Course Introduction & Development Tools

Git, Python Environments & Scientific Computing Foundations

Week 1

Jimmy Jessen Nielsen
jjn@es.aau.dk

Dept. of Electronic Systems
Aalborg University

Course Introduction & Development Tools

Welcome to Numerical Scientific Computing!

- Instructor: Jimmy Jessen Nielsen (jjn@es.aau.dk)
- Course format: **Studio-based learning**
  - Active hands-on coding (not passive lectures!)
  - Everyone in one room, instructor available to help
  - Work at your own pace, collaborate with peers
- Three-stage mini-project evolving through the course
- Grading: Pass/Fail based on **Mini-Project Submissions**
  - Submit all 3 mini-projects with working code
  - Git history showing progressive development
  - Must be able to explain your own code
- Topics: Optimization, parallelization, GPU computing

Course Introduction & Development Tools

Today's Agenda

- **Welcome & Course Overview** (10 min)
- **Mindmap Exercise**: What is Scientific Computing? (15 min)
- **Studio Session 1**: Python Environment Setup (40 min)
  - Conda/mamba environments
  - Install course packages
- **10 MIN BREAK**
- **Studio Session 2**: Git Hands-On (50 min)
  - Version control fundamentals
  - Create repository, first commits
- **10 MIN BREAK**
- **Studio Session 3**: Begin Mandelbrot Implementation (60 min)
  - Understand algorithm
  - Start naive Python version
- **Peer Sharing**: Compare approaches (30 min)

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?
- Who read the slides?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?
- Who read the slides?
- Who installed Anaconda/miniconda/miniforge?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?
- Who read the slides?
- Who installed Anaconda/miniconda/miniforge?
- Who created GitHub account?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?
- Who read the slides?
- Who installed Anaconda/miniconda/miniforge?
- Who created GitHub account?
- Who has used Git before?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?
- Who read the slides?
- Who installed Anaconda/miniconda/miniforge?
- Who created GitHub account?
- Who has used Git before?
- Who has programmed in Python before?

Course Introduction & Development Tools

Pre-Lecture Preparation Check

Quick poll - raise your hand:

- Who listened to the podcast?
- Who watched the video?
- Who read the slides?
- Who installed Anaconda/miniconda/miniforge?
- Who created GitHub account?
- Who has used Git before?
- Who has programmed in Python before?
- Who has programmed in other languages? (Which ones?)

Agenda

Course Overview

Mindmap Exercise

Development Environment Setup

Python Environment Setup

Break

Version Control with Git

Break

Mandelbrot Set Introduction

Peer Sharing & Wrap-up

Course Overview

Course Overview

Studio Format - What's Different?

- **Traditional course:**
  - Instructor lectures for 90 minutes
  - Students do exercises scattered across campus
  - Hard for instructor to help everyone
- **Our studio format:**
  - Short concept recaps (5-10 min)
  - Extended hands-on coding (90+ min per session)
  - Everyone in same room with instructor
  - Work at your own pace
  - Instructor circulates, helps individually
  - Learn from peers
- **Preparation is essential!**
  - Podcast (commute listening)/Video/Slides: Big picture concepts
  - Reading (45-60 min): Technical details
  - Can't participate meaningfully without preparation

Course Overview
Learning Goals

By the end of this course, you will be able to:

- **Optimize** scientific code for performance
  - Profiling and benchmarking
  - NumPy vectorization, Numba compilation
- **Parallelize** computations across multiple cores
  - Multiprocessing, load balancing
  - Understand Amdahl's and Gustafson's laws
- **Distribute** work across clusters
  - Dask for distributed computing
  - Understand distributed memory concepts
- **Accelerate** with GPUs
  - CuPy for GPU array operations
  - Understand when GPU helps vs. doesn't
- **Develop** professional scientific software
  - Version control (Git), testing, documentation

Course Overview

Grading: Mini-Project Submissions (Pass/Fail)

Pass Requirements:

Submit all 3 mini-projects to Moodle with:

- **Working code** implementing required features
- **GitHub repository URL** showing progressive development
- **Git history** with regular commits (not all at deadline!)
- **Performance measurements** and reflection document
- **You must be able to explain your own code!**

We encourage you to:

- Attend and actively engage in studio sessions
- Ask questions, help peers, collaborate
- Use instructor support during hands-on work
- Participate in peer review sessions

Course Overview
Mini-Projects Overview

All projects implement **Mandelbrot fractal** - progressively optimized:

MP1: Foundations & Optimization (Weeks 1-5)

- Naive Python → NumPy → Numba → Data types
- Due: Week 5

MP2: Parallelization & Distribution (Weeks 5-8)

- Multiprocessing → Dask (local & cluster)
- Due: Week 8

MP3: GPU & Quality (Weeks 8-11)

- CuPy GPU acceleration → Testing → Documentation
- Due: Week 11

Each includes:

- Physical peer review session (learn from others)
- Online Moodle review (constructive feedback)
- Reflection on what you learned

Course Overview
Chatbot Usage Policy

AI tools like ChatGPT can help, but have rules:

ENCOURAGED:

- Syntax questions: "How do I create a NumPy array?"
- Debugging specific errors
- Learning new library APIs
- Generating test cases

PROHIBITED:

- Generating complete solutions
- Designing your optimization approach without understanding
- Explaining performance results you don't understand

REQUIRED:

- Mark AI-assisted code with comments
- You must be able to explain any code you submit
- Failure to explain = not understanding your work

You must be able to explain your code! Don't copy without understanding.

Mindmap Exercise

Mindmap Exercise

What is Scientific Computing?

Activity (15 minutes total):

Step 1: Individual Brainstorm (5 min)

- Think: "What words/concepts come to mind when you hear 'scientific computing'?"
- Also: "What do you hope to learn in this course?"
- Write down 5-10 words or short phrases

Step 2: Submit to Word Cloud (3 min)

- Go to Mentimeter [link shown on screen]
- Enter your words/phrases
- Watch real-time word cloud appear!

Step 3: Discussion (7 min)

- Instructor highlights interesting patterns
- Address common themes and expectations
- Calibrate course content to student interests

Purpose: Activate your prior knowledge & surface expectations

Mindmap Exercise

What is Scientific Computing? (Instructor Notes)

Common themes you might see:

- **Disciplines:** Physics, biology, chemistry, engineering, AI/ML
- **Methods:** Simulation, modeling, data analysis, visualization
- **Tools:** Python, MATLAB, C++, clusters, GPUs
- **Concepts:** Algorithms, performance, parallelism, big data
- **Challenges:** Speed, accuracy, scalability, reproducibility

Scientific computing is:

- Using computers to solve scientific problems
- Cross-disciplinary: Math + Science + Computer Science
- About **performance** (fast enough for research)
- About **correctness** (accurate results)
- About **reproducibility** (others can verify)

We'll revisit this word cloud in the final lecture to see how your understanding evolved!

Development Environment Setup

Development Environment
Terminal/Command Line Basics

Opening the terminal:

- **Windows:** Command Prompt (Kommandoprompt) or PowerShell
- **macOS/Linux:** Terminal (Terminal)

Basic navigation:

```
pwd                     # Print working directory (where am I?)
ls                      # List files (dir on Windows)
cd foldername           # Change directory
cd ..                   # Go up one level
mkdir foldername        # Create new folder
```

Recommended folder structure:

```
Documents/
  nsc-course/
    mandelbrot-nsc/          # Git repository
      environment.yml
      mandelbrot.py
      test_install.py
```

Important: Run `mamba env export` and all `git` commands from inside `mandelbrot-nsc/`

Python Environment Setup

Python Environment

Why Virtual Environments?

- **Problem:** Different projects need different package versions
  - E.g., Project A needs NumPy 1.24, Project B needs NumPy 1.26
  - Installing globally breaks compatibility
- **Solution:** Virtual environments (isolated package installations)
- **Tools available:**
  - **Anaconda/Miniconda**: Full or minimal conda distribution
  - **Miniforge/mamba**: Faster alternative (**RECOMMENDED**)
  - **pip + venv**: Built-in Python (**NOT recommended for scientific computing**)

Important for NSC:

- Conda/mamba include **optimized math libraries**
- Intel MKL (Intel CPUs), Apple Accelerate (macOS), OpenBLAS (Linux/others)
- NumPy from pip uses generic BLAS - can be **2-10× slower** for matrix operations
- Scientific packages benefit significantly from conda's optimized builds
- Use **mamba** or conda for best performance

VS Code users:

- VS Code works perfectly with mamba/conda environments!
- Install Miniforge separately, then select environment in VS Code
- Python extension detects conda/mamba environments automatically

Python Environment
Creating Course Environment

Recommended: With mamba (fastest):

```
# Install miniforge if you don't have mamba
# Download from: https://github.com/conda-forge/miniforge

# Create environment named 'nsc2026' with Python 3.11
mamba create -n nsc2026 python=3.11

# Activate environment
mamba activate nsc2026  # or: conda activate nsc2026

# Install optimized numerical packages
mamba install numpy matplotlib scipy numba pytest dask

# Verify optimized BLAS - look for: openblas, mkl, or ACCELERATE
python -c "import numpy; numpy.show_config()" | grep -iE "openblas|mkl|ACCELERATE"
# Should see at least one match. No output = not optimized!
```

Alternative: With conda (slower install, same performance):

```
conda create -n nsc2026 python=3.11
conda activate nsc2026
conda install numpy matplotlib scipy numba pytest dask
```

Why Python 3.11? Stable, well-tested. Python 3.12+ may have package compatibility issues.

Python Environment

Studio Session 1 - Setup (40 min)

Task: Create and configure course environment

Step 1: Create environment (10 min)

1. Choose your tool: **mamba** or conda
2. Create environment named `nsc2026`
3. Activate it
4. Verify: `python --version`

Step 2: Install core packages (15 min)

1. numpy, matplotlib, scipy
2. numba (for JIT compilation)
3. pytest, pytest-cov (for testing)
4. dask, distributed (for parallel computing)
5. Handle any installation errors (ask for help!)

Step 3: Test installation (10 min)

1. Create test script: `test_install.py`
2. Import all packages, print versions
3. Run script, verify no errors
4. Commit to Git later!

Step 4: Export environment (5 min)

1. Save for reproducibility:
2. `mamba env export > environment.yml`
3. (or `conda env export > environment.yml`)
4. Save this file to commit to Git later!

Note: We recommend **mamba** for best performance!

Python Environment
Export Environment (Best Practice)

Create environment.yml file:

```
# Export environment specification
mamba env export > environment.yml
# (or: conda env export > environment.yml)

# Or manual creation:
# environment.yml
name: nsc2026
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy
  - matplotlib
  - scipy
  - numba
  - pytest
  - pytest-cov
  - dask
  - distributed
```

Benefits:

- Others can recreate the exact same environment: `mamba env create -f environment.yml`

Python Environment

IDEs - Choose Your Tool

- **Spyder** (`https://www.spyder-ide.org`)
  - MATLAB-like interface
  - Variable explorer, inline plots
  - Integrated profiler, debugger
  - Good for scientific computing
  - Included with Anaconda or as standalone app
- **VS Code** (`https://code.visualstudio.com`)
  - Lightweight, fast, extensible
  - Excellent Python extension
  - Git integration built-in
  - Most popular among developers
- **JupyterLab**
  - Browser-based notebooks
  - Great for exploration, visualization
  - Good for reports with code + narrative
  - Will use later in course

Recommendation: Try Spyder first (easy), then VS Code if you want more features

10 Minute Break

First break - stretch, grab coffee, return refreshed!

Next: Git Hands-On

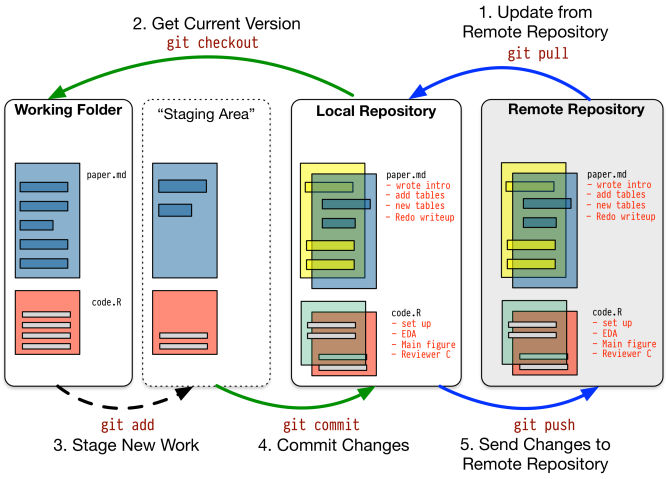
Version Control with Git

Version Control
Why Git Matters

- **Problem without version control:**
  - Files: `code.py`, `code_v2.py`, `code_final.py`, `code_final_FINAL.py`
  - Which version works? What changed between versions?
  - Lose your work if computer crashes
  - Hard to collaborate with others
- **Git solves these problems:**
  - Track every change with meaningful messages
  - Revert to any previous version
  - Backup on GitHub (cloud storage)
  - Collaborate without conflicts
  - Professional standard in industry & research
- **Essential for this course:**
  - Submit mini-projects via Git repository
  - Track your progress week-by-week
  - Share code during peer reviews
  - Good practice for MSc thesis & career

Version Control

Git Workflow - Basic Commands



Version Control

Git Workflow - Basic Commands

Essential commands:

```
git init                        # Initialize repository
git add filename.py             # Stage file for commit
git commit -m "message"         # Save snapshot with message
git status                      # Check what's changed
git log                         # View commit history
git push                        # Upload to GitHub
```

Commit messages should be descriptive!

- **Bad:** "fixed stuff", "v2", "asdf"
- **Good:** "Implement naive Mandelbrot", "Fix convergence test", "Optimize loop"

Version Control
Git Best Practices

- **Commit often, commit early**
  - Small, focused commits (not one huge commit)
  - Commit when feature works, before breaking it
  - Think: "Could I explain this change?"
- **Write good commit messages**
  - First line: Short summary (< 50 chars)
  - Explain **what** and **why**, not how
  - Example: "Add docstrings to mandelbrot functions for MP1"
- **Use .gitignore**
  - Don't commit: `*.pyc`, `__pycache__`, `.ipynb_checkpoints`
  - Don't commit: Large data files, temporary outputs
  - Do commit: Source code, documentation, small test files
- **Push regularly to GitHub**
  - Backup in the cloud
  - Can access from any computer
  - Easy sharing for peer reviews
- **GUI tools available:**
  - GitHub Desktop, VS Code built-in Git, GitKraken, and many others

Version Control

Studio Session 2 - Git Hands-On (50 min)

Task: Create Git repository for mini-project

Step 1: Initialize repository (5 min)

1. Create folder: `mandelbrot-nsc`
2. `cd mandelbrot-nsc`
3. `git init`
4. Verify: `git status`

Step 2: Copy environment file (5 min)

1. Copy `environment.yml` from Python session to this folder
2. Verify it's there: `ls`

Step 3: First commit (10 min)

1. `git add environment.yml`
2. `git commit -m "Add Python environment specification"`
3. `git log` to see commit

Version Control

Example: Python File Skeleton


```
"""
Mandelbrot Set Generator

Author: [Your Name]
Course: Numerical Scientific Computing 2026
"""

def f(x):
    """
    Example function.

    Parameters
    ----------
    x : float
        Input value

    Returns
    -------
    float
        Output value
    """
    # TODO: Implement the algorithm
    pass
```

Version Control
Studio Session 2 - Continued

Step 4: Second commit - Python skeleton (10 min)

1. Create `mandelbrot.py` (example on previous slide)
2. `git add mandelbrot.py`
3. `git commit -m "Add mandelbrot skeleton"`
4. `git log` - see both commits!

Step 5: Push to GitHub (15 min)

1. Go to GitHub.com
2. Create new repository: `mandelbrot-nsc`
3. **GitHub shows commands - copy them!**
4. Paste in terminal (from `mandelbrot-nsc/`):

```
git remote add origin <URL>
git branch -M main
git push -u origin main
```

Step 6: Practice the cycle (5 min)

1. Edit `mandelbrot.py` (add a comment)
2. `git add mandelbrot.py`
3. `git commit -m "Add TODO comment"`
4. `git push`

Troubleshooting:

- `git status` / `git log`
- `git remote -v`
- Google error messages!

10 Minute Break

Second break - stretch and refresh!

Next: Start Mandelbrot Implementation

Mandelbrot Set Introduction

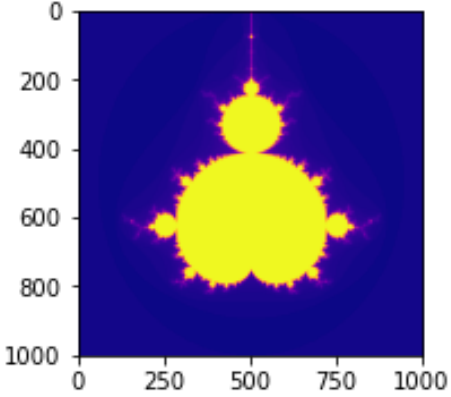
Mandelbrot Set

What is it?

- Famous fractal discovered by Benoit Mandelbrot (1980)
- Mathematical set in complex plane
- Beautiful self-similar patterns
- Infinite detail at any zoom level

Mathematical definition:

- For each complex number c
- Iterate: $z_{n+1} = z_n^2 + c$
- Start with $z_0 = 0$
- If $|z_n|$ stays bounded, c is in the set
- Color by number of iterations



Example Mandelbrot set visualization

Mandelbrot Set

Why Use This for the Course?

- **Computationally intensive**
  - Naive version is slow → need optimization!
  - Obvious performance improvements
  - Easy to measure speedup
- **Embarrassingly parallel**
  - Each pixel independent
  - Perfect for multiprocessing, GPU, and distributed computing
  - Can distribute across cluster
- **Visual feedback**
  - Immediately see if code works (pretty picture!)
  - Easy to debug (if image wrong, something's wrong)
  - Motivating to see results
- **Progressive enhancement**
  - Week 1: Naive loops → NumPy
  - Week 2-3: Numba, data types
  - Week 4-5: Multiprocessing, Dask
  - Week 9-10: GPU acceleration
  - Same problem, many optimization strategies!

Mandelbrot Set
The Algorithm

Algorithm (pseudocode):

1. Define region: $x \in [-2, 1]$, $y \in [-1.5, 1.5]$ (typical view)
2. Create grid of complex numbers c over this region:
   - Generate `width` evenly spaced x-values from x_{min} to x_{max}
   - Generate `height` evenly spaced y-values from y_{min} to y_{max}
   - Form complex numbers: $c = x + iy$ for each grid point
3. For each point c in grid:
   - Initialize $z_0 = 0$
   - For $n = 0$ to max_iter:
     - Compute $z_{n+1} = z_n^2 + c$
     - If $|z_{n+1}| > 2$: Point escapes! Store n, break to next point
   - If loop completes: Point is in set, store max_iter
4. Return 2D array of iteration counts
5. Visualize with colormap

Key details:

- Threshold $|z| > 2$ guarantees divergence
- Typical: $max\_iter = 100$, resolution: 1024×1024 or larger

Mandelbrot Set

Studio Session 3 - Begin Implementation (60 min)

Task: Implement naive Mandelbrot (or get started)

Step 1: Understand the algorithm (10 min)

- Review pseudocode on previous slide
- Discuss with neighbor: "What does each part do?"
- Ask questions if confused!

Step 2: Implement `mandelbrot_point` function (20 min)

- Takes single complex number c
- Returns iteration count
- Test with known points (e.g., $c = 0$ should be max_iter)
- **Commit to Git** with descriptive message

Step 3: Implement `compute_mandelbrot` grid version (20 min)

- Create mesh of complex numbers (**Hint:** use `numpy.linspace` and nested loops)
- Loop over all points, call `mandelbrot_point` for each
- Return 2D array of iteration counts
- **Start with small grid (e.g., 100×100) for testing**
- **Commit to Git** with descriptive message

Mandelbrot Set

Studio Session 3 - Continued

Step 4: Measure execution time (10 min)

- Use `time.time()` or `timeit` to measure performance
- Try grid size 1024×1024 for meaningful baseline
- **Commit to Git** with descriptive message

Example code:

```
import time
start = time.time()
result = compute_mandelbrot(-2, 1, -1.5, 1.5, 1024, 1024)
elapsed = time.time() - start
print(f"Computation took {elapsed:.3f} seconds")
```

Step 5: Visualize the result (10 min)

- Use `plt.imshow()` to display as image
- Try colormaps: 'hot', 'viridis', 'twilight'
- Add colorbar, title. Save with `plt.savefig()`

Important: It's OK if you don't finish! Continue at home.

Mandelbrot Set
Tips & Common Issues

- **Issue:** "My implementation is very slow!"
  - That's expected! Naive version uses Python loops
  - Start with small grid (100×100) for testing
  - Increase size once it works
  - We'll optimize in coming weeks!
- **Issue:** "My image is all one color"
  - Check: Are you assigning to result[i,j]?
  - Check: Are you breaking out of loop when $|z| > 2$?
  - Try printing some iteration values
- **Issue:** "Image is upside-down or flipped"
  - Check your x and y indexing
  - Try: `plt.imshow(..., origin='lower')`
- **Issue:** "TypeError or NameError"
  - Check imports: `import numpy as np`
  - Check function definition matches call
  - Read error message carefully!

Peer Sharing & Wrap-up

Peer Sharing
Compare Approaches (30 min)

Activity: Form pairs or groups of 3

Step 1: Share your progress (10 min per person)

- Show your code (even if incomplete!)
- Run and display Mandelbrot image
- Explain your approach
- What worked? What was challenging?

Step 2: Compare implementations (10 min)

- Different variable names?
- Different loop structures?
- Different convergence tests?
- Any clever optimizations already?

Step 3: Help each other (optional)

- If one person finished, help the other
- Debug issues together
- Share tips and tricks

Wrap-up

What We Accomplished Today

- **Course structure understanding**
  - Studio format, grading, mini-projects
  - Chatbot policy, expectations
- **Mindmap exercise**
  - Surfaced prior knowledge
  - Set expectations for course
- **Python environment**
  - Created isolated environment
  - Installed course packages
  - Exported environment.yml
- **Git version control**
  - Created repository
  - First commits
  - Pushed to GitHub
- **Mandelbrot implementation**
  - Understood algorithm
  - Started naive implementation
  - (Some of you finished!)

Wrap-up
Before Next Lecture

Complete at home:

1. Finish naive Mandelbrot implementation
   - **Keep it simple!** Use plain Python loops - no optimization yet
   - This is your un-optimized baseline for comparison
   - Get it working and visualizing correctly - test with different resolutions
   - Commit to Git with good message
2. Measure execution time
   - Use `time.time()` (or `timeit`)
   - Document baseline: "Grid 1024×1024 takes X seconds"
   - This baseline is crucial for future comparisons!
3. Prepare for L2: Computer Architecture & Memory
   - Choose one format: Podcast, Video, or Slides
   - Review assigned reading materials
   - Details on Moodle!
4. Optional: Explore interesting regions
   - Try zooming into fractal (change xmin, xmax, etc.)
   - Try different max_iter values
   - Have fun with it!

Next Lecture
Preview: Computer Architecture

Next Lecture (Week 2)

- Topics:
  - Memory hierarchy (registers, cache, RAM)
  - Why memory access patterns matter
  - Multi-core systems basics
  - NumPy vectorization
- Studio activities:
  - Memory access experiments
  - Convert Mandelbrot to NumPy (vectorized)
  - Measure speedup vs. naive version
  - Profiling introduction

Preparation is essential! Watch videos, read book sections.

Questions?

Contact:
Jimmy Jessen Nielsen
jjn@es.aau.dk

Git Help: `https://git-scm.com/doc`
Conda Help: `https://docs.conda.io/`

See you next week with working Mandelbrot code!