Open Source Development Best Practices in Ratel

Jeremy L Thompson

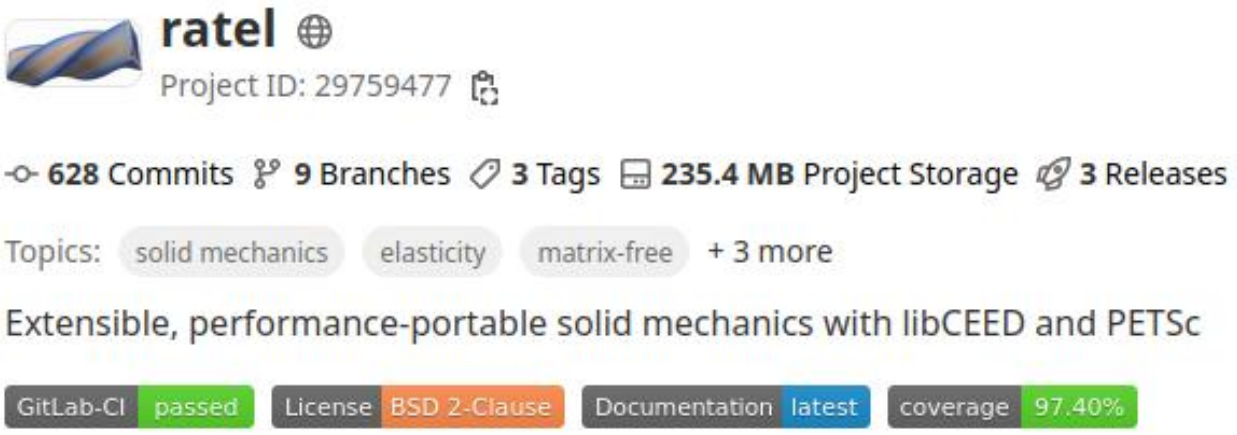
University of Colorado Boulder

jeremy@jeremylt.org

27 September 2022




Overview



Ratel is very young - 'move fast and break things'

Best practices from libCEED, PETSc break fewer things

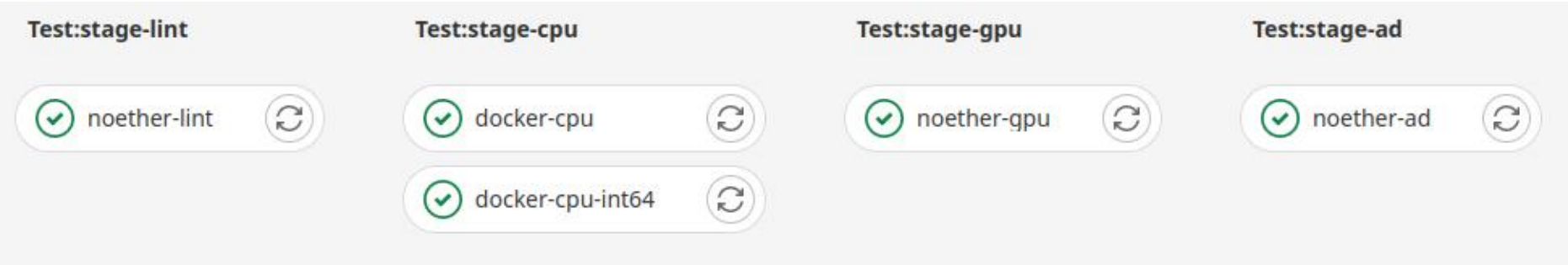
Overview

1. CI Testing
2. CD Containers and Documentation
3. Issue Tracker
4. Community Contributions

CI for commits and MRs

CI is as essential part of MR acceptance

- Automated testing for every commit and merge request
- Testing multiple hardware and build configurations
- Automatic static code analysis and formatting

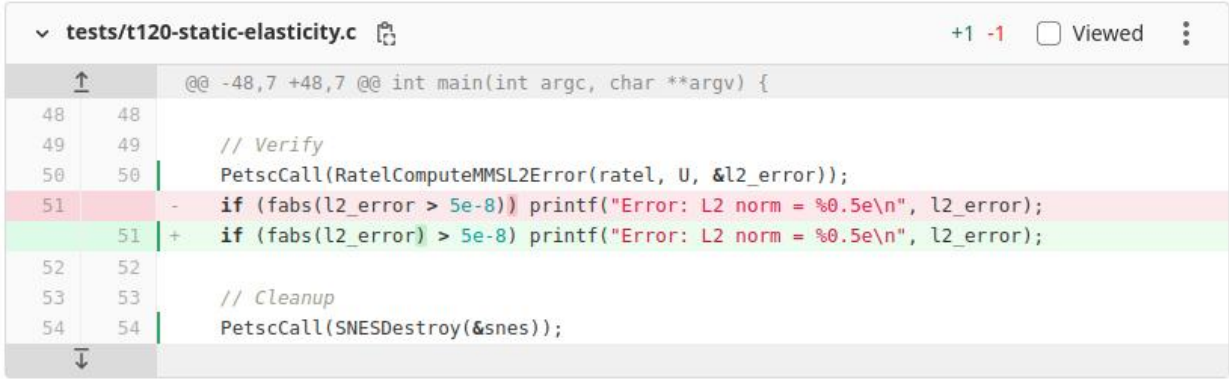


Unit testing

- Leverage upstream testing to lower test burden
- Linear model MMS to test solver setup
- Non-linear model regression tests
- Untested code is broken code; tested code is less broken code

Code coverage

- GitLab has native support for code coverage reports
- Highlights vulnerable, untested code in MR diffs
- Untested code is broken code; tested code is less broken code



Static analysis

`clang-tidy` - Clang-based C++ "linter" tool

- Extensible framework for diagnosing typical errors - style violations, interface misuse, etc

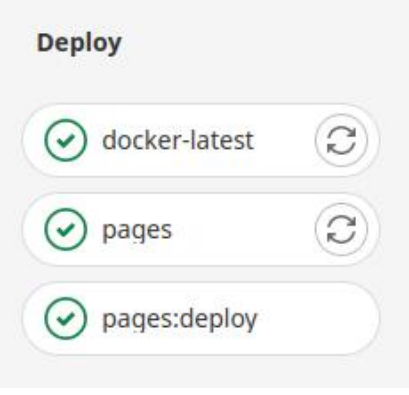
ClangFormat - formatting tools built on top of LibFormat

- Standalone tool with editor integrations
- Prevents format wars bikeshedding

Automated deployment

Automatic deployment upon commit to main

- Docker images for dev environment and latest snapshot
- GitLab pages documentation and theory guide



Docker containers

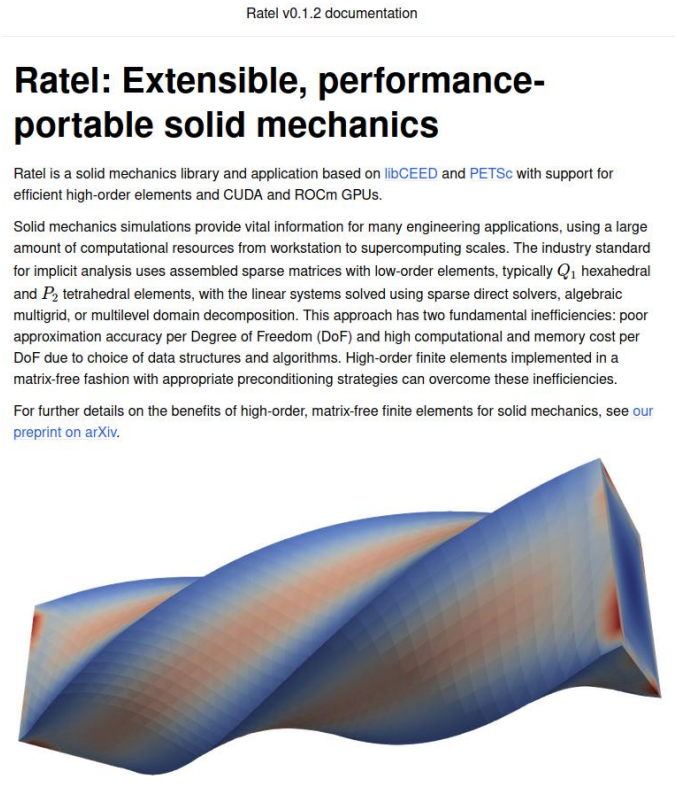
Snapshot and dev environment images

- User quick start with general CPU only image
- Exact dependency commit hashes shipped with Dockerfiles

```
host$ docker run -it --rm -v $(pwd):/work registry.gitlab.com/
    micromorph/ratel
container$ ratel-quasistatic -options_file config.yaml
```

Documentation

Latest documentation and theory guide



Ratel v0.1.2 documentation

Ratel: Extensible, performance-portable solid mechanics

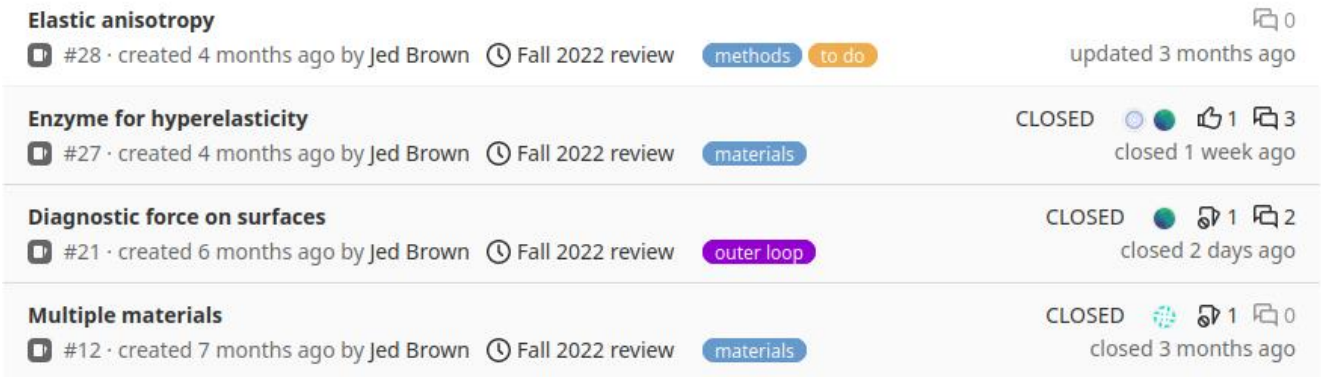
Ratel is a solid mechanics library and application based on libCEED and PETSc with support for efficient high-order elements and CUDA and ROCm GPUs.

Solid mechanics simulations provide vital information for many engineering applications, using a large amount of computational resources from workstation to supercomputing scales. The industry standard for implicit analysis uses assembled sparse matrices with low-order elements, typically Q_1 hexahedral and P_2 tetrahedral elements, with the linear systems solved using sparse direct solvers, algebraic multigrid, or multilevel domain decomposition. This approach has two fundamental inefficiencies: poor approximation accuracy per Degree of Freedom (DoF) and high computational and memory cost per DoF due to choice of data structures and algorithms. High-order finite elements implemented in a matrix-free fashion with appropriate preconditioning strategies can overcome these inefficiencies.

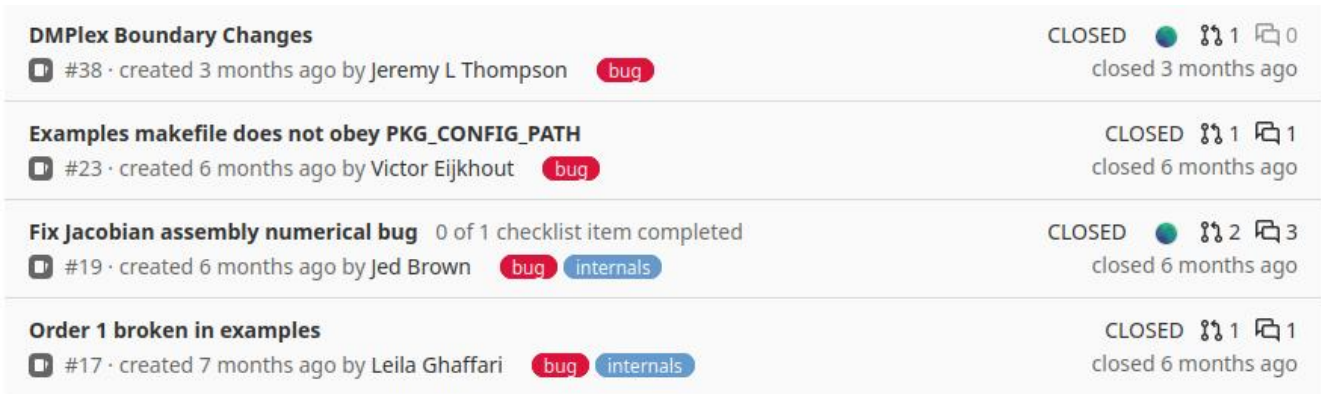
For further details on the benefits of high-order, matrix-free finite elements for solid mechanics, see our preprint on arXiv.

Issue tracker

- Transparent development roadmap

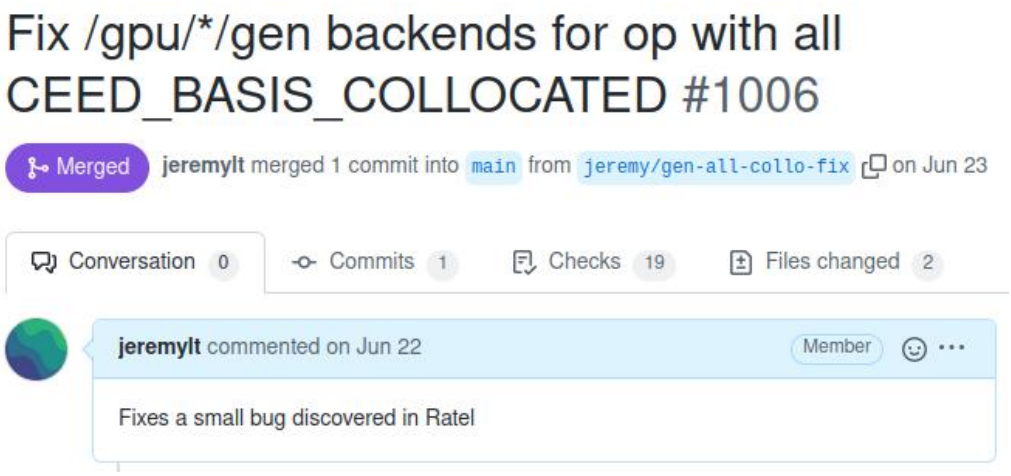


- Bug tracker



Upstream contributions

Upstream improvements to libCEED and PETSc



Questions?

1. CI Testing
2. CD Containers and Documentation
3. Issue Tracker
4. Community Contributions