D, C, Z
xTPMV	( UPLO, TRANS, DIAG,	N, K,	AP, X, INCX )				$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$		S, D, C, Z
xTRSV	( UPLO, TRANS, DIAG,	N,	A, LDA, X, INCX )				$x \leftarrow Ax, x \leftarrow A^{-T}, x \leftarrow A^{-H}$		S, D, C, Z
xTRSV	( UPLO, TRANS, DIAG,	N, K,	A, LDA, X, INCX )				$x \leftarrow Ax, x \leftarrow A^{-T}, x \leftarrow A^{-H}$		S, D, C, Z
xTPSV	( UPLO, TRANS, DIAG,	N, K,	AP, X, INCX )				$x \leftarrow Ax, x \leftarrow A^{-T}, x \leftarrow A^{-H}$		S, D, C, Z
	options	dim	scalar	vector	vector	matrix			
xGR	(	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xy^T + A, A - m \times n$		S, D
xGENU	(	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xy^T + A, A - m \times n$		C, Z
xGERC	(	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xy^H + A, A - m \times n$		C, Z
xHER	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xz^H + A$		C, Z
xHPR	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xz^H + A$		C, Z
xHER2	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xy^H + y(ax)^H + A$		C, Z
xHPR2	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, AP )				$A \leftarrow \alpha xy^H + y(ax)^H + A$		C, Z
xSTR	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xz^T + A$		S, D
xSPR	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, AP )				$A \leftarrow \alpha xz^T + A$		S, D
xSTR2	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, A, LDA )				$A \leftarrow \alpha xy^T + \alpha yx^T + A$		S, D
xSPR2	( UPLO,	M, N,	ALPHA, X, INCX, Y, INCY, AP )				$A \leftarrow \alpha xy^T + \alpha yx^T + A$		S, D

## Level 3 BLAS

	options	dim	scalar	matrix	matrix	scalar	matrix	
xGEMM	( TRANSA, TRANSB,	M, N, K,	ALPHA, A, LDA, B, LDB, BETA, C, LDC )				$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
xSTMM	( SIDE, UPLO,	M, N,	ALPHA, A, LDA, B, LDB, BETA, C, LDC )				$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = AT$	S, D, C, Z
xHEMM	( SIDE, UPLO,	M, N,	ALPHA, A, LDA, B, LDB, BETA, C, LDC )				$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = AH$	C, Z
xSYRK	( UPLO, TRANS,	N,	K, ALPHA, A, LDA, BETA, C, LDC )				$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha CT^T + \beta C, C - n \times n$	S, D, C, Z
xHERK	( UPLO, TRANS,	N,	K, ALPHA, A, LDA, BETA, C, LDC )				$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
xSYRK2K	( UPLO, TRANS,	N,	K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )				$C \leftarrow \alpha AA^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} BA^H + \beta C, C - n \times n$	S, D, C, Z
xHER2K	( UPLO, TRANS,	N,	K, ALPHA, A, LDA, B, LDB, BETA, C, LDC )				$C \leftarrow \alpha AA^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} BA^T + \beta C, C - n \times n$	C, Z
xTRMM	( SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A, LDA, B, LDB )				$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
xTRMM	( SIDE, UPLO, TRANS,	DIAG, M, N,	ALPHA, A, LDA, B, LDB )				$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

2

University of Tennessee, Oak Ridge National Laboratory, Numerical Algorithms Group Ltd. (1997). Basic linear algebra subprograms – A quick reference guide. (<https://www.netlib.org/blas>)

# BLAS in practice

- ▶ BLAS interfaces tend to be **mathematically opaque**.
- ▶ Using the Intel oneAPI MKL C interface:
  - The Julia code  $Ax = A*x; AtAx = A'Ax$  becomes:

```
double *x = (double*)mkl_malloc(m * sizeof(double), sizeof(double));
double *Ax = (double*)mkl_malloc(n * sizeof(double), sizeof(double));
double *AtAx = (double*)mkl_malloc(m * sizeof(double), sizeof(double));
for (int i=0; i<m; i++)
    x[i] = rand() / (double) RAND_MAX;
for (int i=0; i<maxit; i++) {
    cblas_dgemv(CblasColMajor, CblasNoTrans, n, m, 1., A, n, x, 1, 0., Ax, 1);
    cblas_dgemv(CblasColMajor, CblasTrans, n, m, 1., A, n, Ax, 1, 0., AtAx, 1);
```

- Documentation:  
<https://www.intel.com/content/www/us/en/docs/onemkl-developer-reference-dpcpp/2024-2/blas-routines.html>
- ▶ For interfaces to other implementations, see
  - **OpenBLAS**: <https://github.com/OpenMathLib/OpenBLAS>
  - **ATLAS**: <https://github.com/flame/blis>
  - **BLIS**: <http://math-atlas.sourceforge.net/>

## BLAS in practice, cont'd

- The cost of **enhanced portability** often comes in the form of **building challenges**.
  - E.g., MKL and OpenBLAS offer support for various CPU vendors and GPUs.
- For **Intel oneAPI MKL**, there is a dedicated web tool to help with the linking configuration:

Intel® oneAPI Math Kernel Library (oneMKL) Link Line Advisor v6.23

Select Intel® product:	<input type="button" value="oneMKL 2024"/>
Select OS:	<input type="button" value="Select operating system"/>
Select programming language:	<input type="button" value="Select programming language"/>
Select compiler:	<input type="button" value="Select compiler"/>
Select architecture:	<input type="button" value="Select architecture"/>
Select dynamic or static linking:	<input type="button" value="Select linking"/>
Select Interface layer:	<input type="button" value="Select Interface"/>
Select threading layer:	<input type="button" value="Select threading"/>
Select OpenMP library:	<input type="button" value="Select OpenMP"/>
Enable OpenMP offload feature to GPU:	<input type="checkbox"/>
Select cluster library:	<input type="checkbox"/> Parallel Direct Sparse Solver for Clusters (BLACS required) <input type="checkbox"/> Cluster Discrete Fast Fourier Transform (BLACS required) <input type="checkbox"/> ScalAPACK (BLACS required) <input type="checkbox"/> BLACS
Select MPI library:	<input type="button" value="Select MPI"/>
Select the Fortran 95 Interfaces:	<input type="checkbox"/> BLAS95 <input type="checkbox"/> LAPACK95

Select SYCL domain library:	<input type="button" value="Select Domain"/>
Link with Intel® oneMKL libraries explicitly:	<input type="checkbox"/>
Link with DPC++ debug runtime compatible libraries:	<input type="checkbox"/>
Use this link line: <Please select all required parameters above>	
Compiler options:	
Notes:	<p>o Set INCLUDE, MKLROOT, TBBROOT, LD_LIBRARY_PATH, LIBRARY_PATH, CPATH and NLSPATH environment variables in the command shell using the Intel(R) oneAPI setvars script in Intel(R) oneAPI root directory. Please also see the Intel(R) oneMKL Developer Guide.</p>

<https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html>

# Linear algebra package (LAPACK)

## ► What is LAPACK?

- Set of Fortran 90 routines to solve **linear systems**, **eigenvalue problems**, and **SVDs** with **dense but small to moderately sized** as well as **structured sparse** (banded, tridiagonal, ...) matrices.
- Successor to LINPACK (1979, for linear systems and least squares pbs.) and EISPACK (1976, for eigenvalue problems).
- Developed and maintained by an international **team of researchers**.

## ► Key characteristics:

- Optimized for **performance**, **portability** and **numerical stability**.
- Relies heavily on BLAS, especially Level 2 and 3.
- Performance depends critically on the **BLAS implementation** used.
- Handles higher-level algorithms and delegates operations to **BLAS**.

## ► Available through various implementations:

- Reference **LAPACK**: Standard implementation, focus on correctness.
- Intel **MKL**: Optimized LAPACK routines alongside BLAS.
- GPU only: Nvidia **cuSOLVER**, AMD **rocSOLVER**.

# Nomenclature of LAPACK subroutines

LAPACK routines follow a **structured naming convention**: XYYZZZ

► **Data types (X):**

D: double precision real

S: single precision real

C: single precision complex

Z: double precision complex

► **Common matrix types (YY):**

GE: general

SY: symmetric

HG: upper Hessenberg

PO: SPD/HPD

TR: triangular

BD: bidiagonal

► **Common computational tasks (ZZZ):**

SV: solve linear system

TRF: triangular factorization

TRS: solve using factorization

CON: estimate conditioning

EV: solve eigenvalue problem

► **Examples of (driver) subroutines:**

- DGESV: linear solve with real general matrix in double precision.

- CPOSV: linear solve with (complex) HPD matrix in single precision.

- ZGEEV: eigensolve with general complex matrix in double precision.

# Structure of LAPACK subroutines

- There are three types of LAPACK routines:
  - **Driver** routines: solves a **complete problem**, e.g., linear systems, eigenvalue problems, least-squares problems, ...
  - **Computational** routines: performs an **intermediate level task**, e.g., LU factorization, tridiagonal reduction, ...
  - **Auxiliary** routines: **unblocked sub-tasks of block algorithms**, BLAS-like operations, other low level tasks.



- **Driver** routines listed in the online documentation:

<https://www.netlib.org/lapack/explore-html/modules.html>

- **Computational** routines listed by module:

<https://www.netlib.org/lapack/lug/node37.html>

- **Auxiliary** routines listed by category:

<https://www.netlib.org/lapack/lug/node144.html>

# BLAS and LAPACK in Julia

- ▶ Default implementation:
  - Ships with **multi-threaded OpenBLAS** and reference **LAPACK**.
  - **Flexible**, i.e., can use other implementations, e.g., **MKL**, **BLIS**, ...
- ▶ Three **implementation-independent** levels of access (like in Python):
  - **Interface wrappers** via `LinearAlgebra.{BLAS,LAPACK}`:

<code>BLAS.gemm!</code> ,	<code>LAPACK.getrf!</code> ,	...
<b>most control</b>	<b>no extra copies/allocations</b>	<b>math-implicit</b>
  - **Intermediate level functions**:

<code>dot(x,y)</code> ,	<code>mul!(C,A,B)</code> ,	<code>lu(A)</code> ,	...
<b>less control</b>	<b>in-place versions available</b>	<b>good compromise</b>	
  - **High-level syntax**:

<code>A * x</code> ,	<code>A \ b</code> ,	<code>A / B</code> ,	...
<b>least control</b>	<b>extra copies/allocations</b>	<b>math-explicit</b>	
- ▶ Key features:
  - **Matrix type** specified by **data structure**, e.g., `Symmetric`, `Tridiagonal`.
  - **Multiple dispatch**: function behavior depends on types of **all** arguments.
  - Operations **preserve matrix structure** when applicable.

# Sparse matrix data structures

Section 9.1 in Darve & Wootters (2021)

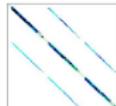
## Sparse matrices

- ▶ **Sparse matrices** are **matrices** with relatively **few non-zero components**.
- ▶ Natural occurrence in scientific applications:
  - **Discretized differential equations:**
    - ODEs: chemical reactions, multi-body systems with short-range interactions, multi-agent systems with local interactions, ...
    - PDEs: fluid dynamics, solid mechanics, electromagnetics, ...
    - DAEs: circuit simulation, power grid modeling, ...
  - **Networks and graphs:**
    - Adjacency, transition and Laplacian matrices of sparse graphs.
  - **Data science:**
    - Feature matrices in high-dimensional data.
- ▶ Important properties:
  - **Inverses** of sparse matrices are generally **dense**, i.e., not sparse.
  - **Factorizations** of sparse matrices **may be reasonably sparse**.
  - **Dense matrices** can be approximated by **sparse matrices**, i.e., using sparse approximate inverses (SPAI).

## Repository of sparse matrices

- ▶ Researchers and developers often need multiple sparse matrices with documented characteristics to benchmark NLA algorithms.
- ▶ In particular, the **SuiteSparse Matrix Collection** is widely used for this:  
<https://sparse.tamu.edu/>
  - Close to 3,000 matrices available.
  - Matrices from all sorts of applications.
  - Metadata available include: author, application field, rank, condition number, singular values, definiteness, symmetry and lack thereof, ...
- ▶ We can generally distinguish between two types of sparse matrices:
  - **Structured**: typically coming from differential equations discretized on structured grids/meshes.

E.g., `sherman5` (computational fluid dynamics problem):



- **Unstructured**: most other cases.

E.g., `bp_1000` (optimization problem):



## Sparse matrix data structures

- ▶ The use of **proper data structures** is essential to
  - limit memory requirements and achieve good performance when deploying basic linear algebra operations and NLA algorithms with sparse matrices.
- ▶ There is **no unique sparse matrix data structure** to optimally serve all purposes in all situations.
- ▶ In general, the choice of a sparse data structure can be influenced by
  - Sparsity pattern of the matrix.
  - Hardware architecture:
    - Memory layout.
    - Sequential vs parallel with shared and/or distributed memory vs GPU.
  - Algorithm and operations:
    - Type of access.
    - BLAS level, i.e., 1, 2 or 3.
  - Implementation requirements.

## Sparse matrix data structures, cont'd

- ▶ There are many sparse matrix data structure formats. In particular:

- #### - Coordinate (coo)

intuitive/explicit      not efficient      large community support  
most convenient/used for construction

- Compressed sparse row (CSR), compressed sparse column (CSC)

**lowest memory need**    **efficient**    **large community support**  
**most used**

- #### ► Variants of CSR and CSC:

## Sparse matrix data structures, cont'd<sub>2</sub>

## Coordinate (COO) format

- ▶ A COO data structures format is composed of:
  - Array of **non-zero components** (`val`)
  - Array of **row indices of each component** (`row_idx`)
  - Array of **column indices of each components** (`col_idx`)

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

`val` = [ $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{33}, a_{34}, a_{43}$ ]

`row_idx` = [1, 1, 1, 2, 2, 3, 3, 4]

`col_idx` = [1, 2, 3, 1, 2, 3, 4, 3]

- ▶ Key characteristics:

- Explicit storage of all indices (higher memory usage)
- **No particular ordering** required
- **Duplicates allowed** (values must be summed)
- **Flexible** for matrix **construction** and **modification**

## Compressed sparse row (CSR) format

- ▶ A CSR data structures format is composed of:
  - Array of **non-zero components** (`val`)
  - Array of **column indices of each component** (`col_idx`)
  - Array of **non-zero value indices where each row starts** (`row_start`)
- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

`val` = [ $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{33}, a_{34}, a_{43}$ ]

`col_idx` = [1, 2, 3, 1, 2, 3, 4, 3]

`row_start` = [1, 4, 6, 8, 9]

- ▶ Key characteristics:
  - Compact storage (lower memory than COO)
  - **Fast row access**
  - **Values must be ordered by row**
  - **Difficult to modify structure dynamically**

## Compressed sparse column (CSC) format

- ▶ A CSC data structures format is composed of:
  - Array of **non-zero components** (`val`)
  - Array of **row indices of each component** (`row_idx`)
  - Array of **non-zero indices where each column starts** (`col_start`)

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

`val` = [ $a_{11}, a_{21}, a_{12}, a_{22}, a_{13}, a_{33}, a_{43}, a_{34}$ ]

`row_idx` = [1, 2, 1, 2, 1, 3, 4, 3]

`col_start` = [1, 3, 5, 8, 9]

- ▶ Key characteristics:

- Compact storage (lower memory than COO)
- **Fast column access**
- **Values must be ordered by column**
- **Difficult to modify structure dynamically**

## Block sparse row (BSR) format

- ▶ A BSR (or BCSR) data structure format is composed of:
  - **Block dimensions** ( $r \times c$ )
  - Array (or matrix) of **all components of non-zero blocks** (val)
  - Array of **non-zero block column indices** (col\_idx)
  - Array of **block indices where each block row starts** (row\_start)

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

$$r = 2, c = 2$$

$$\text{val} = [a_{11}, a_{12}, a_{21}, a_{22}, a_{13}, 0, 0, 0, a_{33}, a_{34}, a_{43}, 0]$$

$$\text{col\_idx} = [1, 2, 2]$$

$$\text{row\_start} = [1, 3, 4]$$

- ▶ Key characteristics:

- **Zero values within non-zero blocks are stored**
- **Similar to CSR but operates on blocks**

## Mapped block row (MBR) format

- ▶ A MBR data structure format is composed of:
  - **Block dimensions ( $r \times c$ )**
  - Array of **non-zero components of non-zero blocks (val)**
  - Array of **non-zero block column indices (col\_idx)**
  - Array of **sparsity pattern encoding (b\_map)**
  - Array of **block indices where each block row starts (row\_start)**

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

$$r = 2, c = 2$$

$$\text{val} = [a_{11}, a_{12}, a_{21}, a_{22}, a_{13}, a_{33}, a_{34}, a_{43}]$$

$$\text{col\_idx} = [1, 2, 2] \quad \text{b\_map} = [15, 1, 7] \quad \text{row\_start} = [1, 3, 4]$$

- ▶ Key characteristic:

- **Non-zero values within non-zero blocks are not stored**

## Modified sparse row (MSR) format

- ▶ A MSR data structure format is composed of:
  - Array of **diagonal elements first**, then **other non-zeros** (`val`)
  - Composite array `idx` :=  $[row\_start, col\_idx]$  where:
    - `row_start` contains the **index of off-diagonal non-zero value where each row starts**.
    - `col_idx` contains **column indices of each off-diagonal non-zero component**.
- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$
$$\text{val} = [a_{11}, a_{22}, a_{33}, 0, -1, a_{12}, a_{13}, a_{21}, a_{34}, a_{43}]$$
$$\text{idx} = [6, 8, 9, 10, 11, 2, 3, 1, 4, 3]$$
- ▶ Key characteristics:
  - **Diagonal elements stored first**  $\implies$  **Fast diagonal access**
  - Dummy element, here  $-1$ , stored in `val` for consistency with `idx` (?)

## Ellpack (ELL) format

- ▶ An ELL data structure format is composed of:
  - Maximum number of non-zero components on a row (`row_nnz`)
  - Array of all components stored in column-major order, from the block of left-aligned non-zero components (`val`)
  - Array of column indices of stored components (`col_idx`)

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

`row_nnz = 3`

`val = [a11, a21, a33, a43, a12, a22, a34, 0, a13, 0, 0, 0]`

`col_idx = [1, 1, 3, 3, 2, 2, 4, -1, 3, -1, -1, -1]`

- ▶ Key characteristics:

- Stores  $2 \times \text{row\_nnz}$  values, including some zeros
- Wasteful if number of non-zero components varies significantly from one row to another

## Diagonal (DIA) format

- ▶ A DIA data structure format is composed of:
  - Array of **components on non-zero diagonals** padded to  $n$  (**val**)
  - Array of **offset indices** (**ioff**)

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

**val** = [ $\ast$ ,  $a_{21}$ , 0,  $a_{43}$ ,  $a_{11}$ ,  $a_{22}$ ,  $a_{33}$ , 0,  $a_{12}$ , 0,  $a_{34}$ ,  $\ast$ ,  $a_{13}$ , 0,  $\ast$ ,  $\ast$ ]

**ioff** = [-1, 0, 1, 2]

- ▶ Key characteristics:

- **Fast diagonal access**
- Wasteful for diagonal with large offset indices (?)

## List of list (LIL) format

- ▶ A LIL data structure format is composed of:
  - A **list (rows) of lists**, one per row, **each list storing column indices of non-zero components**.
  - A **list (data) of lists**, one per row, **each list storing non-zero components, ordered consistently with the indices in rows**.

- ▶ Example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix} \text{rows} = \begin{bmatrix} [1, 2, 3] \\ [1, 2] \\ [3, 4] \\ [3] \end{bmatrix} \text{data} = \begin{bmatrix} [a_{11}, a_{12}, a_{13}] \\ [a_{21}, a_{22}] \\ [a_{33}, a_{34}] \\ [a_{43}] \end{bmatrix}$$

- ▶ Key characteristics:
  - **No particular ordering** required for column indices
  - **Unordered column indices** slows down access
  - Mostly used for matrix **construction**, particularly in **Python**

## Sparse matrix data structures in practice

- ▶ Intel oneAPI MKL supports sparse vectors, and the sparse matrix data structures CSR, CSC, COO and BSR.

For example, using the C interface:

- A COO matrix can be **created** as follows:

```
double val[] = {1., 2., 3.};
MKL_INT row_idx[] = {0, 2, 1};
MKL_INT col_idx[] = {0, 1, 2};
sparse_matrix_t A;
mkl_sparse_d_create_coo(&A, SPARSE_INDEX_BASE_ZERO, 3, 3, 3, row_idx, col_idx, val);
```

- Sparse matrices can be **defined in other formats**, namely CSR, CSC and BSR, **directly from their underlying data structures**.
- Only two functions to **convert constructed sparse matrices** into  
CSR (`mkl_sparse_convert_csr`)  
and BSR (`mkl_sparse_convert_bsr`).

Possible to convert  $A$  into CSC, by using the CSR representation of  $A^T$ .

- **Documentation:**

<https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-2/matrix-manipulation-routines.html>

## Sparse matrix data structures in practice, cont'd

- ▶ **Nvidia cuSPARSE** also supports several vectors, and several sparse matrix data structures:
  - COO, CSR, CSC and BSR
  - Sliced Ellpack (SELL)
  - Blocked Ellpack (BLOCKED-ELL)

### Documentation:

<https://docs.nvidia.com/cuda/cusparse/#cusparse-storage-formats>

- ▶ Other implementations:
  - **AMD ROCsparse**: proprietary, for GPU
  - **SuiteSparse, PETSc, Trilinos, OSKI, PSBLAS, ...** : open-source

# Sparse matrix data structures in Julia

- ▶ Support of basic structured formats through `LinearAlgebra.jl`:  
Diagonal, Bidiagonal, Tridiagonal, SymTridiagonal, ...
- ▶ Standard library support through `SparseArrays.jl`:
  - Only CSC (`SparseMatrixCSC`) is supported by default:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
    m::Int           # Number of rows
    n::Int           # Number of columns
    colptr::Vector{Ti}    # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}    # Row indices of stored values
    nzval::Vector{Tv}      # Stored values, typically nonzeros
end
```

- Construction using COO-style input:

```
Is = [1, 3, 2]; Js = [1, 2, 3]; Vs = [1., 2., 3.]
A = sparse(Is, Js, Vs, 3, 3)
```

with immediate **conversion** to CSC.

- Construction using the `SparseMatrixCSC` struct:

```
A = SparseMatrixCSC(3, 3,
                    [1, 2, 3, 4],
                    [1, 3, 2],
                    [1., 2., 3.])
```

## Sparse matrix data structures in Julia, cont'd

- **Random** constructor for **sparse matrix** of density  $d$  with iid non-zero elements **distributed uniformly** in  $[0, 1]$ , `sprand(m, n, d)`.
  - **Random** constructor for **sparse matrix** of density  $d$  with iid non-zero elements **distributed according to the standard normal distribution**, `sprandn(m, n, d)`.
- More formats supported through other packages:
- `SparseMatricesCSR.jl`: Julia native implementation of CSR formats.
  - `MKLSparse.jl`: Julia wrappers to Intel oneAPI MKL sparse interface.
  - `SuiteSparse.jl`: Julia wrappers to SuiteSparse library.

:

:

# Sparse BLAS

Section 9.1 in Darve & Wootters (2021)

## Sparse basic linear algebra subprograms

- ▶ Sparse BLAS is the extension of BLAS for **sparse matrices and vectors**.
- ▶ Level 1 (vector operations):

Intel oneAPI MKL functions use a compressed sparse vector format:

<https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-2/sparse-blas-level-1-routines.html>

- Sparse  $y \leftarrow ax + y$  (SpAXPY): `mkl_sparse_x_axpy`

- ▶ Level 2-3 functions have **format-specific implementations**.

Intel oneAPI MKL offers access through an Inspector-Executor API:

<https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-2/inspector-executor-sparse-blas-execution-routines.html>

- Level 2 (matrix-vector operations):
  - Sparse matrix-vector product (SpMV): `mkl_sparse_x_mv`
- Level 3 (matrix-matrix operations):
  - Sparse matrix-(dense) matrix product (SpMM): `mkl_sparse_x_mm`
  - Sparse matrix-(sparse) matrix product (SpGEMM): `mkl_sparse_spmm`

# Sparse matrices and graphs

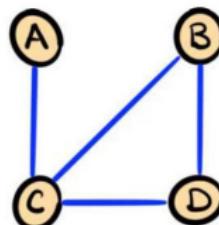
Section 9.2 in Darve & Wootters (2021)

## A few definitions

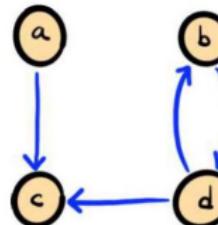
- Basics of graph theory are essential to sparse matrix computation.

### Definition (Graph)

- An **undirected graph** is a pair  $G = (V, E)$  formed by a non-empty finite set  $V$  of **vertices** and a set  $E \subseteq V \times V$  of **unordered pairs of vertices** referred to as **edges**.
- A **directed graph**  $G = (V, E)$  is formed by a set  $E$  of **ordered edges**.



An undirected graph  
with vertices  
 $V = \{A, B, C, D\}$  and  
edges  $E =$   
 $\{(A, C), (C, B), (C, D), (B, D)\}$ .



A directed graph  
with vertices  
 $V = \{a, b, c, d\}$  and  
edges  $E =$   
 $\{(a, c), (d, c), (b, d), (d, b)\}$ .

## A few definitions, cont'd

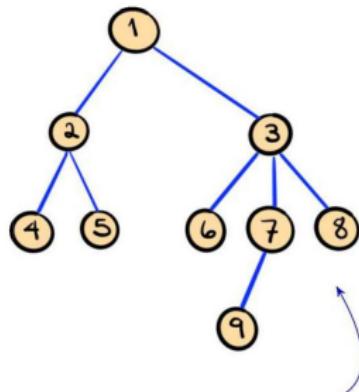
- ▶ A **path** from a vertex  $u$  to another vertex  $v$  is a **sequence of edges**  $(u_0, u_1), \dots, (u_{t-1}, u_t)$  such that  $u_0 = u$  and  $u_t = v$ .
- ▶ A graph is **connected** if there is a **path from any vertex  $u$  to any vertex  $v$** .
- ▶ A **tree** is a **connected graph without cycles**, i.e., with no path from a vertex to itself.

A tree has a **root**, i.e., a **designated vertex** represented **at the top** of the tree.

- ▶ If a tree has an edge  $(u, v)$ , and  $u$  is **closer to the root  $r$  than  $v$  is**, then we say that  $v$  is a **parent** and  $u$  is a **child**.

Each vertex in a tree has a unique parent.

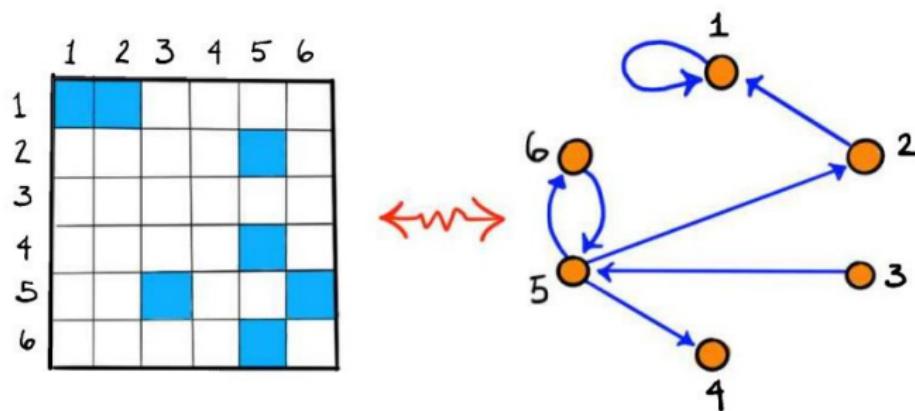
- ▶ A **leaf** is a **vertex** in a tree with **no children**.
- ▶ Family logic applies to define **descendants** and **ancestors**.



A tree. Vertex 1 is the root, and vertices 4, 5, 6, 9, 8 are leaves. Vertex 8 is 3's child, and 3 is 8's parent. Vertex 9 is 3's descendant, and 3 is 9's ancestor.

## Graph representation of sparsity patterns

- ▶ The **sparsity pattern** of a square matrix  $A \in \mathbb{F}^{n \times n}$  can be represented as a **directed graph** with  $n$  **vertices**.
  - ▶ In Darve and Wootters (2021), the convention is that a **directed edge**  $(i, j)$  from vertex  $j$  to vertex  $i$  exists if and only if  $a_{ij} \neq 0$ .
- For example:



- ▶ The sparsity pattern of **symmetric matrices** can be represented by **undirected graphs**.

# Homework problem

## Homework problem

Turn in **your own** solution to the following problem:

**Pb. 15** Consider the matrices

$$A = \begin{bmatrix} \bullet & \bullet & 0 & \bullet & 0 & 0 \\ 0 & \bullet & 0 & 0 & 0 & \bullet \\ 0 & \bullet & \bullet & 0 & 0 & 0 \\ 0 & \bullet & 0 & 0 & \bullet & 0 \\ 0 & 0 & 0 & 0 & \bullet & 0 \\ 0 & 0 & 0 & 0 & 0 & \bullet \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} \bullet & 0 & 0 & 0 & 0 & 0 \\ \bullet & 0 & \bullet & 0 & \bullet & 0 \\ 0 & \bullet & 0 & 0 & 0 & 0 \\ \bullet & \bullet & 0 & 0 & 0 & 0 \\ 0 & \bullet & 0 & \bullet & \bullet & 0 \\ 0 & 0 & \bullet & 0 & 0 & \bullet \end{bmatrix}$$

where each  $\bullet$  denotes a non-zero component.

Show the adjacency graphs of  $A$ ,  $B$ ,  $AB$  and  $BA$ . You may assume that there are no numerical cancellations in computing the products  $AB$  and  $BA$ .

Problem excerpted from Pb. 4 in Chap. 3 of Saad (2003).

Saad, Y. (2003). Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics.

# Practice session

## Practice session

- ① Use the `mmread` function from `MatrixMarket.jl` to read the matrix `cage3` from the `SuiteSparse` website. Investigate the default sparse data structure in which the matrix is stored in Julia.
- ② Write a function called `dcscmv` to perform SpMV in CSC format.
- ③ Write a function called `csc_to_coo` to convert a CSC matrix to COO format. Use the following custom type:

```
mutable struct SparseMatrixCOO
    m::Int # Number of rows
    n::Int # Number of columns
    rowval::Vector{Int} # Starting index for each row
    colval::Vector{Int} # Column indices
    nzval::Vector{Float64} # Matrix entries
end
```

- ④ Write a function called `dcoomv` to perform SpMV in COO format.
- ⑤ Write a function called `coo_to_csr` to convert a COO matrix to CSR format using a custom type `SparseMatrixCSR` with arguments `m::Int`, `n::Int`, `rowptr::Vector{Int}`, `colval::Vector{Int}` and `nzval::Vector{Float64}`.

## Practice session, cont'd

- ⑥ Write a function called `coo_to_csr2` to convert a COO matrix to CSR format making use of the built-in `sparse` function.  
Hint: Think of the relation between CSC and CSR.
- ⑦ Write a function called `dcsrsv` to perform SpMV in CSR format.
- ⑧ Write a function called `coo_to_ell` to convert a COO matrix to ELL format using a custom type `SparseMatrixELL` with arguments `m::Int`, `n::Int`, `rownnz::Int`, `colval::Vector{Int}` and `nzval::Vector{Float64}`.
- ⑨ Write a function called `dellmv` to perform SpMV in ELL format.