Aula 8 – Introdução ao Controle de Versão com git

EMC410235 - Programação Científica para Engenharia e Ciência Térmicas

Prof. Rafael F. L. de Cerqueira

2025.2

Por que usar controle de versão?

- Histórico completo das alterações no código
- Possibilidade de recuperar versões anteriores
- Registro de quem fez o quê e quando
- Comparação de versões com facilidade (`diff`)
- Suporte à colaboração e trabalho em equipe
- Redução de erros por sobrescrita ou perda de arquivos

Exemplo clássico (a evitar!)

```
versao_final_FINAL_v2_ok_corrigida_pra_valer.docx
```

Problemas comuns sem versionamento

- Perda de código por sobrescrita ou exclusão acidental
- Dificuldade em saber o que mudou e por quê
- Colaboração confusa: arquivos enviados por e-mail, pendrives etc.
- Falta de rastreabilidade de bugs e regressões
- Retrabalho por falta de sincronização entre membros da equipe
- Acúmulo de arquivos com nomes indecifráveis

O que é o Git?

- Sistema de controle de versão distribuído
- Criado por Linus Torvalds em 2005
- Projetado para ser rápido, confiável e seguro
- Armazena todo o histórico de alterações localmente
- Permite ramificações (branches) e fusões (merges) com facilidade
- Muito usado em projetos de software, ciência e engenharia

Git x GitHub: qual a diferença?

- **Git** é o sistema de controle de versão — funciona localmente
- **GitHub** é uma plataforma online para hospedar repositórios Git
- GitHub permite colaboração, revisão de código e integração com CI/CD
- Você pode usar Git sem GitHub, mas não o contrário
- Existem outras plataformas semelhantes: GitLab, Bitbucket, etc.

Resumo: Git faz o versionamento; GitHub facilita o trabalho em equipe na nuvem e compartilhamento.

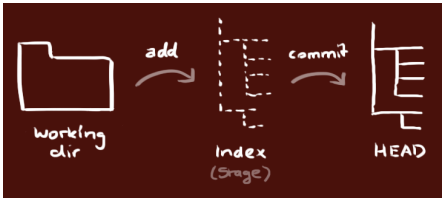
Instalando o Git

- **Windows:**
  - Baixar o instalador: `https://git-scm.com/downloads`
  - Usar as opções padrão na instalação
  - Após instalar, usar o `Git Bash` para os comandos
- **Linux:**
  - Ubuntu/Debian: `sudo apt install git`
  - Fedora: `sudo dnf install git`
  - Arch: `sudo pacman -S git`
- Verifique se funcionou: `git -version`

O ciclo básico do Git (init → add → commit)

- `git init`: inicia um repositório Git local
- `git add`: seleciona arquivos para entrar no próximo commit
- `git commit`: registra uma versão com mensagem descritiva
- Ciclo pode ser repetido a cada alteração no projeto
- Ideal para projetos com múltiplas versões e experimentações

"Adicionar" não é o mesmo que "salvar" — o commit é o ponto chave.



https://rogerdudler.github.io/git-guide/

git init, git status e git add

- `git init`
  - Inicializa um novo repositório Git no diretório atual
  - Cria a pasta oculta `.git`
- `git status`
  - Mostra o estado atual dos arquivos: modificados, não rastreados, prontos para commit
- `git add <arquivo>`
  - Adiciona arquivos modificados à staging area
  - Pode usar `.` para adicionar todos os arquivos

git commit: salvando versões

- `git commit -m "mensagem"`:
  - Salva uma nova versão dos arquivos da staging area
  - É obrigatório fornecer uma mensagem descritiva
- Cada commit recebe um identificador único (hash)
- O histórico de commits pode ser consultado com `git log`
- Commits ajudam a rastrear mudanças, colaborar e desfazer erros

Dica 1: escreva mensagens ***claras e objetivas*** *— elas serão lidas no futuro!*

Dica 2: escreva mensagens claras, ***em inglês e no imperativo***
— ex: `Add script for data preprocessing`

.gitignore: evitando arquivos desnecessários

- Arquivo especial que lista arquivos e pastas que o Git deve ignorar
- Evita versionar arquivos temporários, de sistema ou grandes (e.g. `.pyc`, `*.log`, `__pycache__/`)
- Útil para manter o repositório limpo e leve
- Pode ser criado manualmente: `touch .gitignore`
- Exemplo de conteúdo:
  - `*.log`
  - `*.pyc`
  - `__pycache__/`
  - `dados/tmp/`

Dica: use o site `gitignore.io` para gerar arquivos prontos para seu projeto.

git log e git diff: olhando o histórico

- `git log`:
  - Exibe o histórico de commits do repositório
  - Mostra hash, autor, data e mensagem do commit
  - Use q para sair da visualização
- `git diff`:
  - Compara diferenças entre arquivos modificados e a última versão ***commitada***
  - Útil para revisar o que será commitado
  - Pode comparar entre commits: `git diff <commit1> <commit2>`

Dica: combine com `git log -oneline` para uma visão compacta do histórico.

A estrutura oculta do Git (.git) - *For Linux Users Eyes Only...*

- Ao rodar `git init`, o Git cria um diretório oculto chamado `.git`
- Essa pasta armazena todo o histórico, configurações e metadados do repositório
- **Nunca edite essa pasta manualmente**
- Principais subpastas e arquivos:
  - `HEAD` – aponta para o último commit da branch atual
  - `objects/` – armazena os commits e arquivos versionados
  - `refs/` – guarda ponteiros para branches e tags
  - `config` – configurações específicas do repositório
- Você pode explorar com `ls -a` no terminal

Por que usar o GitHub?

- Permite hospedar repositórios Git na nuvem
- Facilita o trabalho em equipe com controle de acesso e colaboração
- Interface web para visualizar código, histórico e arquivos
- Integração com ferramentas de CI/CD, testes e deploy automático
- Possui recursos extras: issues, pull requests, wiki, projetos, etc.
- Gratuito para projetos públicos e privados (com limitações)

Git é local, GitHub é remoto — juntos, tornam o versionamento poderoso e colaborativo.

Criando um repositório no GitHub

- Acesse: `https://github.com` e faça login ou crie uma conta
- Clique em **"New repository"** no canto superior esquerdo
- Preencha:
  - Nome do repositório
  - Descrição (opcional, mas recomendada)
  - Visibilidade: público ou privado
  - **Não marque** a opção de criar README (se for conectar com um repositório local já existente)
- Clique em **Create repository**
- GitHub exibirá instruções para conectar seu repositório local

Dica: deixe o repositório vazio para facilitar o primeiro push.

Conectando Git local com GitHub (git remote)

- Após criar o repositório no GitHub, copie a URL (HTTPS ou SSH)
- No terminal, adicione o repositório remoto ao seu projeto local:
  - `git remote add origin <URL>`
- Verifique se foi adicionado corretamente:
  - `git remote -v`
- Primeiro envio de código:
  - `git push -u origin main`
- Depois disso, basta usar `git push` nos próximos commits

"origin" é apenas um apelido para a URL do repositório remoto.

git push e git pull: enviando e recebendo alterações

- `git push`:
  - Envia commits do repositório local para o repositório remoto (GitHub)
  - Primeira vez: `git push -u origin main`
  - Depois: `git push`

- `git pull`:
  - Baixa e incorpora alterações do repositório remoto
  - Equivale a `git fetch` + `git merge` - **mais indicado (e seguro)!**
  - Evita conflitos quando outros colaboradores já alteraram o projeto

Dica: dê pull antes de começar a trabalhar para evitar conflitos.

HTTPS vs SSH: como autenticar?

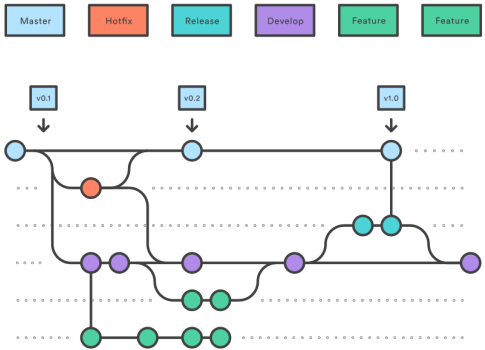
- **HTTPS**:
  - Autenticação via login/senha (obsoleto) ou **token de acesso pessoal**
  - Mais fácil de configurar inicialmente
  - Exige autenticação em cada push (a menos que gerencie credenciais)
  - **No PyCharm**, o token é integrado de forma prática e segura
- **SSH**:
  - Usa chaves criptográficas (`id_rsa`, `id_ed25519`)
  - Mais seguro e prático no longo prazo
  - Não pede senha após a configuração
- Verifique qual está usando: `git remote -v`

Recomendação: use Token com PyCharm ou configure SSH para linha de comando.

O que são branches?

- Uma **branch** é uma linha de desenvolvimento independente
- Permite modificar o projeto sem afetar o código principal
- Ideal para testar ideias, corrigir bugs ou desenvolver novas funcionalidades
- A branch principal costuma se chamar `main`
- Podemos unir branches com `git merge`

Exemplo clássico: criar uma branch chamada `feature-x`, desenvolver nela e depois juntar.



Criando e trocando de branch

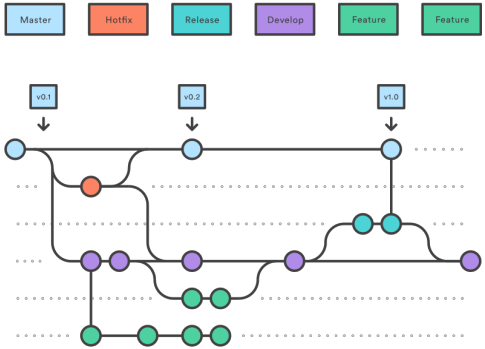
- `git branch <nome>`:
  - Cria uma nova branch com o nome especificado
- `git checkout <nome>`:
  - Troca para a branch desejada
- Ou use o atalho:
  - `git switch -c <nome>` (cria e troca)
  - `git switch <nome>` (só troca)
- Veja todas as branches:
  - `git branch`

Você pode ter várias branches no projeto, mas apenas uma está ativa por vez.

git merge: juntando alterações

- `git merge <branch>` junta o conteúdo da branch especificada na branch atual
- Exemplo: estando na `main`, use `git merge feature-x`
- Git tenta unir os commits automaticamente
- Após o merge, o histórico do projeto inclui as duas linhas de desenvolvimento
- Pode haver conflitos se as duas branches alteraram a mesma parte do código

Dica: sempre dê `git pull` antes de um merge para evitar conflitos desnecessários.



Conflitos de merge: exemplo simples

- Ocorre quando duas branches modificam a mesma linha em um mesmo arquivo
- O Git não sabe qual versão manter — exige intervenção manual
- Arquivo afetado fica com marcações como:
  - `<<<<< HEAD`
  - `=======`
  - `>>>>> feature-x`
- Você deve editar o arquivo, remover os marcadores e decidir o conteúdo final
- Após resolver:
  - `git add <arquivo>`
  - `git commit` (com ou sem mensagem, dependendo do Git)

Dica: editores como VS Code e PyCharm ajudam a visualizar e resolver conflitos.

Exemplo prático: criando e versionando um projeto

1. Criar uma pasta local com um arquivo de exemplo
2. Inicializar o repositório Git:
   - `git init`
3. Adicionar os arquivos:
   - `git add .`
4. Fazer o primeiro commit:
   - `git commit -m "Initial commit"`
5. Criar repositório vazio no GitHub
6. Conectar repositório remoto:
   - `git remote add origin <url>`
7. Enviar o projeto:
   - `git push -u origin main`

Exemplo completo para praticar: da criação até o envio para o GitHub.

Passo a passo: repositório local + GitHub

1. Crie um projeto local (ex: `meu-projeto/`)
2. Rode `git init` na pasta do projeto
3. Crie um arquivo (ex: `main.py`) e salve algo dentro
4. Adicione e commit:
   - `git add .`
   - `git commit -m "Initial commit"`
5. Vá ao GitHub, crie um novo repositório *sem README*
6. Conecte o repositório remoto:
   - `git remote add origin https://github.com/usuario/meu-projeto.git`
7. Envie com:
   - `git push -u origin main`

Pronto! Seu projeto está versionado localmente e publicado no GitHub.

Dicas de boas práticas em commits

- Faça commits pequenos e frequentes
- Cada commit deve representar uma ideia ou mudança completa
- Escreva mensagens claras, objetivas e no imperativo:
  - **Use:** `Add plot for velocity field`
  - **Evite:** `mudei umas coisas`
- Prefira escrever as mensagens em inglês, mesmo em projetos pessoais
- Use `git status` e `git diff` antes de commit
- Evite adicionar arquivos desnecessários ao repositório

Bons commits facilitam o trabalho em equipe e a manutenção do projeto.

Pull Requests (PRs) no mesmo repositório

- Mesmo com acesso de escrita ao repositório, é uma boa prática trabalhar em **branches separadas**
- Após concluir uma funcionalidade ou correção, abre-se um **Pull Request (PR)** para a `main`
- Vantagens:
  - Permite revisão de código antes do merge
  - Registra discussão, comentários e histórico da mudança
  - Pode acionar testes automáticos (CI)
- Evita merges diretos sem revisão e melhora a qualidade do projeto
- Ideal mesmo em repositórios pequenos ou individuais (com revisão própria)

PRs promovem revisão e controle de qualidade mesmo dentro de equipes pequenas.

Forks e PRs entre repositórios

- **Fork**: cópia completa de um repositório em sua conta
  - Usado quando você não tem permissão de escrita no repositório original
- Fluxo típico:
  1. Fazer fork do repositório
  2. Criar uma branch na sua cópia
  3. Fazer alterações e `push`
  4. Abrir um Pull Request para o repositório original
- Muito comum em projetos open source
- Ainda assim, segue o mesmo princípio de revisão via PR

O fork permite contribuir sem precisar de permissões diretas.

Recapitulando: fluxo completo do Git

1. `git init` – inicializa o repositório
2. `git add` – prepara arquivos para commit
3. `git commit` – salva uma versão local com mensagem
4. `git remote add origin <url>` – conecta ao GitHub
5. `git push` – envia alterações para o repositório remoto
6. `git pull` – traz alterações do repositório remoto
7. `git branch`, `git switch`, `git merge` – controle de branches
8. `.gitignore` – evita versionar arquivos indesejados
9. `git status`, `git log`, `git diff` – comandos para inspecionar

Esses são os blocos fundamentais para versionar e colaborar com segurança.

Materiais recomendados

- **Videoaula: Git e GitHub para iniciantes – Willian Justen**
  `https://www.youtube.com/playlist?list=PL1AbYrWSYTiPA2iEiQ2PF_A9j__C4hi0A`
- **Livro gratuito: Pro Git (Scott Chacon e Ben Straub)**
  `https://git-scm.com/book/pt-br/v2`
- **Site interativo: Learn Git Branching**
  `https://learngitbranching.js.org/`
- **Gerador de .gitignore personalizado**
  `https://www.toptal.com/developers/gitignore`
- **Documentação oficial do GitHub**
  `https://docs.github.com/`