Lecture 01 · Introduction to Modern Simulation Software Development

Detailed Notes

1 Motivation

- **Complexity of Modern Simulation Software**: Developing industry-ready computational science and engineering (CSE) software, such as FEM-based tools, requires handling realistic geometries, multiphysics, efficient solvers, adaptive meshes, and scalability. Building such software from scratch would require approximately 100k+ lines of code, highlighting the need for collaborative and modular development approaches.
- **Challenges in Academic Software Development**: Research software in academia is often developed by graduate students and maintained by postdocs, who may lack software engineering experience and have limited time. Faculty advisors, who also often lack software expertise, provide limited oversight. This results in software that is primarily viewed as a tool for publishing rather than a sustainable, high-quality product.
- **Industry vs. Academia Workflows**: Industry prioritizes ready-made solutions, integration with existing tools, and robustness, while academia focuses on experimentation and publishing new methods. Bridging this gap requires software that is both flexible for research and reliable for industrial applications.
- **Sustainability and Collaboration**: Collaboration, modular design, and community involvement are essential to creating software that can evolve and remain relevant over decades. Collaboration is only possible when code is well-structured, testable, documented.

2 Course Goals

The course aims to introduce students to modern numerical methods and software development practices used in cutting-edge research. Key objectives include:

- Understanding advanced numerical methods for solving PDEs (e.g., finite differences, finite elements, discontinuous Galerkin, particle-based methods).
- Learning modern software development practices such as version control, testing, and continuous integration.
- Gaining hands-on experience with open-source frameworks like FEniCSx and Trixi.jl for setting up numerical simulations.
- Exploring applications in thermal transfer, fluid mechanics, gas dynamics, and uncertainty quantification.

3 Lecturers

In the summer semester of 2026, the course is taught by:

- **Dr. Lambert Theisen** (PhD 2024): Specializes in finite element methods (FEM), domain decomposition methods, linear system preconditioning.
- **Dr. Georgii Oblapenko** (PhD 2017): Focuses on models for multi-species reacting flows, finite volume methods (FVM), discontinuous Galerkin (DG) methods, particle methods for rarefied flows, and uncertainty quantification (UQ) for supersonic flows.

4 Organizational Details

- **Lectures**: Held every Wednesday from 10:30 to 12:00.
- **Exercises**: Schedule to be determined (TBD).
- **Projects**: Three projects will be assigned throughout the semester to apply the concepts learned in lectures.

5 What is "Modern" Simulation Software Development?

Modern simulation software development leverages advanced tools and methodologies to streamline the creation, testing, and deployment of simulation software. This includes:

- **Verifiable code and reproducible research**: Extensive testing and verification of code, versioning of code to ensure correctness and reproducibility of results
- **Automated workflows**: Integrating continuous integration and deployment (CI/CD) pipelines to ensure code quality and reproducibility.
- **Collaborative tools**: Using version control systems (e.g., Git) and platforms like GitHub/GitLab to facilitate teamwork and code sharing.

6 Capabilities of Modern Simulation Software

Modern simulation software is characterized by its ability to handle complex and diverse engineering problems. Key capabilities include:

- **Multi-physics**: Simulating interactions between different physical phenomena (e.g., fluid-structure interaction, thermo-mechanical coupling, phase transitions, etc.).
- **High-order methods**: Using high-order numerical schemes to achieve greater accuracy and efficiency.
- **Hybrid methods**: Combining different numerical methods (e.g., FEM with particle methods) to facilitate multi-physics simulations.
- **Uncertainty quantification**: Incorporating methods to assess and manage uncertainties in simulation inputs and outputs, provide safety tolerances, assess impact of parameter uncertainties on solution quality.

7 Software Aspects

Modern simulation software emphasizes robust and maintainable code development practices:

- **Unit tests**: Testing individual components of the code to ensure they function correctly in isolation.
- **Reproducible research**: Ensuring that simulations can be replicated and verified by others.
- **Modular design**: Structuring code into reusable and interchangeable modules.
- **Exascale-ready**: Efficiently supporting parallel computing technologies like MPI and GPU acceleration.
- **Mixed precision**: Utilizing different levels of numerical precision to optimize performance and accuracy.
- **Automated code generation**: Using tools for automatic differentiation (AD) and code generation to reduce manual coding errors.

8 Verification and Validation

- **Verification**: Ensures that the equations are solved correctly. This involves both mathematical correctness (e.g., using appropriate numerical methods) and programming correctness (e.g., implementing the methods accurately).
- **Validation**: Ensures that the correct equations are solved, addressing the physical and engineering relevance of the model. This requires experimental data, a ground truth model, or a combination thereof!

9 Code Verification

Code verification involves several levels of testing to ensure the reliability and correctness of the software:

9.1 Unit Testing

Tests individual units of code (e.g., functions, classes) to verify their correctness. Example:

```
E0 = energy(particles)  # assumes energy has already been tested
collide_2particles!(particles, collision_parameters, i1, i2)
@test isapprox(energy(particles), E0; atol=eps(), rtol=eps()) == true
```

9.2 Integration Testing

Tests the interaction between different units to ensure they work together as intended. Example:

```
E0 = energy(particles)  # assumes energy has already been tested
collide_all_particles!(particles, collision_parameters)
@test isapprox(energy(particles), E0; atol=eps(), rtol=eps()) == true
```

9.3 Regression Testing

Ensures that existing functionality continues to work as expected after changes or updates. Example:

```
heat_flux = simulation_result()
reference_heat_flux = parse(Float64, read("reference_q.txt", String))
@test isapprox(heat_flux, reference_heat_flux; atol=eps(), rtol=eps()) == true
```

10 Continuous Integration (CI): example 1 (Github Actions)

CI involves automating the build and test processes to ensure that code changes do not introduce errors. CI pipelines are typically integrated with version control systems and run automatically when code is committed. Example CI setup using Github Actions (from the Merzbild.jl code of Georgii Oblapenko):

```
name: Tests

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

# needed to allow julia-actions/cache to delete old caches that it has created
permissions:
  actions: write
  contents: read

jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        julia-version: ['lts', '1', 'pre']
        julia-arch: [x64, x86]
        os: [ubuntu-latest, windows-latest]

    steps:
      - uses: actions/checkout@v4
      - uses: julia-actions/setup-julia@v2
        with:
          version: ${{ matrix.julia-version }}
          arch: ${{ matrix.julia-arch }}
      - uses: julia-actions/cache@v2
      - uses: julia-actions/julia-buildpkg@v1
      - uses: julia-actions/julia-runtest@v1
      - uses: julia-actions/julia-processcoverage@v1
      - name: Upload results to Codecov
```

```
        uses: codecov/codecov-action@v5
        with:
            fail_ci_if_error: true
            token: ${{ secrets.CODECOV_TOKEN }}
```

This sets up to run tests on 2 operating systems (Ubuntu and Windows) using 3 Julia versions (LTS, latest stable, and pre-release) compiled for 2 architectures (x64 and x86). The tests are run and code coverage statistics are uploaded to codecov.io. The tests are run every time a pull request to the `main` branch receives a commit, or the `main` branch receives a commit directly. So a pull request where tests fail (code produces errors or new code is not covered by tests) will display a warning.

11 Continuous Deployment (CD)

CD automates the release process, ensuring that new features and bug fixes are deployed reliably and frequently. CD can include:

- Automatic deployment of documentation websites (documentation a mix of docstrings in code and additional documentation).
- Analysis of code coverage to ensure comprehensive testing (see above).
- Archiving releases to platforms like Zenodo for persistent DOIs.

The effectiveness of CD depends on the quality and coverage of the tests.

Example of documentation CD from Merzbild.jl:

```
name: Documentation

on:
  push:
    branches:
      - main # update to match your development branch (master, main, dev, trunk, ...)
    tags: '*'
  pull_request:
    branches:
      - main

jobs:
  build:
    permissions:
      actions: write
      contents: write
      pull-requests: read
      statuses: write
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: julia-actions/setup-julia@v2
        with:
          version: '1'
      - uses: julia-actions/cache@v2
```

```
      - name: Install dependencies
        run: julia --project=docs/ -e 'using Pkg; Pkg.develop(PackageSpec(path=pwd())); Pkg.in
      - name: Build and deploy
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # If authenticating with GitHub Actions to
        run: julia --project=docs/ docs/make.jl
```

12 Continuous Integration (CI/CD): example 2 (Gitlab)

Example CI/CD setup using Gitlab (adapted from the fenicsR13 code of Lambert Theisen), this both tests the code and builds and deploys documentation:

```
variables:
  APP_DIRECTORY: .
  DOCS_DIRECTORY: ${APP_DIRECTORY}/docs
  DOCS_LATEX_NAME: fenicsr13
stages:
- prepare
- build
- test
- deploy

# ****************************************************************************** #
# prepare
# ****************************************************************************** #

prepare:docker:
  stage: prepare
  image: docker:26.1.4
  services:
    - docker:26.1.4-dind
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  before_script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker pull $CI_REGISTRY_IMAGE:latest || true
    - docker pull $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA || true
    - docker pull $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME || true
    - docker build
      --cache-from $CI_REGISTRY_IMAGE:latest
      --tag $CI_REGISTRY_IMAGE:latest
      --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_NAME .
    - docker push $CI_REGISTRY_IMAGE:latest
  only:
    - master
    - tags
  tags:
    - dockerindocker
```

```
# ****************************************************************************** #
# build
# ****************************************************************************** #

build:docs:
  stage: build
  dependencies:
    - prepare:docker
  image:
    name: $CI_REGISTRY_IMAGE:latest
    entrypoint: [""]
  script:
    - cd ${DOCS_DIRECTORY}
    - sphinx-apidoc -f -o source/fenicsR13 ../fenicsR13
    - sphinx-apidoc -f -o source/tests/2d_heat ../tests/2d_heat
    - sphinx-apidoc -f -o source/tests/2d_stress ../tests/2d_stress
    - sphinx-apidoc -f -o source/tests/2d_r13 ../tests/2d_r13
    - sphinx-apidoc -f -o source/tests/3d_heat ../tests/3d_heat
    - sphinx-apidoc -f -o source/tests/3d_stress ../tests/3d_stress
    - sphinx-apidoc -f -o source/tests/3d_r13 ../tests/3d_r13
    - sphinx-apidoc -f -o source/examples ../examples
    - make html
    - make latex
  artifacts:
    paths:
      - ${DOCS_DIRECTORY}/_build/html/
      - ${DOCS_DIRECTORY}/_build/latex/
    expire_in: 6 month
  tags:
    - docker

# ****************************************************************************** #
# test
# ****************************************************************************** #

.test: # dot means "hidden", acts as base class
  stage: test
  dependencies:
    - prepare:docker
  before_script:
    - pip install -e . # local install to have right coverage
  image:
    name: $CI_REGISTRY_IMAGE:latest
    entrypoint: [""]
  tags:
    - docker

test:flake8:
  extends: .test
  script:
```

```
    - cd ${APP_DIRECTORY}
    - flake8 fenicsR13 examples tests

test:pytest:
  extends: .test
  script:
    - cd ${APP_DIRECTORY}
    - pytest -v --durations=10
  after_script:
    - mkdir convergence_plots
    - cp tests/*/*/convergence_plot* convergence_plots/
    - mkdir errors
    - for file in tests/*/*/errors.csv; do foldername=${file%/errors.csv}; casename=$(basename
    # - for file in tests/*/*/errors.csv; do foldername=${file%/errors.csv}; casename=$(basena
  coverage: '/TOTAL.*? (100(?:\.0+)?\%|[1-9]?\d(?:\.\d+)?\%)$/'
  artifacts:
    when: always
    paths:
      - ${APP_DIRECTORY}/convergence_plots
      - ${APP_DIRECTORY}/errors
      - ${APP_DIRECTORY}/htmlcov
    expire_in: 6 month

# ************************************************************************** #
# deploy
# ************************************************************************** #

deploy:latex:
  stage: deploy
  image:
    # name: blang/latex:ctanfull # only TexLive 2017
    name: aergus/latex:latest
  dependencies:
    - build:docs
  script:
    - cd ${DOCS_DIRECTORY}/_build/latex/
    - make
  artifacts:
    paths:
      - ${DOCS_DIRECTORY}/_build/latex/${DOCS_LATEX_NAME}.pdf
    expire_in: 6 month
  only:
    - master
    - tags
  tags:
    - docker

pages:
  stage: deploy
  dependencies:
    - build:docs
```

```yaml
script:
  - mv ${DOCS_DIRECTORY}/_build/html/ ${CI_PROJECT_DIR}/public/
artifacts:
  paths:
    - public
only:
  - master
  - tags
tags:
  - shell
```

13 Examples of Open-Source Codes by course lecturers

- **fenicsR13** (Lambert Theisen): A FEM code based on FEniCSx for solving the linear R13 equations. Uses GitLab CI for continuous integration.
- **Merzbild.jl** (Georgii Oblapenko): A DSMC code for variable-weight particles, implemented in Julia. Uses GitHub CI for continuous integration.

14 Examples of other codes that do it "right"

- Deal.II - highly scalable FEM code
- AMReX - block-structured meshes with adaptive mesh refinement
- t8code - hybrid meshes with adaptive mesh refinement
- DifferentialEquations.jl - library for ODE solvers
- Trixi.jl - DGSEM code for solving hyperbolic PDEs

15 Sustainable Computational Engineering

The course aligns with broader initiatives in sustainable computational engineering, such as the course "Sustainable Computational Engineering" (42.00019) from MBD. This course focuses on:

- Continuous Integration (CI)
- Research Data Management (RDM)
- Licensing and legal aspects of software development

16 Key Take-Aways

1. **Learning from Mistakes**: The second code you write is often the worst because you aim to avoid all the mistakes from your first attempt. Embrace iterative improvement.
2. **Testing and Documentation**: These should not be afterthoughts but integral parts of the development process.
3. **Modularization**: Avoid monolithic functions (e.g., a `solve!()` function with 10,000 lines). Break down the code into manageable, reusable modules.

17 References

- How do we make large opensource projects sustainable? (lessons learned from 25 years of deal.II)