The Journal of Open Source Software

Empirical: A scientific software library for research, education, and public engagement

Anya Vostinar [4], **Alexander Lalejini** [7], **Charles Ofria** [1,2,3], **Emily Dolson** [1,2,3], **and Matthew Andres Moreno** [4,5]

1 BEACON Center for the Study of Evolution in Action **2** Computer Science and Engineering, Michigan State University **3** Ecology, Evolutionary Biology, and Behavior, Michigan State University **4** Computer Science, Carleton College **5** Ecology and Evolutionary Biology, University of Michigan **6** Center for the Study of Complex Systems, University of Michigan **7** Computer Science, Grand Valley State University

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 14 December 2023
Published: unpublished

Summary

Empirical is a C++ library designed to promote open science and facilitate the development of scientific software that is efficient, reliable, and easily distributable to researchers and non-experts alike. Specifically, the library sets out to fulfill the following goals:

1. **Utility:** Empirical features a wide selection of helper tools to streamline common scientific computing tasks such as configuration, data management, random number generation, and mathematical manipulations. Documentation, examples, and project templates streamline developer onboarding.
2. **Efficiency:** Empirical tools emphasize efficiency to make heavy scientific computing workloads tractable. Where possible, computation is moved to compile-time, heap allocation is avoided, and memory-locality is respected.
3. **Reliability:** Scientific simulation and data analysis is meaningful insofar as it is correct. In conjunction with extensive unit testing, Empirical benefits from — and provides — sophisticated debug-mode instrumentation tools. These include audited memory management, drop-in replacements for standard C++ containers with safety checking, and self-documenting assertions.
4. **Distributability:** Scientific software should be easily used by any researcher, student, or citizen scientist who wants to experiment, explore, or replicate studies. As such, Empirical is highly portable, uses common data formats, facilitates flexible runtime configuration, and simplifies compilation to a performant web app.

In addition to many helpful utilities to improve the scientific programming experience, the core Empirical support library comprises four major features:

1. debug-instrumented fundamental type and C++ standard library container wrappers,
2. implementations of general-purpose data structures and algorithms,
3. integrated, end-to-end frameworks for data and configuration management, and
4. object-oriented bindings for Emscripten/WebAssembly GUI elements.

Statement of Need

Modern web-based interfaces give computational research the unique potential to embody open science objectives: they can make the scientific process more transparent with auditable and extensible code, clear and replicable methodologies, and production of accessible results (Woelfle et al., 2011). In practice, however, many scientific software applications are difficult to obtain, install, or use, and produce data in proprietary formats.

High quality open-science tools encourage researchers to follow effective software development practices by simplifying development and helping them improve code quality, scientific rigor, and ease of replication or extension. In the process, researchers get more out of their own code. For example, adding a GUI not only helps other users of the software, but could also help the researcher themselves gain "soft knowledge" of their system (Dudley & Kristensson, 2018).

Recent developments in web technology such as WebAssembly enable compilation of native source code to browser-based interactive interfaces. However, many scientists lack web development training and do not have the bandwidth to learn new languages and frameworks. Empirical catalyzes progress toward open science ideals by streamlining the development of in-browser software in C++.

To be reuseable and extendable, software must remain robust at scales and in contexts that were not originally envisioned. Empirical seeks to make writing correct, efficient scientific software easier. Empirical's debugging suite helps protect against common C++ programming pitfalls such as iterator invalidation, memory leakage, and out-of-bounds indexing. Bundled algorithms and data structures provide optimized, well-tested drop-in implementations for common scientific computing tasks. Throughout, library design obviates trade-offs between performance and safety; compile-time switches toggle safety checks for undefined or incorrect behavior.

Empirical Features

Facilitating Better Code for Scientific Software

Software produced by academics, especially for one-off use, often foregoes rigorous programming practices for the sake of expediency. By furnishing prepackaged components that address common tasks for scientific software, the Empirical library helps scientific developers efficiently write high quality C++ code. Utilities for common tasks empower users to craft more readable and maintainable code. Bugs can be avoided by replacing one-off implementations with Empirical components subjected to structured code review, unit testing (with coverage tracking), and other modern software development practices detailed in our documentation. Sustained effort invested into optimization of the library's utilities enables developers to produce efficient software at far less effort. Finally, off-the-shelf solutions reduce barriers to computational research, especially for developers new to scientific software design patterns.

To these ends, Empirical provides a comprehensive framework to manage runtime configuration and flexible tools for data aggregation and recording. For example, Empirical's configuration framework includes utilities to

- define and document default configuration values in a single line,
- set configuration values via a combination of a configuration file or command line flags,
- save configuration values to a file,
- perform on-the-fly configuration adjustments, and
- support multiple independent configuration subsystems.

Where appropriate, Empirical's scientific software tools include features that integrate directly into the browser environment. The configuration framework, for example, accepts input via URL query parameters and ties in with a pre-built, in-browser GUI for setting-by-setting adjustments.

In addition to the core C++ Library, we maintain a template project that streamlines laying out crosscompilation boilerplate via a command-line wizard.

High-quality software cannot succeed without a thriving community of engineers. As detailed in our documentation, our development practices encourage the meaningful inclusion of all interested contributors, and emphasize the importance of diverse backgrounds and perspectives in the community.

The Journal of Open Source Software

Realizing the Promise of Emscripten-based Web UIs

Educational or outreach versions of scientific software can provide accessible windows into contemporary scientific work, promoting classroom learning, social media engagement, and citizen science. Scientific software projects that want to reach these additional audiences must typically maintain multiple codebases. As the codebases diverge, it becomes time consuming to synchronize changes across them, a problem that is only compounded when also maintiaining crossplatform interfaces. These development costs preclude many scientific projects from providing easy access to the public. Even in better-resourced projects, this splintering effect absorbs limited developer hours and often leads to some versions of the code falling into neglect and drifting out of sync.

The Emscripten compiler promises to remedy this source splintering by enabling a single codebase to target web browsers alongside traditional native runtime environments (Zakai, 2011). Browser-based delivery can yield particularly effective public-facing apps due to widespread cross-platform compatibility, no-install access, and rich graphical interfaces. Native compilations are still required by most scientific applications, however, due to greater speeds and compatibility with high-performance computing environments.

Empirical amplifies the potential of Emscripten by fleshing out its rudimentary interface for interaction with browser elements. At the lowest level, Empirical provides tools for reciprocal data transfer between C++ code and the browser. DOM elements (such as `<button>`, `<div>`, and `<canvas>`) are given corresponding C++ objects (`emp::Button`, `emp::Div`, and `emp::Canvas`) and can be easily used from within C++ code. With these tools, users no longer need to manage JavaScript resources, and thus need much less preexisting web-programing knowledge. At a higher level of abstraction, Empirical packages pre-configured, pre-styled collections of DOM elements as prefabricated widgets (e.g., configuration managers, collapsible read-outs, modal messages, etc.). Empirical's tools aim to make generating a mobile-friendly, web-based GUI for existing software so trivial that the practice becomes ubiquitous.

Below, we give an example of Empirical's DOM interface in action. This example creates a button that increments an on-screen counter every time the button is clicked. You can view the resulting web page live at https://devosoft.github.io/empirical-joss-demo/.

C++ source:

```
#include "emp/web/web.hpp"

emp::web::Document doc("target");

int x = 5;
int main() {
  doc << "<h1>Hello World!</h1>";
  doc << "Original x = " << x << ".<br>";
  doc << "Current x = " << emp::web::Live(x) << ".<br>";

  // Create a button to modify x.
  emp::web::Button my_button( [](){ x+=5; doc.Redraw(); }, "Click me!" );
  doc << my_button;
}
```

HTML source:

```
<body>
  <div id="target"> </div>
</body>

<script src="https://code.jquery.com/jquery-1.11.2.min.js" integrity="sha256-Ls0pXSlb7AY
<script type="text/javascript" src="main.js"></script>
```

A live demo of more sophisticated Empirical widgets, presented alongside their source C++ code, is available on our prefab demos page.

Facilitating Runtime Efficiency

WebAssembly's runtime efficiency is a major driver of its increasing popularity for web app development. WebAssembly is a virtual bytecode, but just-in-time compilation engines translate critical sections into native machine code (Haas et al., 2017). This compilation model allows WebAssembly to achieve at least 50% — and at times closer to 90% — of native performance, providing an order of magnitude speed increase over JavaScript alone (Jangda et al., 2019). These performance improvements open the door to entirely new possibilities for browser-based scientific computation. For example, the Avida-ED web viewer at https://avida-ed.msu.edu/, uses WebAssembly to simulate hundreds of thousands of generations of digital organisms within the span of a class period. Such rich, intensive in-browser experiences necessitate efficient source code.

More broadly, Empirical caters to the necessity for runtime efficiency across all scientific computing, including in-browser applications, local runs, or high-performance computing (HPC) cluster deployments. In some contexts, even modest performance gains can save substantial hardware costs, meaningfully reduce energy use, and shave days or even weeks off run times. Order-of-magnitude performance gains can meaningfully broaden the scope of scientific questions that are tractable.

Empirical supports runtime efficiency in scientific computing by providing optimized tools for performance-critical tasks. For example, `emp::BitSet` and `emp::BitVector` classes are faster drop in replacements for their standard library equivilents (`std::bitset` and `std::vector<bool>`) while providing extensive additional functionality for rapid bit manipulations. Likewise,`emp::Random` wraps a cutting-edge high-performance pseudorandom number generator algorithm (Widynski, 2020). Benchmark-informed development practices ensure that optimizations translate into consistent performance enhancements. At a more fundamental level, Empirical's header-only design prioritizes ease of use and runtime performance at the cost of somewhat longer compilation times.

Facilitating Debugging

Identifying and correcting incorrect program behavior consumes a large fraction of developer hours for any software project. Software bugs that slip through into production can inflict even greater costs, especially in scientific contexts where the validity of generated data and analyses is paramount.

In conjunction with unit tests and integration tests, runtime safety checks are commonly used to flag potential bugs. Assert statements typify runtime safety checks. These statements abort program execution at the point of failure with a helpful error message if an expected runtime condition is not met. Runtime safety checks like `assert` don't necessarily oblige a performance cost to compute the asserted runtime condition; these checks can be verified only in debug mode and ignored in production mode to maximize performance.

Indeed, the C++ standard library's `assert` macro follows this paradigm. Empirical provides an extended `emp_assert` macro that prints custom error messages with current values of specified expressions, and dispatches a UI alert when triggered in a web environment.

In addition to user-defined asserts, most programming languages (Java, Python, Ruby, Rust, etc.) provide built-in support to detect common runtime violations, such as out-of-bounds indexing or bad type conversions.C++ does not in an effort to maximize performance. However, standard library vendors — like GCC's `libstdc++`, Clang's `libc++`, and Microsoft's `stl` — do provide some proprietary support for such safety checks. This support, however, is limited and

poorly documented[1]. Empirical supplements vendors' runtime safety checking by providing drop-in replacements for `std::array`, `std::optional`, and `std::vector` with stronger runtime safety checks, but only while in debug mode. In addition, Empirical furnishes a safety-checked pointer wrapper, `emp::Ptr`, that identifies memory leaks and invalid memory access in debug mode while retaining the full speed of raw pointers in release mode.

Because of poor support for built-in runtime safety checks, C++ developers typically use an external toolchain to detect and diagnose runtime violations. Popular tools include Valgrind, GDB, and runtime sanitizers. Although this tooling is very mature and quite powerful, there are fundamental limitations to the runtime violations it can detect. For example, Clang 12.0.0's sanitizers cannot detect the iterator invalidation described above (live example). Additionally, most of this tooling is not available when debugging WASM code compiled with Emscripten — a core use case targeted by the Empirical library. Although Emscripten provides some sanitizer support and other debugging features, tooling limitations (such as the lack of a steppable debugger) make runtime safety checking particularly critical.

Outlook and Future Plans

Empirical remains under active development. Current priorities include assembling higher-level web widgets for common tasks, making existing classes more web-friendly (such as file management and rich text handling), and adding more step-by-step tutorials to our documentation.

We are committed to maintaining a stable interface for existing users. Last year, we took a major step towards fulfilling this objective on an ongoing basis by completing a major reorganization informed by best practices to expose sustainable, consistent API to our end-users. We maintain an extensive suite of unit tests and integration tests to ensure that continuing development retains backward compatibility. In addition, our software releases are archived on Zenodo in order to guarantee uninterrupted, perpetual access to our software for those who depend on it (Ofria et al., 2020).

Empirical has already been successfully incorporated into major projects within our research group's primary domains: digital evolution, artificial life, and genetic programming. We aim for potential utility across a much broader swath of the scientific software ecosystem, particularly among projects that prioritize open science objectives. To this end, we look forward to welcoming new collaborations and supporting a wider collection of end-users.

Related Software Packages

Software Addressing Related Needs

There are many existing software platforms that provide functionalities overlapping with Empirical. However, most are not in C++, and there is value in this functionality being easily available to C++ programmers. See the Non-C++ Comparable Software section for citations to software platforms that provide some of Empirical's functionality in different languages.

RepastHPC

RepastHPC, accessible at https://repast.github.io/, is a C++ modeling framework targeted at large computing clusters and supercomputers(Collier & North, 2013; North et al., 2013). A Java-based counterpart, Repast Simphony, provides interactive GUI support. As such, simultaneous on-cluster and in-browser support requires maintenance of two separate code bases.

[1]For example, neither GCC 10.3 nor Clang 12.0.0 detect `std::vector` iterator invalidation when appending to a `std::vector` happens to fall within existing allocated buffer space (GCC live example; Clang live example).

Boost C++ Libraries

Boost C++ Libraries, available at https://www.boost.org/, provide an enormous range of software components. Several of the Boost libraries have been already incorporated into the C++ standard, and we build off of those with Empirical. However, Boost does not contain libraries for web-based GUI tools, configuration management, or data management specifically tailored to scientific software.

Emscripten

Emscripten is available at https://emscripten.org/ (Zakai, 2011). It provides cross-compilation from C++ to WebAssembly and we use it in Empirical. Empirical's tools build abstractions from Emscripten intrinsics tailored to visualization and interactive control of scientific simulations.

Cheerp

Cheerp, another C++ to WebAssembly compiler, is available at https://leaningtech.com/cheerp/. Like Emscripten, Cheerp provides primariliy low-level APIs for interaction with browser GUI elements.

Non-C++ Comparable Software

- TinyGo
- WebIO
- GWT
- yew
- Pyodide (Droettboom & Developers, 2021)
- Shiny (Chang et al., 2020)

Projects Using the Software

- Aagos (Gillespie et al., 2018)
  - An interactive model for empirically studying how evinronmental change affects the evolution of genetic architectures.
- conduit (Moreno et al., 2020)
  - A C++ library that wraps intra-thread, inter-thread, and inter-process communication in a uniform, modular, object-oriented interface, with a focus on asynchronous high-performance computing applications.
- Dishtiny (Moreno & Ofria, 2019)
  - An interactive web demo of an artificial life platform built to study major transitions in evolution.
- ecology in evolutionary computation explorer (Dolson & Ofria, 2018)
  - Interactive exploration of interaction networks in evolutionary computation under different selection schemes.
- Symbulation (Vostinar, 2017)
  - An interactive artificial life model focused on the evolution of symbiosis between parasitism to mutualism.
- SignalGP (Lalejini & Ofria, 2018)
  - SignalGP is a new GP technique designed to give evolution direct access to the event-driven programming paradigm where computations are triggered response to signals from the environment, from other agents, or that are internally generated.
- Model of cancer evolution on an oxygen gradient
  - A companion model to a series of wet lab experiments on cancer evolution in spatially heterogenous environments

The Journal of Open Source Software

Acknowledgements

This research was supported in part by NSF grants DEB-1655715 and DBI-0939454, by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1424871, by Michigan State University through the computational resources provided by the Institute for Cyber-Enabled Research, and by the Eric and Wendy Schmidt AI in Science Postdoctoral Fellowship, a Schmidt Futures program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Chang, W., Cheng, J., Allaire, J., Xie, Y., & McPherson, J. (2020). *Shiny: Web application framework for r*. https://CRAN.R-project.org/package=shiny

Collier, N., & North, M. (2013). Parallel agent-based simulation with repast for high performance computing. *SIMULATION*, *89*(10), 1215–1235. https://doi.org/10.1177/0037549712462620

Dolson, E., & Ofria, C. (2018). Ecological theory provides insights about evolutionary computation. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 105–106. https://doi.org/10.1145/3205651.3205780

Droettboom, M., & Developers, P. (2021). *Shiny: Web application framework for r*. https://pyodide.org/

Dudley, J. J., & Kristensson, P. O. (2018). A review of user interface design for interactive machine learning. *ACM Trans. Interact. Intell. Syst.*, *8*(2). https://doi.org/10.1145/3185517

Gillespie, L., Dolson, E., Lalejini, A., & Ofria, C. (2018). Changing environments drive the separation of genes and increased evolvability in NK-inspired landscapes. *Late Breaking Abstract at The 2018 Conference on Artificial Life*.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). Bringing the web up to speed with WebAssembly. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 185–200. https://doi.org/10.1145/3062341.3062363

Jangda, A., Powers, B., Berger, E. D., & Guha, A. (2019). Not so fast: Analyzing the performance of webassembly vs. Native code. *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 107–120. ISBN: 9781939133038

Lalejini, A., & Ofria, C. (2018). Evolving event-driven programs with SignalGP. *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18*, 1135–1142. https://doi.org/10.1145/3205455.3205523

Moreno, M. A., & Ofria, C. (2019). Toward Open-Ended Fraternal Transitions in Individuality. *Artificial Life*, *25*(2), 117–133. https://doi.org/10.1162/artl_a_00284

Moreno, M. A., rodsan0, perryk12, & Badger, C. (2020). *mmore500/conduit: Initial release* (Version v0.1.0). Zenodo. https://doi.org/10.5281/zenodo.4118608

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., & Sydelko, P. (2013). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, *1*(1), 1–26. https://doi.org/10.1186/2194-3206-1-3

Ofria, C., Moreno, M. A., Dolson, E., Lalejini, A., rodsan0, Fenton, J., perryk12, Jorgensen, S., hoffmanriley, grenewode, & al., et. (2020). *Devosoft/empirical*. https://doi.org/10.5281/zenodo.2575606

Vostinar, A. E. (2017). *Suicide, signals, and symbionts: Evolving cooperation in agent-based systems*. Michigan State University. ISBN: 978-0-355-07992-0

Widynski, B. (2020). Squares: A fast counter-based RNG. *arXiv Preprint arXiv:2004.06278*. https://arxiv.org/abs/2004.06278v3

Woelfle, M., Olliaro, P., & Todd, M. H. (2011). Open science is a research accelerator. *Nature Chemistry*, *3*(10), 745–748. https://doi.org/10.1038/nchem.1149

Zakai, A. (2011). Emscripten: An LLVM-to-JavaScript compiler. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 301–312. https://doi.org/10.1145/2048147.2048224