BERKELEY LAB
Bringing Science Solutions to the World

U.S. DEPARTMENT OF ENERGY
Office of Science

Experiences with Idiomatic Vibe Coding

Damian Rouson
Computer Languages and Systems Software

International Fortran Conference, November 4-5, 2025

Acknowledgements

The Ancestors

Dr. W. Ervin and Mrs. Vivian Rouson

The Berkeley Lab Fortran Team

Dan Bonachea, Paul Hargrove, Hugh Kadhem, Katherine Rasmussen

Collaborators

Fortran Package Manager: Brad Richardson

Julienne: Katherine Rasmussen, Dan Bonachea, Desvaun Drummond

Fiats: Zhe Bai, Jeremiah Bailey, Baboucarr Dibba, Ethan Gutmann, David Torres, Federica Villani, Kareem Jabbar Weaver, Jordan Welsman, Yunhao Zhang

LLVM Flang: Kareem Ergawy, Jeff Hammond, Michael Klemm, Jean-Didier Paillex, Etienne Renault

Matcha: David Torres, Dominick Martinez, Joseph Hellmers

MOLE: Jose Castillo, Johnny Corbino, Joseph Hellmers

Sponsors

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, and Office of Nuclear Physics. This research was supported by the Lawrence Berkeley National Laboratory Laboratory Research and Development (LDRD) program. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

Overview

01

Introduction:
Correctness checking with Julienne

02

Methodology:

Vibe coding

03

Methodology:

Idiomatic vibe coding

04

Results:

Vibe coding with LLMs

04

Conclusions

Fortran Unit-Testing Software

Metric	Package Count	Package Names (if < 5)
Fortran $> 70\%$	21	
Fortran $\approx 100\%$	3	forunittest, Julienne, par-funnel
HEAD age < 12 months	9	
Archived (read-only)	1	funit
Organizationally hosted	3	Julienne, pFUnit, test-drive
Releases + tags > 0	11	
Contributors > 1	7	
Contributors ≥ 5	2	pFUnit, test-drive, veggies
Commits > 100	4	FortUTF, Julienne pFUnit, veggies
User-defined operators	1	Julienne
Multi-image support	3	Julienne, pFUnit, Garden

GitHub Data as of 15 May 2025

Fortran Assertion Utilities

Metric	Package Names
Fortran $> 70\%$	Assert, fassert
Fortran $= 100\%$	Assert, fassert
HEAD age < 12 months	Assert, fassert
Organizationally hosted	Assert
Releases + tags > 0	Assert, assert-fortran-git
Contributors > 1	Assert
Contributors ≥ 5	Assert
Commits > 100	Assert, fassert
Pure procedure support	Assert, fassert
Multi-image support	Assert

GitHub Data as of 15 May 2025

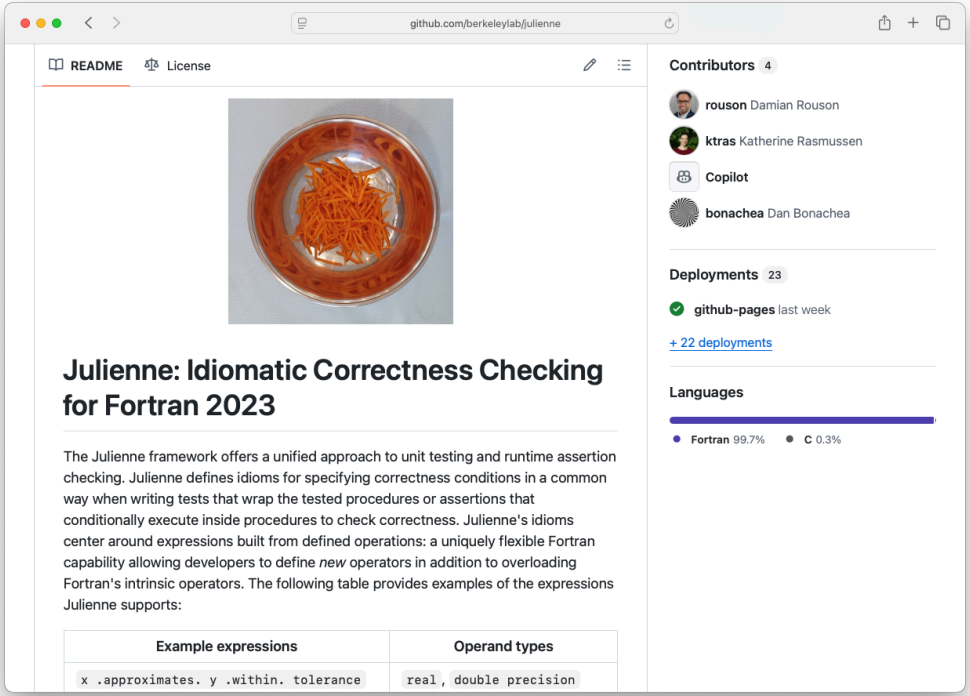
Correctness-Checking with Julienne

Unified Idioms for writing

- Unit tests
- Assertions

Support for Fortran 2023 native parallelism

- Multi-image testing: a collective reduction detects failure on a subset of images
- Assertions are `pure` procedures as required for invocation inside a `do concurrent` construct.



https://go.lbl.gov/julienne

Julienne Idioms

Row	Example expressions	Supported operand types of the bold operator
1	`x .approximates. y .within. tolerance`	`real, double precision`
2	`x .approximates. y .withinFraction. tolerance`	`real, double precision`
3	`x .approximates. y .withinPercentage. tolerance`	`real, double precision`
4	`.all. ([i,j] .lessThan. k)`	`test_diagnosis_t`
5	`.all. ([i,j] .lessThan. [k,m])`	`test_diagnosis_t`
6	`.all. (i .lessThan. [k,m])`	`test_diagnosis_t`
7	`(i .lessThan. j) .also. (k .equalsExpected. m)`	`test_diagnosis_t`
8	`x .lessThan. y`	`integer, real, double precision`
9	`x .greaterThan. y`	`integer, real, double precision`
10	`i .equalsExpected. j`	`integer, character, type(c_ptr)`
11	`i .isAtLeast. j`	`integer, real, double precision`
12	`i .isAtMost. j`	`integer, real, double precision`
13	`s .isBefore. t`	`character`
14	`s .isAfter. t`	`character`
15	`(.expect. allocated(A)) // '(expected an allocated array "A")'`	`logical`

String Utilities in Julienne

Example expression	Result
`s%bracket()`, where `s=string_t("abc")`	`string_t("[abc]")`
`s%bracket("_")`, where `s=string_t("abc")`	`string_t("_abc_")`
`s%bracket("{","}")`, where `s=string_t("abc")`	`string_t("{abc}")`
`string_t(2)`	`string_t("2")`
`string_t(["a", "b", "c"])`	`[string_t("a"),string_t("b"),string_t("c")]`
`.cat. string_t([9,8,7])`	`string_t("987")`
`.csv. string_t([1.5,2.0,3.25])`	`string_t("1.50000000,2.00000000,3.25000000")`
`["do", "re", "mi"] .sv. "|"`	`string_t("do|re|mi")`
`string_t("ab") // string_t("cd")`	`string_t("abcd")`
`"ab" // string_t("cd")`	`string_t("abcd")`
`string_t("ab") // "cd"`	`string_t("abcd")`

An Idiomatic Assertion

```
$ cat test/assertion_failure_demo.F90
#include "julienne-assert-macros.h"
program assertion_failure_demo
  !! Demonstrate a failing idiomatic assertion
  use julienne_m, only : call_julienne_assert_, operator(.equalsExpected.)
  implicit none
  print '(a)', 'Testing intentional failure of idiomatic assertion:'
  call_julienne_assert(2+2 .equalsExpected. 5)
end program
```

Assertion Output

```
$ fpm test --compiler flang-new assertion_failure_demo --flag -DASSERTIONS
<build output elided...>
Testing intentional failure of idiomatic assertion:
Fortran ERROR STOP: Assertion failure at ./test/assertion_failure_demo.F90:7:
call_julienne_assert(2+2 .equalsExpected. 5)
expected 5; actual value is 4
<ERROR> Execution for object " assertion_failure_demo " returned exit code  1
<ERROR> *cmd_run*:stopping due to failed executions
STOP 1
```

A Specimen Test

```
7
8   type, extends(test_t) :: specimen_test_t
9   contains
10    procedure, nopass :: subject
11    procedure, nopass :: results
12  end type
13
```

Defining A Test Subject

```
16   pure function subject() result(test_subject)
17     character(len=:), allocatable :: test_subject
18     test_subject = 'A specimen'
19   end function
```

Describing and Running Tests

```
21  function results() result(test_results)
22    type(specimen_test_t) specimen_test
23    type(test_result_t), allocatable :: test_results(:)
24    test_results = specimen_test%run( &
25      [test_description_t('doing something', do_something) &
26      ,test_description_t('checking something', check_something) &
27      ,test_description_t('skipping something') &
28    ])
29  end function
30
```

Constructing a Test Diagnosis

```
31  function check_something() result(test_diagnosis)
32    type(test_diagnosis_t) test_diagnosis
33    type(specimen_t) specimen
34    test_diagnosis = .all.( &
35       [22./7., 3.14159] .approximates. specimen%pi() .within. 0.001 &
36    ) // ' (pi approximation)'
37  end function
```

Unit Test Output

```
A specimen
   passes on doing something.
   FAILS  on checking something.
      diagnostics:
        expected 3.141592741013 within a tolerance of 0.1000000047497E-02;
        actual value is 3.142857074738 (pi approximation)
   SKIPS  on skipping something.
 1 of 3 tests passed. 1 tests were skipped.
```

Julienne Compiler Support

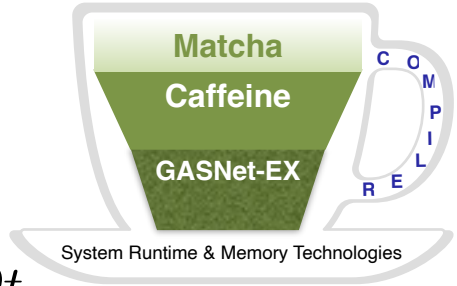
Vendor	Compiler	Version(s) Tested
GCC	`gfortran`	13.4.0, 14.3.0, 15.1.0
LLVM	`flang-new`	19, 20.1.8, 21.1.0
NAG	`nagfor`	7.2 Build 7235
Intel	`ifx`	2025.2.1 Build 20250806

Julienne Use Case: Computational Science

Matcha: Motility analysis of T-cell histories in activation

A parallel virtual T-cell model.

- Matcha tracks the stochastic T-cell motions according to multiple distributions of speeds and angles, accounting for the dependence of speed on the turning angle and on the previous speed.
- T cells must mount a coordinated attack in order to avoid overwhelming the host tissue.
- The study of T-cell/T-cell interactions remains in its infancy [1].
- Some communication occurs via secreting soluble mediators, e.g., cytokines and chemokines.
- Matcha models mediator spread via a 3D diffusion equation implemented with defined operations implemented in pure functions:



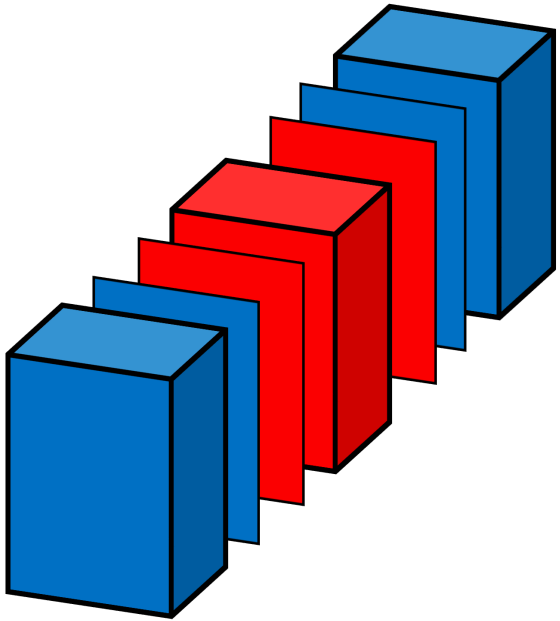
$$\phi_t = D\nabla^2\phi \quad \text{, where} \quad \phi_t = \partial\phi/\partial t$$

Collaborator:
Prof. David Torres
Northern New Mexico College
via Sustainable Research Pathways

[1] L.F. Uhl and A. Ge´rard A. “Modes of communication between T cells and relevance for immune responses.” *Int. J. Mol. Sci.* **2020**, *21*, 2674; doi:10.3390/ijms21082674

Problem Domain

- Unit cube
- Partitioned along one spatial direction
- Boundary subdomains (blue rectangular solids)
- Internal subdomains (red rectangular solids)
- Halos (blue and red planes)



```
type subdomain_t
  private
  real, allocatable :: s_(:,:,:)
contains
  generic :: operator(.laplacian.) => laplacian
  procedure, private :: laplacian
  ! ... lines ommitted
end type
! ... lines omitted
interface
  pure module function laplacian(rhs) result(laplacian_rhs)
    implicit none
    class(subdomain_t), intent(in) :: rhs
    type(subdomain_t) laplacian_rhs
  end function
  ! ... lines ommitted
end interface
```

Fig. 2. Excerpted `subdomian_t` derived type, `operator(.laplacian.)` generic binding, and `laplacian` function interface body from the Matcha repository `git tag ai4dev-workshop`.

Target Solution

- ~46-line separate module procedure
- Pure function
- Do concurrent with locality specifiers
- Array statements
- Associate

```
module procedure laplacian
  integer i, j, k
  real, allocatable :: halo_west(:,:), halo_east(:,:)

  allocate(laplacian_rhs%s_(my_nx, ny, nz))
  halo_west = merge(halo_x(west,:,:), rhs%s_(1,:,:), me/=1) ! conditionally use halo value
  i = my_internal_west
  ! Compute Laplacians throughout the low-x boundary subdomain using non-allocatable associations:
  associate( laplacian_phi =>  laplacian_rhs%s_, inbox => halo_west, phi=>rhs%s_)
    do concurrent(j=2:ny-1, k=2:nz-1) &
      default(none) shared(laplacian_phi, inbox, phi, dx_, dy_, dz_, i) !Fortran 2018 loacality specifiers
      laplacian_phi(i,j,k) = (inbox(j,k   ) - 2*phi(i,j,k) + phi(i+1,j  ,k  ))/dx_**2 + &
                             (phi(i,j-1,k  ) - 2*phi(i,j,k) + phi(i  ,j+1,k  ))/dy_**2 + &
                             (phi(i,j  ,k-1) - 2*phi(i,j,k) + phi(i  ,j  ,k+1))/dz_**2
    end do
  end associate
  ! Compute Laplacians throughout non-boundary subdomains with non-allocatable associations:
  associate(laplacian_phi => laplacian_rhs%s_, phi=>rhs%s_)
    do concurrent(i=my_internal_west+1:my_internal_east-1, j=2:ny-1, k=2:nz-1) &
      default(none) shared(laplacian_phi, phi, dx_, dy_, dz_) ! Fortran 2018 locality specifiers
      laplacian_phi(i,j,k) = (phi(i-1,j  ,k  ) - 2*phi(i,j,k) + phi(i+1,j  ,k  ))/dx_**2 + &
                             (phi(i  ,j-1,k  ) - 2*phi(i,j,k) + phi(i  ,j+1,k  ))/dy_**2 + &
                             (phi(i  ,j  ,k-1) - 2*phi(i,j,k) + phi(i  ,j  ,k+1))/dz_**2
    end do
  end associate

  halo_east = merge(halo_x(east,:,:), rhs%s_(my_nx,:,:), me/=num_subdomains) !conditionally use halo value
  i = my_internal_east
  ! Compute Laplacians throughout the high-x boundary subdomain using non-allocatable associations:
  associate(laplacian_phi =>  laplacian_rhs%s_, inbox => halo_east, phi=>rhs%s_)
    do concurrent(j=2:ny-1, k=2:nz-1) & ! compute Laplacian in low-x boundary subdomain
      default(none) shared(laplacian_phi, inbox, phi, dx_, dy_, dz_, i) ! Fortran 2018 locality specifiers
      laplacian_phi(i,j,k) = (phi(i-1,j  ,k  ) - 2*phi(i,j,k) + inbox(  j  ,k  ))/dx_**2 + &
                             (phi(i  ,j-1,k  ) - 2*phi(i,j,k) + phi(i  ,j+1,k  ))/dy_**2 + &
                             (phi(i  ,j  ,k-1) - 2*phi(i,j,k) + phi(i  ,j  ,k+1))/dz_**2
    end do
  end associate

  laplacian_rhs%s_(:, 1,:) = 0. ! low-y boundary
  laplacian_rhs%s_(:,ny,:) = 0. ! high-y boundary
  laplacian_rhs%s_(:,:, 1) = 0. ! low-z boundary
  laplacian_rhs%s_(:,:,nz) = 0. ! high-z boundary

  if (me==1) laplacian_rhs%s_(1,:,:) = 0. ! low-x boundary
  if (me==num_subdomains) laplacian_rhs%s_(my_nx,:,:) = 0. ! high-x boundary
end procedure
```

Vibe Coding Workflow

1. In a Linux or macOS shell, set your present working directory to Matcha's scripts[a] directory. Enter the command `./create-single-source-file-programs.sh`, which creates a copy of the Matcha test suite and supporting software stack all concatenated into one file in the following path relative to the project's root directory: "./build/single-file-programs/test-suite.F90".
2. Compile and execute test-suite.F90 to check that the output reports that all tests pass. For example, enter

```
gfortran -fcoarray=single -o test-suite test-suite.F90
./test-suite
```

3. Use the contents of vibe-coding/README.md[b] as your prompt to a large language model (LLM).
4. Use an editor to edit the test-suite.F90 lines beginning and ending with `module procedure laplacian` and `end procedure laplacian`, respectively, replacing those lines with the LLM's response.
5. Compile test-suite.F90 as in step 2 again.
6. If the test-suite program doesn't compile or if running the compiled program doesn't produce output indicating that all tests pass, then start over at step 1 but when you reach step 3, edit the prompt as follows:
   - Append the previous iteration's compile-time error message(s) or run-time error message(s) or test output.
   - Append the text "Please fix the above errors that were generated by compiling the following candidate solution to this prompt:". If the program compiled but runtime errors resulted, replace "compiling" with "running" in the previous sentence. If the program ran without errors, but tests failed replace "compiling" with "running" and replace "errors" with "test failures".
   - Append LLM-generated code from the previous iteration.

[a] ../../scripts
[b] ./vibe-coding/README.md

A Vibe Coding Workflow

1. In a Linux or macOS shell, set your present working directory to Matcha's scripts[a] directory. Enter the command `./create-single-source-file-programs.sh`, which creates a copy of the Matcha test suite and supporting software stack all concatenated into one file in the following path relative to the project's root directory: "./build/single-file-programs/test-suite.F90".
2. Compile and execute test-suite.F90 to check that the output reports that all tests pass. For example, enter

```
gfortran -fcoarray=single -o test-suite test-suite.F90
./test-suite
```

3. Use the contents of vibe-coding/README.md[b] as your prompt to a large language model (LLM).
4. Use an editor to edit the test-suite.F90 lines beginning and ending with `module procedure laplacian` and `end procedure laplacian`, respectively, replacing those lines with the LLM's response.
5. Compile test-suite.F90 as in step 2 again.
6. If the test-suite program doesn't compile or if running the compiled program doesn't produce output indicating that all tests pass, then start over at step 1 but when you reach step 3, edit the prompt as follows:
   - Append the previous iteration's compile-time error message(s) or run-time error message(s) or test output.
   - Append the text "Please fix the above errors that were generated by compiling the following candidate solution to this prompt:". If the program compiled but runtime errors resulted, replace "compiling" with "running" in the previous sentence. If the program ran without errors, but tests failed replace "compiling" with "running" and replace "errors" with "test failures".
   - Append LLM-generated code from the previous iteration.

[a] ../../scripts
[b] ./vibe-coding/README.md

Idiomatic Vibe Coding Workflow

Follow the vibe coding[a] steps except as described below:

- In your first prompt, attach test-suite.F90 with the lines from `module procedure laplacian` to `end procedure laplacian` removed. If the LLM rejects ".F90" file extensions, change the name to "test-suite.txt". Append following to the prompt: "Inserting a correct response to this prompt into the subdmodule `subdomain_s` and then compiling and running the attached program must generate output indicating that all tests pass."
- In subsequent iterations, if replacing the laplacian procedure with the LLM's most recent response leads to compile-time or runtime errors or if any tests fail, update the test-suite program, replacing laplacian with the most recent LLM response. Attach the updated program to each prompt.
- In the second bullet of vibe coding step 6, replace the suggested text with "Please fix the above errors that were generated by compiling the attached program, which contains an incorrect laplacian procedure." If the program compiled but runtime errors resulted, replace "compiling" with "running" in the previous sentence. If the program ran without errors, but tests failed, replace "compiling" with "running" and replace "errors" with "test failures".
- Skip the third bullet in vibe coding step 6 because with idiomatic vibe coding, the complete software stack is contained in an attachment.

[a]#vibe-coding

Julienne Tests in Matcha

```
associate(dt => T%dt_stable(alpha))
  do step = 1, steps
    T =  T + dt * alpha * .laplacian. T
  end do
end associate

associate(residual => T%values() - T_steady)
  test_diagnosis = .all. ((residual .isAtLeast. 0.) .and. (residual .isAtMost. tolerance))
end associate
```

Steady state solution

```
  associate( phi_f => phi_functional(), phi_p => phi_procedural())
    associate(L_infinity_norm => maxval(abs(phi_f - phi_p)))
      test_diagnosis = .all. (phi_f .approximates. phi_p .within. tolerance)
    end associate
  end associate
```

Point-wise (elemental) comparison of functional and traditional procedural solution.

Julienne Tests in Matcha

```
1  associate(geometrical_properties => [ &
2    internally_zero, constant_away_from_edges, concave_at_faces
3    ,doubly_concave_at_edges, triply_concave_in_corners])
4
5    test_diagnosis = test_diagnosis_t( &
6       test_passed = all(geometrical_properties) &
7      ,diagnostics_string = "expected T,T,T,T,T, actual " // .csv. string_t(geometrical_properties) &
8    )
9  end associate
```

Concavity test

Results: Vibe Coding

1. **Anthropic Claude 4.0 Sonnet (42-line compile-time error):**
    a) 1st iteration: compile-time errors (syntax errors in do concurrent and private attributes.
    b) 2nd iteration: 2 of 3 tests pass!

3. **Google Gemini 2.5 Pro (155-line compile-time error):** Exhibited errors from misplaced private and public attributes and module statements improperly placed after contains, where procedure definitions go. Additional issues were invalid variable declarations, misused import statements, and undeclared result variables.

5. **OpenAI ChatGPT 4.1 (72-line compile-time error):** Similar placement errors as Gemini with incorrectly positioned module statements, invalid variable declaration locations, and invalid cycle statements within do concurrent.

7. **xAI Grok 3 (215-line compile-time error):** The most extensive errors, including similar code placement issues, type resolution errors for symbols like real64 duplicate interface definitions, argument mismatches, and submodule organization problems.

Results: Idiomatic Vibe Coding

Anthropic Claude 4.0 Sonnet:

1. Because only Claude provided code that compiled without error during VC, we proceeded to IVC only with Claude.
2. With IVC, Claude produced code that compiled without error after the first prompt, which suggests that the additional context offered with IVC improved the response.
3. 1st iteration:
    a) Code compiled (progress)
    b) 1 of 3 tests passed (regression).
4. 2nd iteration:
    a) compile-time errors (further regression).

Conclusions

- Vibe Coding with 4 LLMs
  - After 2 iterations, only Claude 4.5 Sonnet produced compilable code.
  - Claude code passed 2 of 3 tests.
- Idiomatic Vibe Coding with Claude 4.5 Sonnet
  - Code compiles on 1st iteration (progress)
  - 1 of 3 tests pass (regression)
  - Compile-time errors on 2nd iteration (further regression)

Thank You