1. Team Git Merge Excercise

This document is written to guide your team through the workshop step by step. In this workshop you will be copying provided code snippets into the repository files, you will not be writing the code from scratch.

- Note: It will be very important to follow the instructions here carefully and sequentially. If you miss a step or do something out of order, you may have trouble completing the exercise.

1.1. Overview

One teammate should fork the given repo and invite the others. Each student takes one feature branch (A, B, C, D) and will copy the provided code for their feature into the repo. An instructor feature branch will then introduce an intentional change that causes merge conflicts. Your job is to use git workflows (branching, merge, PRs) to resolve those conflicts, pass the tests, and merge everything into `main`.

These are the day-to-day workflows used in real software teams. Knowing how to merge, reason about conflicts, and keep tests green is essential for working on any team.

1.2. Objectives

- Use forks, remotes, branches, and PRs to collaborate.
- Resolve merge conflicts.

1.3. Team Setup

Steps for the Repo Owner:

1. Fork the repository on GitHub, **make sure to select 'Copy the main branch only'** or you will not get all of the provided code.
2. In your fork: Settings -> Collaborators -> add teammates with write access.
3. (Optional but recommended) Protect `main` so merges require PRs, approvals, and passing status checks.

All teammates then clone the team fork and add the instructor repo as `upstream`:

```
git clone https://github.com/<RepoOwner>/<team-fork>.git
git fetch --all --prune
```

Create and push a shared `development` branch (one-time):

```
git checkout -b development origin/main
git push -u origin development
```

Install the project dependencies:

MacOS with Homebrew

```
brew install cmake
```

Ubuntu/Debian

```
sudo apt-get install cmake
```

Windows

- Install CMake installer from https://cmake.org/download/
- During installation, check "Add CMake to the system PATH"

Verify CMake is installed:

```
cmake --version
ctest --version
```

Note that c++17 is recommended. If your compiler is older, you may need to update it.

The development branch serves as the team's central integration point. It's where feature pull requests are merged and continuous integration runs its checks before any changes are promoted to main

1.4. Create your feature branch and copy code

To start, each student (A, B, C, D) will create a feature branch from `development`. After creating this branch, the team will first merge in the instructor's `feature/normalization` branch to introduce conflicts that each student will later resolve.

Roles and feature branch names:

- Student A — feature/inverse-derivative
- Student B — feature/newton-iteration
- Student C — feature/definite-integral
- Student D — feature/function-linearization

1. Create your feature branch from `development`:

```
git checkout development
git pull origin development
git checkout -b feature/<feature-name>
```

1.5. Create Pull Request for the `feature/normalization` Branch

One student on the team will now need to merge the instructor's `feature/normalization` branch into `development` to introduce merge conflicts that each student will later resolve. Complete this before merging your individual feature branch into `development`, or you will have difficulty resolving conflicts later.

Steps in GitHub:

1. Open your personal fork repository on GitHub.
2. Click **Pull requests → New pull request**.
3. Set the branches:
   - **Base**: `development`
   - **Compare**: `feature/normalization`
5. Click **Create pull request**.
6. Use this title:

   ```
   Merge feature/normalization into development
   ```

Now, we will walk through the provided code for each student.

1.6. Feature work

Each student (A, B, C, D) will create a feature branch from `development` and copy the provided snippet for their role into the specified files. Your job will be to copy the code, run the tests, and merge your changes into `development` via a PR.

Roles and where to paste the provided snippets:

- Student A — Implement the inverse derivative
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double computeInverseDerivative(double x) {
    double deriv = fprime(x);
    if (std::abs(deriv) >= EPS) {
        return 1.0 / deriv;
    }
    return x;
}
```

  - In evaluate(...) (place where indicated):

```
if (modes & DERIVATIVE) {
    result = computeInverseDerivative(result);
}
```

- Student B — Implement the Newton-Raphson iteration
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double applyNewtonStep(double x) {
    double deriv = fprime(x);
    if (std::abs(deriv) >= EPS) {
        return x - f(x) / deriv;
    }
    return x;  // Guard: if f'(x) ≈ 0, no refinement
}
```

    - In evaluate(...) (place where indicated):

```
if (modes & NEWTON_STEP) {
    result = applyNewtonStep(result);
}
```

- Student C — Implement the definite integral evaluation
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double computeDefiniteIntegral(double x) {
    return Fantiderivative(x) - Fantiderivative(0.0);
}
```

    - In evaluate(...) (place where indicated):

```
if (modes & INTEGRAL) {
    result = computeDefiniteIntegral(result);
}
```

- Student D — Implement the function value with linearization
  - src/math.cpp
    - At the beginning of the namespace (place where indicated):

```
double computeFunctionValue(double x) {
    // Check if near critical points where f'(x) = 0 (x = ±1)
    if (std::abs(x - 1.0) < DELTA || std::abs(x + 1.0) < DELTA) {
        // Use linearization for stability
        double a = (std::abs(x - 1.0) < DELTA) ? 1.0 : -1.0;
        return f(a) + fprime(a) * (x - a);
    }
```

```
    return f(x);
}
```

– In evaluate(...) (place where indicated):

```
if (modes & VALUE) {
    result = computeFunctionValue(result);
}
```

1.7. Copy, build, test, and commit your feature branch

1. Copy the code snippets into the target files. Use the placeholder markers above to find the right content and location.

2. Build and run the tests locally:

```
mkdir -p build
cd build
cmake ..
cmake --build .
ctest --output-on-failure
./app
cd ..
```

It's important to run tests locally as soon as possible since they help catch issues and bugs early on. Identifying these problems right away not only saves time during conflict resolution but also ensures that pull requests ready for review. If merge conflicts are not resolved before opening a PR, it won't be able to be merged.

– Git status is good to know to see what files have changed:

```
git status
```

1. Commit and push your branch:

```
git add src/math.cpp src/math.hpp
git commit -m "Implemented <feature description>"
git push -u origin feature/<your-feature>
```

1. Sync yours with the latest development branch

```
git checkout development
git pull origin development
git checkout feature/<your-feature>
git merge development
```

In general, it is a good idea to sync with development often like this as you work on your feature branch to minimize conflicts later.

– Now, at this point, you should run into merge conflicts due to the feature/normalization PR you accepted earlier.

1.8. Resolve merge conflicts

You've just merged `development` into your feature branch and now each student must bring their work up to date and resolve any merge conflicts. Below we'll walk through VS Code Source Control (Any Git tool like GitKraken, GitHub Desktop, JetBrains, etc. can be used similarly). The goal is to produce a single `evaluate()` where the students' code is ordered: Student `A` at the top, then `B`, then `C`, then `D` last.

1.8.1. Open the conflicts in VS Code:

1. Open the **Source Control** panel (left sidebar icon with the branch).
2. In **Changes** you'll see files labeled **"Merge Changes"** or **"Conflicts"**. Click a conflicted file.
3. When prompted, choose **Open in Merge Editor** (you can also click the file's **Resolve in Merge Editor** button).

1.8.2. Understand the Merge Editor layout:

- **Left ("Current")**: your branch version (what you had locally before merging).

- **Right (“Incoming”)**: the `development` branch version you merged in.
- **Center (“Result”)**: the file you are building by choosing/ordering changes.
- Each conflict block has actions at the top: **Accept Current**, **Accept Incoming**, **Accept Both**, **Auto-merge**, and **Compare**.

1.8.3. Resolve conflicts inside `evaluate()`

You’ll need to ensure that all four students’ logic appears in the final `evaluate()` function in the correct order. The following is the expected pattern for the final `evaluate()` function after resolving conflicts. The order should be:

```
int evaluate(...) {
  // ----- feature/normalization -----
  // Normalization logic...

  // ----- Student A (top) -----
  // A’s logic...

  // ----- Student B (next) -----
  // B’s logic...

  // ----- Student C (next) -----
  // C’s logic...

  // ----- Student D (last) -----
  // D’s logic...

  // return statement
}
```

The steps to resolve:

1. In each conflict block that contains parts from **A, B, C, D**, click **“Accept Both”** (or manually choose pieces) so that eventually **all four students’ blocks** appear in the **center “Result”** in the specified order above.

   In the case of this excercise, we’ll prefer the union of both sides (Accept Both) because each student added their own helper function calls. However, in other scenarios, you may need to choose a resolution based on context.

2. Mark each conflict as resolved

   – In the Merge Editor, when a conflict block is handled, it will show **“Resolved”** for that block.
   – Repeat until **all conflict blocks** in the file are resolved.
   – Click **Complete Merge** (top right) or simply **Save**. The file leaves the conflicted state.

3. Stage, commit, and push

   1. Return to the source control panel.
   2. Review diffs for the files you touched.
   3. Click + to stage the resolved files.
   4. Enter a message like:

   ```
   Resolve merged conflicts in evaluate()
   ```

   1. Click **Commit & Push**.

4. Build and run tests as shown earlier to ensure the merge is correct.

1.9. Reviewing and merging PRs into `development`

Each student should open a PR with `base = development` and `compare = feat/your-role` using the ideas described in Section 1.5. Use brief descriptions and ask at least one teammate to review and approve. After approval, merge into `development`.

Code reviews are important because they ensure that another person looks over your work, providing an opportunity to catch errors or inconsistencies before any changes are merged into the main branch later.

1.10. Final integration into `main` and submission

Finally, once all four feature PRs are merged into `development`, the team should merge `development` into `main`. This should be done via a PR as well, with all team members reviewing and approving. In other scenarios, the development branch and main branch may have merge conflicts, and the steps we covered above for resolving conflicts will apply here as well.

Once each feature PR is merged into `development`, and `development` is merged into `main`, run the tests one last time to ensure everything is passing.

If all tests pass, your team has successfully completed the exercise!

Good job and remember to push often.