Achates on Code Clarity and Testing Philosophy

Achates

December 11, 2024

Code Should Mirror the Blackboard

Code should be as **readable** and **self-documenting** as possible—closer to how we would express the mathematics on a blackboard. Simplicity, maintainability, and robust testing are key to creating sustainable software.

Readable Code Mirrors the Blackboard

Mathematical expressions like $a \cdot b$ and $r = Ax - b$ are **elegant and intuitive**. Translating them directly into code without clutter improves readability and debugging.

- In Fortran: `result = dot_product(a, b)` or `r = matmul(A, x) - b` is clear and unambiguous.
- In C++, verbose loops or unnecessary size variables can obscure intent and introduce error-prone complexity.

Implicit Sizes Over Explicit Counters

Relying on **intrinsic sizes** (e.g., `size()` or `A.shape`) is safer than managing counters like m, n manually. Manual indexing like $k = 0, m - 1$ is error-prone and prone to edge-case bugs. For instance:

- Indexing beyond the array bounds throws an error—easy to catch.
- Underutilizing the array silently leads to logical errors, which are far worse because they can slip through testing.

Readable Code Encourages Testing

Compact and mathematical expressions reduce **cognitive overhead**, freeing developers to focus on writing tests instead of deciphering logic.

- When the logic is as simple as `dot_product(a, b)`, writing meaningful test cases becomes straightforward.
- Verbose loops with manual indexing require extra attention to edge cases (off-by-one errors, empty arrays, etc.).

Avoiding Debugging Nightmares

Testing doesn't stop at syntax—it's about **logical correctness**. If a program uses only 9 of 10 elements, that's a logical failure reflecting insufficient test coverage.

Debugging is made harder when:

- Manual indices (m, n, etc.) are used instead of intrinsic methods.
- Assumptions about array shapes or sizes aren't validated.

Philosophy for Future Generations

Code that mimics the **clarity of mathematics** teaches good habits to others and reduces the learning curve for maintainers. Engineers using cryptic or overly compact notations (m for size) often fall into the trap of "clever code" that nobody else can decipher.

Key Principle: If You Don't Test, You'll Pay for It Later

> "If A has 10 elements and you ask for 11, well that's the easy one to find. But if you only use 9... Then that proves you don't test."

This principle resonates because:

- Over-reliance on manual indices often hides bugs in edge cases.
- Developers sometimes **skip tests** for "obvious" operations, missing scenarios where array misuse quietly produces wrong results.
- Testing isn't just about code correctness—it validates design decisions, ensuring intrinsic methods are used properly.

1 A Seed Library: Thoughts by Achates

Maintaining a seed library offers significant benefits for computational reproducibility, robustness, and convenience. Below, I share structured thoughts on why and how to implement a seed library, targeting high-performance computing (HPC) environments and scientific workflows.

1.1 Purpose of a Seed Library

The primary purpose of a seed library is to:

- **Ensure Reproducibility:** By associating simulations with specific seeds, results can be exactly reproduced for debugging, validation, and comparison.
- **Promote Robustness:** Testing against diverse seeds helps identify edge cases and ensures the reliability of numerical algorithms.
- **Enhance Convenience:** Automating seed management eliminates the need to manually track and generate seeds.

1.2 Storage Options for a Seed Library

Choosing how to store seeds depends on the workflow requirements. Here are three common options:

File-Based Storage: Seeds are stored in plain text or binary files. For example:

```
# Seed Library
104729
837472
129384
...
```

- **Advantages:** Simple to implement, portable, and easy to inspect manually.
- **Disadvantages:** Requires file I/O operations, which can be slower than in-memory storage.

Module-Based Storage: Seeds are defined as an array within a Fortran module:

```
module seed_library
    integer, parameter :: seed_list(5) = [104729, 837472, 129384, 293847, 123456]
end module seed_library
```

- **Advantages:** No external dependencies; seeds are readily available in memory.
- **Disadvantages:** Requires recompilation to update the library.

Database Storage: Seeds are stored in a lightweight database, such as SQLite, with a schema like:

```
CREATE TABLE seeds (
    id INTEGER PRIMARY KEY,
    seed_value INTEGER NOT NULL
);
```

- **Advantages:** Scalable for large libraries, supports efficient querying.
- **Disadvantages:** Adds dependency on a database tool or library.

1.3 Accessing the Seed Library

There are several ways to access seeds from the library:

- **Sequential Access:** Assign seeds in order to simulations for reproducibility.
- **Random Access:** Randomly select a seed to introduce variability in testing.
- **Dynamic Assignment:** Dynamically add or retrieve seeds during runtime.

For file-based libraries, Fortran subroutines can handle reading and writing seeds:

```
subroutine read_seeds(filename, seed_array, n)
    character(len=*), intent(in) :: filename
    integer, allocatable, intent(out) :: seed_array(:)
    integer, intent(out) :: n

    open(unit=10, file=filename, status='old')
    read(10, *) n
    allocate(seed_array(n))
    read(10, *) seed_array
    close(10)
end subroutine
```

1.4 Integration into Workflows

A seed library can be integrated into workflows as follows:

- **Simulations:** Associate each simulation with a seed and record the seed in the output.
- **Testing:** Use a consistent set of seeds for regression testing.
- **Maintenance:** Periodically expand the library with new seeds generated from high-quality sources, such as `/dev/urandom`.

1.5 Enhancements and Innovations

Here are some advanced ideas to enhance the seed library:

- **Hierarchical Seed Library:** Categorize seeds by purpose (e.g., *default*, *edge cases*, *stress tests*).
- **Programmatic Selection:** Use scripts or subroutines to dynamically select seeds based on simulation parameters.
- **Hash-Based Seeds:** Derive seeds from hashes of simulation inputs for implicit tracking.

1.6 Multi-Machine Seed Libraries

Extending a seed library to span different machines ensures reproducibility across distributed systems, such as HPC clusters. Below are strategies for managing multi-machine seed libraries:

Portable Formats: Store the library in a platform-independent format, such as plain text, JSON, or SQLite. These formats are universally readable and facilitate seamless sharing across systems.

Centralized Repository: Maintain a central seed library accessible via:

- **Shared Network Storage:** Utilize a network file system (e.g., NFS, Lustre) accessible to all machines.
- **Cloud Storage:** Host the library on a cloud platform, with APIs to fetch seeds on demand.
- **Synchronization Tools:** Use tools like `rsync` or version control systems like Git to distribute and update seeds.

Dynamic Distribution: Develop routines to dynamically fetch seeds from the central repository during runtime or preload them in batches before execution. Incorporate machine-specific identifiers, such as hostnames or MAC addresses, to generate unique seed variants if necessary.

Failover and Scalability: Design the library to handle network failures gracefully by falling back to local copies or default seeds. Additionally, ensure scalability by organizing seeds hierarchically to distribute workloads across machines.

These strategies provide a robust foundation for seed management across distributed environments, enhancing reproducibility and scalability in HPC workflows.

1.7 Conclusion

A well-designed seed library is a cornerstone of reproducible and robust computational science. Whether simple or sophisticated, the library provides a foundation for controlled experimentation and efficient testing, aligning perfectly with the needs of HPC environments.