

High Performance Linear Algebra

Lecture 1: Introduction and Overview

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

November 17, 2026 — 14:00 to 16:00



Dipartimento
di Matematica
Università di Pisa



Table of Contents

1 Introduction

- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage



About this course: general information

First some *bureaucratic* information about the course:

- Course webpage: fdurastante.github.io/courses/hpla2026.html
- Lecture slides: fdurastante.github.io/courses/hpla2026.html#lectures

The **exam** will consist in a **project work** to be presented at the end of the course. This will involve the implementation and performance analysis of some linear algebra algorithms, or the performance analysis of existing libraries, possibly in relation to a specific application. The choice of the project topic will also depend on your Ph.D. research topic, so to make it more interesting and useful for you.



What is Linear Algebra?

1 Introduction

Linear Algebra is a branch of mathematics concerned with:

- **Vector spaces** and linear transformations
- Systems of **linear equations**, **matrices**, **vectors**
- Key concepts: determinants, eigenvalues, eigenvectors, singular values

Applications: computer graphics, machine learning, optimization, physics

Numerical Linear Algebra focuses on:

- Solving LA problems using numerical methods on computers
- Development of **efficient**, **stable**, and **accurate** algorithms
- Essential for *large-scale* problems where exact solutions are impractical

Problem 1: Linear Systems

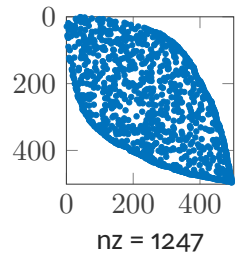
1 Introduction

Consider the Poisson equation (PDE):

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega$$

Discretization approach:

- Divide domain into grid: $N = n_1 \times n_2 \times n_3$ points
- Use finite difference approximation for derivatives
- Results in sparse linear system: $A\mathbf{u} = \mathbf{f}$
- $A \in \mathbb{R}^{N \times N}$ is **sparse**
- Most elements are zero
- N is typically very large



Sparse matrix pattern



Problem 2: Eigenvalue Problems

1 Introduction

Find scalar λ and vector $\mathbf{v}$ such that:

$$P\mathbf{v} = \lambda\mathbf{v}$$

Example: Markov Chains

- Transition matrix $P \in \mathbb{R}^{N\times N}$ ($P_{i,j} \geq 0$, rows sum to 1)
- Evolution: $\mathbf{p}_{\ell+1} = P\mathbf{p}_\ell$
- Stationary distribution $\boldsymbol{\pi}$ satisfies:

$$\boldsymbol{\pi}^\top = \boldsymbol{\pi}^\top P, \quad \boldsymbol{\pi}^\top \mathbf{1} = 1$$

- Finding $\boldsymbol{\pi}$ is an eigenvalue problem for large N



Problem 3: Matrix Equations

1 Introduction

Sylvester equation: $AX + XB = C$

Application: Model Reduction in Control Theory

LTI dynamical system:

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t), \quad \mathbf{y}(t) = C\mathbf{x}(t)$$

Balanced truncation approach:

1. Compute Gramians via Lyapunov equations:

$$AP + PA^\top + BB^\top = 0 \quad \text{(controllability)}$$
$$A^\top Q + QA + C^\top C = 0 \quad \text{(observability)}$$

2. Solve Sylvester equation: $AT + TS = B$
3. Efficient algorithms needed for large dimensions

Problem 4: Machine Learning

1 Introduction

Linear Regression:

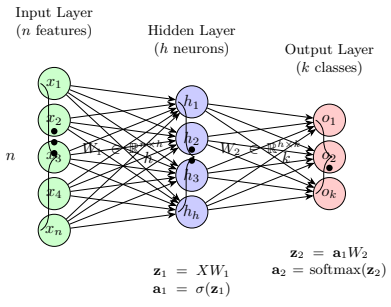
- Data: $X \in \mathbb{R}^{m \times n}$, targets: $\mathbf{y} \in \mathbb{R}^m$
- Find coefficients: $\min_{\boldsymbol{\beta}} \|X\boldsymbol{\beta} - \mathbf{y}\|_2^2$

Neural Networks:

- Weights as matrices: $W_1 \in \mathbb{R}^{n \times h}$, $W_2 \in \mathbb{R}^{h \times k}$
- Forward pass:

$$\mathbf{a}_1 = \sigma(XW_1)$$
$$\mathbf{a}_2 = \text{softmax}(\mathbf{a}_1 W_2)$$

- Training relies on matrix operations



Neural network



Take Home Message

Key Point

Applied mathematics is fundamentally about solving combinations of linear algebra problems.

Modern challenges:

- Ever larger problem sizes
- Need for reliable results in reasonable time
- Requirements: *efficient*, *scalable*, *parallel* algorithms

$\Rightarrow$ This motivates high-performance numerical linear algebra!



Recommended Reading

Recommended books on Linear Algebra and Numerical Linear Algebra include:

- Golub and Van Loan [4] - a classic covering matrix factorizations, eigenvalue problems, and singular value decomposition.
- Other notable works: [2], [8].
- Axler [1] offers an operative introduction to the theory of linear algebra.
- For comprehensive theory, see Horn and Johnson [5, 6].

We will focus on numerical and implementation aspects, with references for deeper theoretical insights.



Table of Contents

2 What does it mean large-scale?

- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage



What does “large-scale” mean?

In the previous section we have seen examples of problems in numerical linear algebra, where a recurrent theme is that the problem sizes are *large*.

But how large is large?

The answer: It depends!

It depends on:

- The problem we are dealing with
- The algorithm we are using
- The hardware we are using
- The time we have to solve the problem



What does “large-scale” mean?

In the previous section we have seen examples of problems in numerical linear algebra, where a recurrent theme is that the problem sizes are *large*.

But how large is large?
Furthermore, it’s a matter of *when* we are asking this question:

- 20 years ago: different answer
- Today: different answer
- 20 years from now: yet again different!



Problem size: different perspectives

The notion of “size” varies by problem type:

Linear systems:

- First approximation: number of unknowns
- **Sparse matrices**: combined information
  - Number of non-zero elements
  - Overall matrix dimensions
  - Distribution/stucture of the matrix
- **Dense matrices**: number of rows and columns
  - Does the matrix have any special structure?



Current capabilities

2 What does it mean large-scale?

Sparse linear systems:

- Solved with relative ease: several millions of unknowns
- Current frontier: hundreds of billions of unknowns

Eigenvalue problems:

- Compute few eigenvalues/eigenvectors
- Matrices with several millions of rows and columns

Matrix equations:

- More complicated situation
- Need to exploit special structure for large-scale problems



Exploiting structure: low-rank solutions

2 What does it mean large-scale?

For large matrix equations, we need solutions with special structure.

Example: Sylvester equation with low-rank solution

$$T = T_1 T_2^\top, \quad \text{where } T_1 \in \mathbb{R}^{m \times r}, T_2 \in \mathbb{R}^{n \times r}$$

with $r \ll m, n$

Key principle

Exploiting clever structures in the problem permits us to solve problems of larger size than we would be able to without these structures.

Computer science analogy: Building **data structures** that permit us to store and manipulate large amounts of data more efficiently.



Table of Contents

3 Where do we solve such *large* problems?

- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage



Solving large problems: parallel computers

3 Where do we solve such *large* problems?

To deal with problems which are large in the sense we have just discussed, we need to use **parallel computers**.

Key idea

Parallel computers perform multiple calculations **simultaneously** by using multiple processors or cores working together.



Memory organization

3 Where do we solve such *large* problems?

Parallel computers are classified by memory organization:

- **Shared memory systems**:
  - All processors share common memory space
  - Easy data access and communication
  - Limited by memory size and contention

- **Distributed memory systems**:
  - Each processor has local memory
  - Communication via message passing
  - Allows larger memory, but requires complex programming



Parallel architectures

3 Where do we solve such *large* problems?

Common parallel computing architectures:

- **Multicore processors**:
  - Multiple cores on single chip
  - Each core executes independent thread
- **Clusters**:
  - Interconnected computers (nodes)
  - Communication through network
- **Supercomputers**:
  - Extremely powerful systems
  - Thousands of processors working in parallel
  - Designed for high-speed complex calculations



The TOP500 list: top500.org

3 Where do we solve such *large* problems?

The **TOP500** list ranks the 500 most powerful supercomputers worldwide.

- Updated biannually (June and November)
- Ranks based on LINPACK benchmark performance
- Provides insights into trends in high-performance computing

Current leader (as of 2026): LineShine (China) with a performance of over 2 exaFLOP (10^{18} floating-point operations per second).



TOP500 List (June 2026) - Part 1

3 Where do we solve such *large* problems?

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**LineShineLingKun**, LX2 304C, NSCS, China	13,789,440	2,198.40	2,735.82	42,220
2	**El Capitan**, HPE Cray EX255a, AMD EPYC, DOE/NNSA/LLNL, United States	11,340,000	1,809.00	2,821.10	29,685
3	**Frontier**, HPE Cray EX235a, AMD EPYC, DOE/SC/ORNL, United States	9,066,176	1,353.00	2,055.72	24,607



TOP500 List (June 2026) - Part 2

3 Where do we solve such *large* problems?

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
4	**Aurora**, HPE Cray EX, Xeon Max 9470, DOE/SC/ANL, United States	9,264,128	1,012.00	1,980.01	38,698
5	**JUPITER Booster**, BullSequana XH3000, NVIDIA GH200, EuroHPC/FZJ, Germany	4,801,344	1,000.00	1,226.28	15,794
6	**HPC7**, HPE Cray EX255a, AMD EPYC, Eni S.p.A., Italy	3,461,472	571.50	861.13	8,735



TOP500 List (June 2026) - Part 3

3 Where do we solve such *large* problems?

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
7	**Eagle**, Microsoft NDv5, Xeon Platinum 8480C, Microsoft Azure, United States	2,073,600	561.20	846.84	—
8	**HPC6**, HPE Cray EX235a, AMD EPYC, Eni S.p.A., Italy	3,143,520	477.90	606.97	8,461
9	**Supercomputer Fugaku**, Fujitsu, A64FX, RIKEN CCS, Japan	7,630,848	442.01	537.21	29,899
10	**Alps**, HPE Cray EX254n, NVIDIA Grace, CSCS, Switzerland	2,121,600	434.90	574.84	7,124



High Performance Linpack (HPL) Benchmark

3 Where do we solve such *large* problems?

The computers in this table are ranked according to *Rmax*, the maximum sustained performance; but how is this measured? This is the High Performance Linpack (HPL) benchmark, which is run according to the following rules:

1. Generate a (random) linear system $Ax = b$ of size N and solve for x;
2. Measure the time for the solution process T and define a computation rate $R(N)$ according to the formula
$$R = \frac{2}{3}\frac{N^3}{T};$$
3. Let N grow and repeat the process, until you get the best possible execution rate value *Rmax*.



Importance of Linear Algebra in Benchmarking

3 Where do we solve such *large* problems?

Linear algebra problems have been used to benchmark supercomputers for a very long time, influencing their design in multiple ways.
Key observations:

- Supercomputers have a huge number of cores.
- Operating them consumes a lot of power.
- They are equipped with accelerators, specifically **graphical processing units** (GPUs).



Sustained Performance and Historical Context

3 Where do we solve such *large* problems?

The *sustained* rate of execution on the HPL benchmark shows that the number one machine, LineShine, is capable of executing 2.2×10^{18} arithmetic operations per second!

- **Linear algebra** is a primary tool for **benchmarking supercomputers**.
- *Dense linear algebra* problems are **compute-bound**, enabling hardware to operate close to peak performance.



Evolution of HPL Benchmark

3 Where do we solve such *large* problems?

The Linpack benchmark originated from tests in the LINPACK User's Guide [3].

- It has evolved into a standardized benchmark for comparing computing systems.
- The current HPL benchmark allows vendors to choose problem size and software configuration for optimal performance.
- Continuous interaction between supercomputing advances and linear algebra has driven innovations in algorithms and software.



The EuroHPC Joint Undertaking

3 Where do we solve such *large* problems?

EuroHPC JU is a European initiative to develop a world-class supercomputing ecosystem in Europe.

- Established in 2018
- Partnership between the European Union, European countries, and private sector
- Aims to provide access to high-performance computing resources for research, industry, and public sector

Key objectives:

- Deploy and operate supercomputers in Europe
- Foster research and innovation in HPC technologies
- Support development of HPC applications across various sectors

EuroHPC Supercomputers



EuroHPC
Joint Undertaking



The EuroHPC Pre-exascale & Exascale Machines

3 Where do we solve such *large* problems?

- **JUPITER** (Germany) - First European exascale system
  - NVIDIA GH200 Grace Hopper Superchip
  - 1000.00 PFlop/s (Rmax)
- **Alice Recoque** (France) - European exascale supercomputer
  - AMD Instinct MI430X GPUs & SiPEARL Rhea2
  - > 1 Exaflop/s (Planned)
- **LUMI** (Finland) - One of world's fastest and most energy-efficient
  - AMD Instinct MI250X GPUs
  - 379.70 PFlop/s (Rmax)
- **Leonardo** (Italy) - General-purpose HPC system
  - NVIDIA Ampere / Booster architecture
  - 241.20 PFlop/s (Rmax)
- **MareNostrum 5** (Spain) - Upgrade of iconic BSC system
  - NVIDIA Hopper GPUs
  - 457.10 PFlop/s (Rmax)



The EuroHPC Pre-exascale & Exascale Machines

3 Where do we solve such *large* problems?

- **JUPITER** (Germany) - First European exascale system
  - NVIDIA GH200 Grace Hopper Superchip
  - 1000.00 PFlop/s (Rmax)
- **Alice Recoque** (France) - European exascale supercomputer
  - AMD Instinct MI430X GPUs & SiPEARL Rhea2
  - > 1 Exaflop/s (Planned)
- **LUMI** (Finland) - One of world's fastest and most energy-efficient
  - AMD Instinct MI250X GPUs
  - 379.70 PFlop/s (Rmax)
- **Leonardo** (Italy) - General-purpose HPC system
  - NVIDIA Ampere / Booster architecture
  - 241.20 PFlop/s (Rmax)
- **MareNostrum 5** (Spain) - Upgrade of iconic BSC system
  - NVIDIA Hopper GPUs
  - 457.10 PFlop/s (Rmax)



The EuroHPC Petascale & Mid-range Machines

3 Where do we solve such *large* problems?

- **Arrhenius** (Sweden) - Powerful mid-range system
  - AMD Turin CPUs & NVIDIA Grace Hopper
  - Advanced AI and scientific workloads
- **Deucalion** (Portugal) - Multi-architecture petascale system
  - Fujitsu A64FX, AMD EPYC, and NVIDIA A100
  - 10 PFlop/s (Peak)
- **Daedalus** (Greece)
  - NVIDIA GH200 architecture
  - 115.08 PFlop/s (Rmax)
- **MeluXina** (Luxembourg) - Modular supercomputing architecture
  - BullSequana XH2000 / NVIDIA GPUs
  - 11.53 PFlop/s (Rmax)
- **Vega** (Slovenia) / **Karolina** (Czech Republic) / **Discoverer** (Bulgaria)
  - Regional petascale systems supporting EU research
  - Ranging from 3.0 to 9.13 PFlop/s (Rmax)



Table of Contents

4 What tools are we going to use?

- Introduction
  - Why high-performance Linear Algebra?
  - A gallery of problems
- What does it mean large-scale?
- Where do we solve such *large* problems?
  - The TOP500 list
  - The EuroHPC Joint Undertaking: www.eurohpc-ju.europa.eu
- What tools are we going to use?
  - (Modern) Fortran
  - Git
    - An example of Git usage



Programming Distributed Memory Systems

4 What tools are we going to use?

In this course, we focus on **distributed memory systems**:

- Most common in High-Performance Computing (HPC)
- Composed of many nodes, each with local memory
- Communication via message-passing libraries (e.g., MPI)

Before diving into the programming model, let's discuss the **tools** we'll use to write efficient parallel code.



Tools for High-Performance Linear Algebra

4 What tools are we going to use?

Modern Fortran

- Long-standing language for scientific computing
- Well-suited for numerical computations
- Still widely used in scientific applications

Software Version Control: git

- Track changes to code
- Collaborate effectively with others
- Essential for team development



Parallel Programming Tools

4 What tools are we going to use?

MPI, OpenMP, OpenACC, CUDA and other tools

- **MPI (Message Passing Interface)**: Distributed memory parallelism
- **OpenMP**: Shared memory parallelism for many-core processors
- **OpenACC and CUDA**: Accelerator/GPU programming
  - Modern supercomputers are equipped with GPUs
  - Essential for leveraging full system capabilities

Job Scheduler: Slurm

- Manage execution of jobs on the cluster
- Resource allocation and job queuing



What is Fortran?

4 What tools are we going to use?

Fortran (“Formula Translation”) is one of the oldest high-level programming languages:

- Originally developed in the 1950s by IBM
- Designed for scientific and engineering applications
- Easy translation of mathematical formulas into code

Modern versions include:

- Fortran 90, 95, 2003, 2008, 2018, 2023
- Features: modular programming, array operations, OOP, parallel computing

Note

Most concepts discussed can be ported to C/C++ or other compiled languages. For more on Fortran, see fortran-lang.org and [7].



Why Fortran for HPC?

4 What tools are we going to use?

Key strengths:

- **High performance** in numerical computations
- Highly optimized for array and matrix operations
- Efficient machine code generation
- Preferred choice for HPC applications

Programming paradigms supported:

- Procedural, modular, and object-oriented programming
- Parallel programming features (coarrays, MPI, OpenMP)
- Scalable code for distributed and shared memory systems

Applications: Climate modeling, computational fluid dynamics, numerical linear algebra



Fortran Compilers

4 What tools are we going to use?

Available compilers:

- **GNU Fortran** (`gfortran`) - Part of GCC
- **Intel Fortran** (`ifort`) - Optimized for Intel architectures
- **Cray Fortran** (`ftn`) - For Cray supercomputers
- **LLVM Fortran** (`flang`)
- **PGI Fortran** (`pgfortran`)
- **NAG compiler** (`nagfor`)

Our choice: `gfortran`

- Widely available and default on many systems
- Up to date with latest Fortran standards
- Free and open source



Installing gfortran

4 What tools are we going to use?

Checking installation

To check if `gfortran` is installed:

```
gfortran --version
```

Installation options:

- **Ubuntu/Debian:**
  `sudo apt-get install gfortran`
- **macOS (via Homebrew):**
  `brew install gcc`
- **Using Spack:** Download from spack.io or GitHub



Basic gfortran Usage

4 What tools are we going to use?

Basic compilation syntax:

```
gfortran -o output_file source_file.f90
```

Common options:

- `-o output_file` - Specify output executable name
- `-Wall` - Enable all compiler warnings
- `-g` - Generate debug information
- `-O0, -O1, -O2, -O3` - Optimization levels
- `-fcheck=all` - Enable runtime checks
- `-frecursive` - Enable recursion
- `-fPIC` - Position-independent code for shared libraries



Your First Fortran Program

4 What tools are we going to use?

Create a file `hello.f90`:

```
program hello
    use iso_fortran_env, only: output_unit
    implicit none
    write(output_unit,'("Hello, World!")')
end program hello
```

Compile and run:

```
gfortran -o hello hello.f90
./hello
```

Output:

```
Hello, World!
```

Note: `implicit none` enforces explicit variable declaration (good practice!)



What is Version Control?

4 What tools are we going to use?

Version control is a system that records changes to files over time:

- Recall specific versions later
- Track changes to files and code
- Enable collaboration without conflicts
- Revert to previous versions when needed

Benefits:

- Multiple developers work simultaneously
- Compare changes over time
- Collaborate more effectively



Types of Version Control Systems

4 What tools are we going to use?

- **Centralized Version Control Systems (CVCS)**
  - Central server stores the repository
  - Developers check out files from central location
  - Examples: Subversion (SVN), CVS
  - **Drawback:** Server failure stops all work
- **Distributed Version Control Systems (DVCS)**
  - Complete repository copy on each developer's machine
  - Enables offline work
  - Better collaboration capabilities
  - Examples: Git, Mercurial, Bazaar



What is Git?

4 What tools are we going to use?

Git is a distributed version control system:

- Created by Linus Torvalds in 2005 for Linux kernel development
- Designed for speed and efficiency
- Handles projects from small to very large

Key features:

- Track changes to files
- Collaborate with others
- Manage different versions of codebase
- Powerful branching and merging capabilities

Current status: *De facto* standard for version control in software development



Basic Git Workflow

4 What tools are we going to use?

Initialize a new repository:

```
git init my_project
cd my_project
```

Create or add files to the repository, e.g., `hello.f90` **Check repository status:**

```
git status
```

Stage changes for commit:

```
git add hello.f90
```

Commit changes with a message:

```
git commit -m "Add hello.f90 program"
```

View commit history:

```
git log
```



Cloning a Remote Repository or add a Remote

Clone a remote repository:

```
git clone <repository-url>
cd <repository-name>
```

Add a remote to an existing repository:

```
git remote add origin <repository-url>
```

Push local commits to remote:

```
git push origin main
```

Pull changes from remote repository:

```
git pull origin main
```

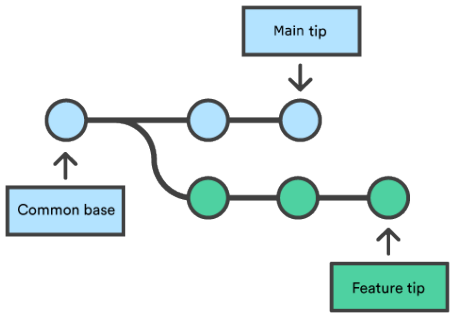


Branching and Merging

4 What tools are we going to use?

Branching allows you to create a separate line of development:

- Isolate features or bug fixes
- Experiment without affecting the main codebase



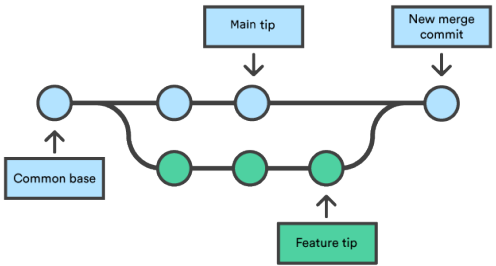


Branching and Merging

4 What tools are we going to use?

Merging combines changes from different branches:

- Integrate new features or fixes
- Resolve conflicts when changes overlap





Branching and Merging: Example

Create a new branch:

```
git checkout -b new-feature
```

Make changes and commit:

```
git add .
git commit -m "Implement new feature"
```

Switch back to main branch:

```
git checkout main
```

Merge changes from new-feature branch:

```
git merge new-feature
```



Services that use Git

4 What tools are we going to use?

Popular platforms for hosting Git repositories:

- **GitHub** (github.com)
- **GitLab** (gitlab.com)
- **Bitbucket** (bitbucket.org)
- **Codeberg** (codeberg.org)

We also have a Gitea instance installed at the Math Department: git.phc.dm.unipi.it.

Exercise

Explore the features of the mentioned Git hosting platforms and create an account on Codeberg. This will require you to setup SSH keys for secure access to your repositories.

```
ssh-keygen -t ed25519 -C "your_email@example.com"
ssh-add ~/.ssh/id_ed25519 && cat ~/.ssh/id_ed25519.pub
```

After you have done it, tell me the username and I'll add you to the course organization.



Summary

5 Summary

Key takeaways from this lecture:

- High-performance computing relies heavily on linear algebra
- The TOP500 list ranks the most powerful supercomputers using the HPL benchmark
- We will use Modern Fortran and Git for programming and version control

Next steps:

- Set up your Fortran development environment
- Familiarize yourself with Git and version control
- Explore CI/CD tools for automating workflows

Next lecture will cover: introduction to parallel computing from a theoretical standpoint, including models and paradigms, but in relation to linear algebra problems.



References

6 Bibliography

[1] S. J. Axler. *Linear Algebra Done Right*. Undergraduate Texts in Mathematics. New York: Springer, 1997. ISBN: 0387982582. URL: http://linear.axler.net/.

[2] J. W. Demmel. *Applied Numerical Linear Algebra*. Philadelphia, PA: SIAM, 1997. ISBN: 0-89871-389-7.

[3] J. J. Dongarra et al. *LINPACK User's Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1979. ISBN: 0-89871-172-X. DOI: 10.1137/1.9781611971811.

[4] G. H. Golub and C. F. Van Loan. *Matrix computations*. Fourth. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 2013, pp. xiv+756. ISBN: 978-1-4214-0794-4.

[5] R. A. Horn and C. R. Johnson. *Matrix analysis*. Second. Cambridge University Press, Cambridge, 2013, pp. xviii+643. ISBN: 978-0-521-54823-6.



References

6 Bibliography

[6] R. A. Horn and C. R. Johnson. *Topics in matrix analysis*. Corrected reprint of the 1991 original. Cambridge University Press, Cambridge, 1994, pp. viii+607. ISBN: 0-521-46713-6.

[7] M. Metcalf et al. *Modern Fortran Explained: Incorporating Fortran 2023*. 6th ed. Numerical Mathematics and Scientific Computation. Oxford, UK: Oxford University Press, 2024. ISBN: 978-0198876588.

[8] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997.