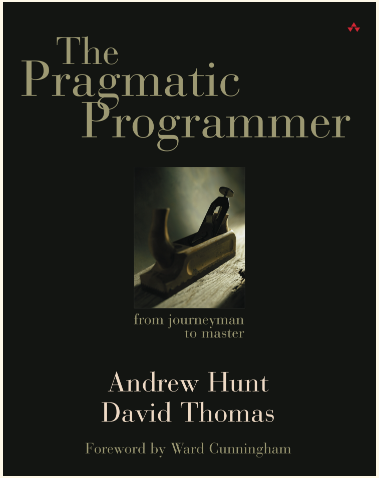
The Pragmatic Probabilistic Programmer

Mitzi Morris

2024-09-12

Stan Development Team

The Pragmatic Programmer, 1999



- Fight software rot;
- Capture real requirements;
- Avoid the trap of duplicating knowledge;
- Write flexible, dynamic, and adaptable code;
- Avoid programming by coincidence;
- Bullet-proof your code with contracts, assertions, and exceptions;
- Test ruthlessly and effectively;
- Make your developments more precise with automation.

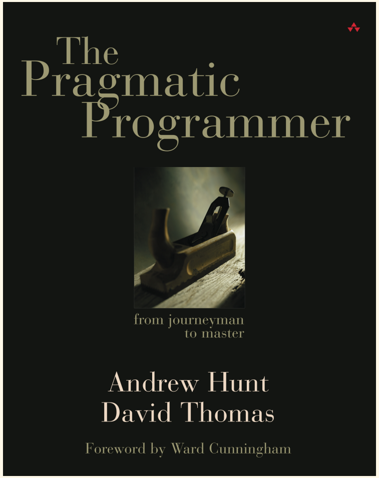
Stan Developer Process

- The three main goals of The Stan Developer Process are to support correctness, maintainability and ease of development.
    - All code must be unit tested for its main and edge case behavior against its API-level documentation.
    - All code should be designed as simply and modularly as possible using idiomatic C++ style for understandability.
    - Ease of co-operative development: the develop branch of every repository should always be in a releasable state.
    - Ease of development and maintainability: code should be written for future developers to read, with the aim of a future developer being able to easily figure out how to call a function and what its return value is.

Why This Talk at StanCon?

- *Talk about what you know:* I have (lots) more experience developing software than I do developing statistical models.
- Questions to keep in mind: what is a pragmatic probabilistic programmer?
- Is TPP and the Stan developer process relevant for your work?
  - Is this relevant to the process of developing a good analysis of a dataset? How?
  - What about generalizing that analysis for other, similar problems?
  - What about about writing a package so that others can run your analysis?

Part One: the Code



- Fight software rot;
- Capture real requirements;
- Avoid the trap of duplicating knowledge;
- Write flexible, dynamic, and adaptable code;
- Avoid programming by coincidence;
- Bullet-proof your code with contracts, assertions, and exceptions;
- Test ruthlessly and effectively;
- Make your developments more precise with automation.

Software Rot

- Address problems as soon as they are discovered
  - ***File a GitHub issue***
  - Do it on behalf of others, as needed.
- Fix or document
  - If bug is critical, there will be a patch release.
  - Otherwise, fix
- Demonstrate that you're on top of the situation
  - ***We filed a GitHub issue***
- Why?
  - *Project mindset: bad code leads to more bad code.*

Capture Real Requirements

- *"There's a simple technique for getting inside your users' requirements that isn't used often enough:* ***become a user****"*
  - Requirements specify **need**, not architecture, design, or the user interface.
  - Favor abstract specifications
- *"It's not whether you think inside the box or outside the box. The problem lies in finding the box:* ***identifying the real constraints****"*
  - Are you trying to solve the right problem, or is this a peripheral technicality?
  - Why is this thing a problem? What is it that's making it so hard to solve?
  - Does it have to be done this way?
- Fight feature creep
  - **Does it have to be done at all?**

GitHub Issues

What (if anything) can we glean about the state of the software from GitHub?

Repo	Open Issues	Closed Issues	Open PRs from 2024	Closed PRs	Date Created
CmdStan	54	609	2	604	02/16/14
Docs	83	236	2	484	11/25/18
Math	278	1025	8	1782	02/19/12
Stan	136	1573	4	1525	02/19/12

- Some issues have label “Bug” (software rot)
- Most issues have label “Feature” (feature creep)

Stan developer poll: how are we doing?

Real Requirements - Tracer Bullets (How to Shoot in the Dark)

*Tracer bullets operate in the same environment and under the same constraints as the real bullets. They provide **immediate feedback***

Tracer code is not disposable: you write it for keeps. It contains all the error checking, documentation, and self-checking that any piece of production code has. But it is not fully functional. Once you have an end-to-end connection among the components of your system, check how close to the target you are. Once you're on target, adding functionality is easy.

Tracer bullets show what you're hitting. This may not always be the target. You then adjust your aim until they're on target. That's the point.

***Not a Prototype** - With a prototype, you will throw it away and recode it properly using the lessons you've learned.*

Avoid Duplicating Knowledge, Write Adaptable/Flexible Code

- Don't Repeat Yourself

  Foster an environment where it's easy to find and reuse existing stuff. If it isn't easy, people won't do it. If you fail to reuse, you risk duplicating knowledge.

- Good API design is key. Desiderata:
  - Orthogonal - changing (internals) of one module doesn't affect others
  - "Shy" - document exposes functions, not internals

- Key References:
  - Refactoring: Improving the Design of Existing Code
  - Design Patterns: Elements of Reusable Object-Oriented Software
  - "Effective Programming" (in C++, in Python, etc)

Avoid programming by coincidence / Bullet-proof your code

- *Design by Contract*: specify preconditions, postconditions
  - If preconditions are met, then postconditions will be too.
  - Otherwise: raise exception, or terminate (gracefully).
- Stan developer process mandates:
  - Functions have initial documentation comment
  - Describe parameters, return
  - Note exceptions thrown
- DBC is supported directly in some programming languages
  - C++ is not one of them
  - Enforced by code review by qualified Stan developer

Example

```cpp
/**
 * Validate that diag inverse Euclidean metric is positive definite
 *
 * @param[in] inv_metric  inverse Euclidean metric
 * @param[in,out] logger Logger for messages
 * @throws std::domain_error if matrix is not positive definite
 */
inline void validate_diag_inv_metric(const Eigen::VectorXd& inv_metric,
                                     callbacks::logger& logger)
{
  try {
    stan::math::check_finite("check_finite", "inv_metric", inv_metric);
    stan::math::check_positive("check_positive", "inv_metric", inv_metric);
  } catch (const std::domain_error& e) {
    logger.error("Inverse Euclidean metric not positive definite.");
    throw std::domain_error("Initialization failure");
  }
}
```

Test Ruthlessly and Effectively: Unit Tests

- *"We suggest that every module have its own unit test built into its code, and that these tests be performed automatically as part of the regular build process."*
- Unit tests and bug fixes test orthogonality.
  - Orthogonality reduces the interdependency among components, therefore an orthogonally designed and implemented system is easier to test.
- How modular is the code?
  - What does it take to build and link a unit test?
  - How many files/functions/modules does the bug fix touch?

Unit Test for `validate_diag_inv_metric`

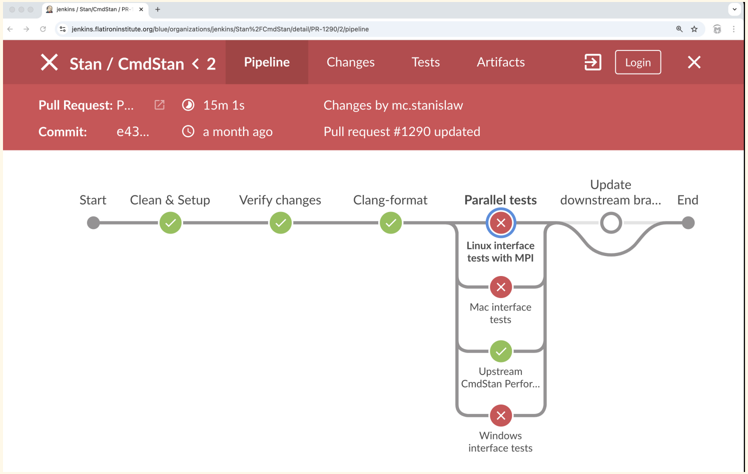
```cpp
TEST(inv_metric, validate_diag_imm) {
  stan::callbacks::logger logger;
  Eigen::VectorXd v1(1);
  v1(0) = 0.0;
  EXPECT_THROW(stan::services::util::validate_diag_inv_metric(v1, logger),
               std::domain_error);
  v1(0) = 1.0;
  EXPECT_NO_THROW(stan::services::util::validate_diag_inv_metric(v1, logger));

  Eigen::VectorXd v2(2);
  v2(0) = 1.0;
  v2(1) = 0.0;
  EXPECT_THROW(stan::services::util::validate_diag_inv_metric(v2, logger),
               std::domain_error);
  ...
```

Test Ruthlessly and Effectively: Integration Tests

- Single Model Integration Tests
    - Tests that a valid Stan model can be transpiled to C++ and compiled to an executable program:
- Command line interface (CLI) tests
    - Expected behavior for all combinations of user config
- End-to-end inference tests
    - Correct outputs given correct inputs
    - Correct behavior on incorrect inputs
- Performance tests
    - Run CmdStan against selected benchmarks

Test Ruthlessly, Effectively and Automatically

Automated testing using Jenkins CI



Part Two: the Code of Conduct

Another ***very*** *influential book:* The Psychology of Computer Programming

Gerald M Weinberg.

Introduces the concept of **Egoless Programming**

(in the context of *code review*, where programmers review each other's work to ensure quality)

The 10 Commandments of Egoless Programming

1. Understand and accept that you will make mistakes
   - Understand that mistakes are part of programming. Acknowledging them allows you to improve.
2. You are not your code
   - Your code may have flaws, but that doesn't reflect on your abilities. Criticism of your code should not be taken personally.
3. No matter how much "karate" you know, someone else will always know more
   - Programming is vast and there will always be someone who knows something you don't. Embrace continuous learning.

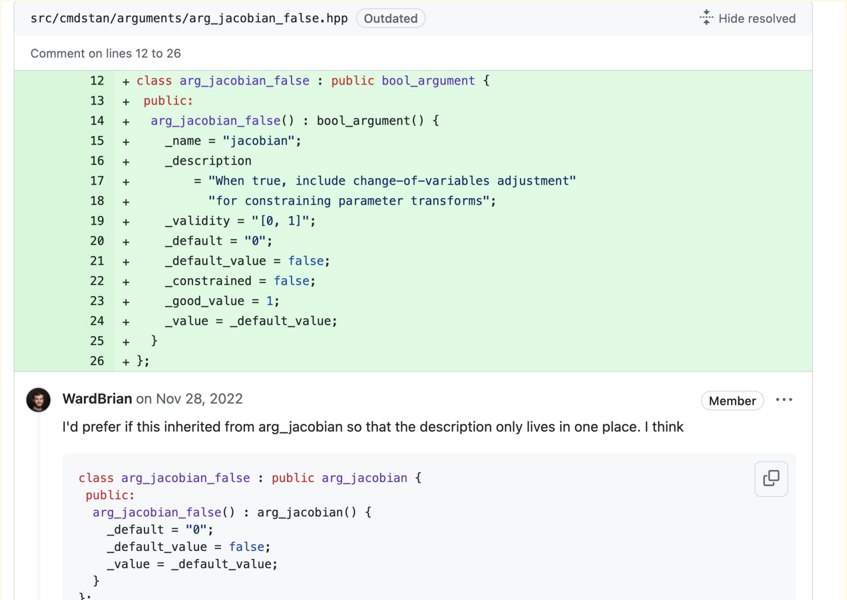
The 10 Commandments of Egoless Programming, continued

4. Don't rewrite code without consultation
    - Before rewriting someone else's code, understand it and consult with the author. There might be reasons for its current form.
5. Treat people who know less than you with respect, deference, and patience
    - Helping others learn is part of the job. Teaching with patience fosters a positive and collaborative environment.
6. The only constant in the world is change
    - Be adaptable. Technologies, requirements, and tools are always evolving. Don't resist change.
7. The only true authority stems from knowledge, not from position
    - Respect comes from your knowledge and expertise, not from your title or job rank.

The 10 Commandments of Egoless Programming

8. Fight for what you believe, but gracefully accept defeat
   - Advocate for your ideas, but if the team chooses a different path, accept it gracefully and move forward.
9. Don't be "the coder in the corner"
   - Engage with your team, share your knowledge, and ask for help when needed. Programming is a collaborative effort.
10. Critique code instead of people – be kind to the coder, not the code
    - When reviewing code, focus on improving the code, not on criticizing the person who wrote it. Provide constructive feedback.

Code review



src/cmdstan/arguments/arg_jacobian_false.hpp Outdated Hide resolved

Comment on lines 12 to 26

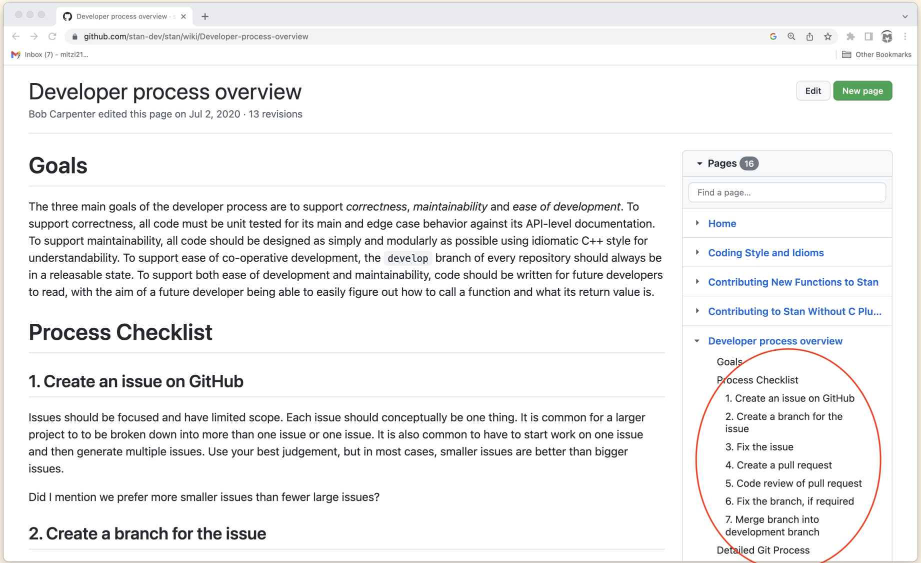
```
+ class arg_jacobian_false : public bool_argument {
+  public:
+   arg_jacobian_false() : bool_argument() {
+     _name = "jacobian";
+     _description
+         = "When true, include change-of-variables adjustment"
+           "for constraining parameter transforms";
+     _validity = "[0, 1]";
+     _default = "0";
+     _default_value = false;
+     _constrained = false;
+     _good_value = 1;
+     _value = _default_value;
+   }
+ };
```

WardBrian on Nov 28, 2022 Member

I'd prefer if this inherited from arg_jacobian so that the description only lives in one place. I think

```
class arg_jacobian_false : public arg_jacobian {
 public:
  arg_jacobian_false() : arg_jacobian() {
    _default = "0";
    _default_value = false;
    _value = _default_value;
  }
};
```

Stan Developer Process



Developer process overview · s

github.com/stan-dev/stan/wiki/Developer-process-overview

Inbox (7) - mitzi21...

Other Bookmarks

Developer process overview

Bob Carpenter edited this page on Jul 2, 2020 · 13 revisions

Edit | New page

Goals

The three main goals of the developer process are to support *correctness*, *maintainability* and *ease of development*. To support correctness, all code must be unit tested for its main and edge case behavior against its API-level documentation. To support maintainability, all code should be designed as simply and modularly as possible using idiomatic C++ style for understandability. To support ease of co-operative development, the `develop` branch of every repository should always be in a releasable state. To support both ease of development and maintainability, code should be written for future developers to read, with the aim of a future developer being able to easily figure out how to call a function and what its return value is.

Process Checklist

1. Create an issue on GitHub

Issues should be focused and have limited scope. Each issue should conceptually be one thing. It is common for a larger project to to be broken down into more than one issue or one issue. It is also common to have to start work on one issue and then generate multiple issues. Use your best judgement, but in most cases, smaller issues are better than bigger issues.

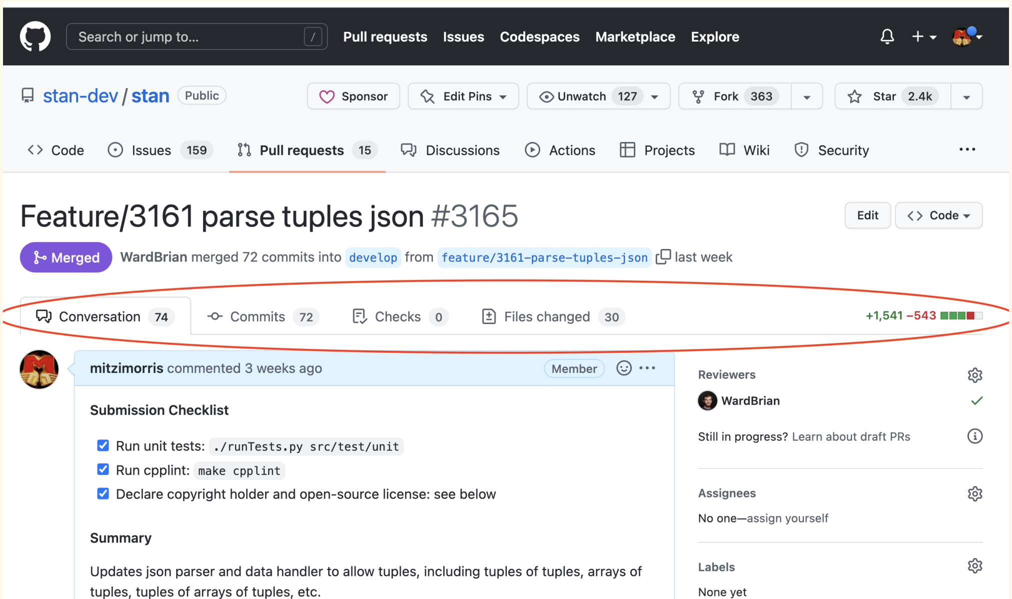
Did I mention we prefer more smaller issues than fewer large issues?

2. Create a branch for the issue

Stan Developer Process for a New Feature

- Any new feature involves
    - multiple iterations
    - successive refinements
    - extensive testing
    - developer review and consensus

Example PR: JSON parser



Feature/3161 parse tuples json #3165

Merged **WardBrian** merged 72 commits into `develop` from `feature/3161-parse-tuples-json` last week

Conversation 74 | Commits 72 | Checks 0 | Files changed 30 | +1,541 −543

mitzimorris commented 3 weeks ago · Member

Submission Checklist

- [x] Run unit tests: `./runTests.py src/test/unit`
- [x] Run cpplint: `make cpplint`
- [x] Declare copyright holder and open-source license: see below

Summary

Updates json parser and data handler to allow tuples, including tuples of tuples, arrays of tuples, tuples of arrays of tuples, etc.

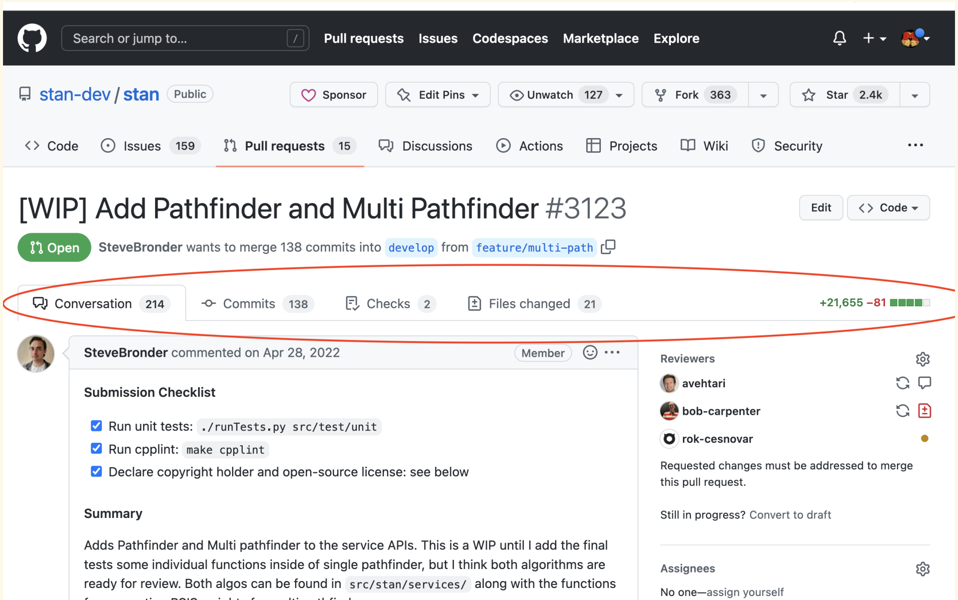
Reviewers
WardBrian ✓

Still in progress? Learn about draft PRs

Assignees
No one—assign yourself

Labels
None yet

Example PR: Pathfinder



[WIP] Add Pathfinder and Multi Pathfinder #3123

Open SteveBronder wants to merge 138 commits into `develop` from `feature/multi-path`

Conversation 214 | Commits 138 | Checks 2 | Files changed 21 | +21,655 −81

SteveBronder commented on Apr 28, 2022 — Member

Submission Checklist

- [x] Run unit tests: `./runTests.py src/test/unit`
- [x] Run cpplint: `make cpplint`
- [x] Declare copyright holder and open-source license: see below

Summary

Adds Pathfinder and Multi pathfinder to the service APIs. This is a WIP until I add the final tests some individual functions inside of single pathfinder, but I think both algorithms are ready for review. Both algos can be found in `src/stan/services/` along with the functions

Reviewers
avehtari
bob-carpenter
rok-cesnovar

Requested changes must be addressed to merge this pull request.

Still in progress? Convert to draft

Assignees
No one—assign yourself

Bayesian Workflow vs. Pragmatic Programming

- Fundamental Pragmatic priorities: code-centric
  - Robustness
  - Maintainability
- Bayesian Workflow priorities: model-centric
  - Demonstrate goodness of inference
  - Research priority: novelty
- Clear Similarities: importance of test datasets
  - Simulated data for testing
  - Real-world data for testing
- Not So Clear:
  - Upfront requirements: speed, scalability, accuracy
  - Code reuse through modularization
  - Code review

Discussion

What is a pragmatic probabilistic programmer?

- Is TPP and the Stan developer process relevant for your work?
  - Is this relevant to the process of developing a good analysis of a dataset? How?
  - What about generalizing that analysis for other, similar problems?
  - What about about writing a package so that others can run your analysis?

Many thanks to all my fellow Stan Developers!