




wrenfold: Symbolic code generation for robotics

Gareth Cross ¹

¹ Independent Researcher, USA

DOI: [10.21105/joss.07303](https://doi.org/10.21105/joss.07303)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: Sébastien Boisgérault 

Reviewers:

- @ushu
- @abougouffa

Submitted: 18 August 2024

Published: 25 January 2025

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

Summary

Real-time robotic software systems often solve one or more numerical optimization problems. For example, accurate estimates of past vehicle motion are typically obtained as the solution of a non-linear optimization or filtering problem ([Barfoot, 2024](#)). Similarly, the behavior of an autonomous system can be selected via a numerical optimization problem that reasons about the relative merits of different future actions ([Lynch & Park, 2021](#)).

Problems of this form can be solved using packages like Google Ceres ([Agarwal et al., 2023](#)) or GTSAM ([Dellaert & Contributors, 2022](#)). These optimizers require that the user provide a mathematical objective function and - in some instances - the derivatives of said function with respect to the desired decision variables. In order to achieve real-time deadlines, the optimization is usually implemented in a performant compiled language such as C++.

wrenfold is a framework that converts symbolic math expressions (written in Python) into generated code in compiled languages (C++, Rust). The primary goals of the framework are:

- Bridge the gap between expressive prototyping of objective functions in symbolic form, and the performant code required for real-time operation.
- Improve on existing symbolic code generation solutions by supporting a greater variety and complexity of expressions.

Statement of need

Researchers and engineers working in robotics and related domains (eg. motion planning, control theory, state estimation, computer vision) regularly implement numerical optimization problems. This process presents a number of challenges:

- Robotic systems often feature non-trivial kinematics and dynamics. Describing the motion or sensor models may involve reasoning about complex chains of 3D transformations.
- Many popular optimization methods require derivatives for the objective function. Manually computing derivatives is tedious and error-prone, particularly in the presence of complicated geometry or compounded transformations.
- Performant systems languages like C++ do not necessarily have syntax conducive to the elegant expression of elaborate mathematical functions.

Symbolic code generation can help address these issues:

- Symbolic functions can be easily composed in Python, and reasonably performant¹ C++ implementations are obtained automatically. The developer time cost required to experiment with different optimization parameterizations is thereby reduced.
- Correct derivatives require no additional work - they can be obtained directly from the objective function via symbolic differentiation. Common terms that appear in

¹A comparison with handwritten and auto-diff based implementations is accessible at <https://wrenfold.org/performance.html>.

the objective function and its Jacobians are automatically de-duplicated by the code generation step.

Compilable source code is a suitable output format because it is straightforward to integrate into downstream projects. By customizing the code generation step, any number of additional languages and development environments can be targeted. Generated functions can be applied to a number of use cases, such as:

- Defining measurement expressions for a factor graph or Kalman filter.²
- Expressing terms in a motion planning or control optimization.³

Symbolic code generation has been shown to be an effective tool in robotics. For example, the MATLAB symbolic code generation toolbox has been applied directly to motion planning (Hereid & Ames, 2017). The open-source SymForce framework (Holtz, 2022) couples the SymEngine (Fernando, 2024) mathematical backend with Python code generation utilities and mathematical primitives specific to robotics.⁴ wrenfold draws inspiration from the design of SymForce, while aiming to support a greater variety and complexity of functions. We improve on the concept with the following contributions:

- **Symbolic functions may incorporate piecewise conditional statements** - these produce if-else logic in the resulting code. This enables a broader range of functions to be generated.
- **Emphasis is placed on ease of adaptability of the generated code.** Math expressions are simplified and converted into an abstract syntax tree (AST). Formatting of any element of the AST (such as function signatures and types) can be individually customized by defining a short Python method.
- **Times for code generation are meaningfully reduced**, thereby enabling more complex expressions.

wrenfold aims to support researchers and engineers in robotics by bringing symbolic code generation to a greater variety and complexity of optimization problems.

Design

Internally, wrenfold can be thought of as four distinct parts:

1. A **symbolic math frontend** implemented in C++ that exposes a Python interface. Programmers implement a Python function in order to specify the mathematical operations they wish to generate.
2. The symbolic expression tree is converted into a flat **intermediate representation** (IR). The IR is manipulated in order to eliminate common sub-expressions. Additional optimizations can be performed at this stage - for example factorizing common terms out of sum-of-product expressions.
3. An **abstract syntax tree** is built from the simplified IR. This representation is intended to be generic, such that nearly any language can be emitted downstream.
4. A **code generation** step converts the AST into a compilable language like C++ or Rust. This stage is easily customizable from Python.

²Examples of integrating wrenfold functions into Ceres and GTSAM are provided at <https://github.com/wrenfold/wrenfold-extra-examples>.

³A toy example of a Model Predictive Control (MPC) algorithm that uses wrenfold is available at <https://github.com/gareth-cross/cart-pole-mpc>.

⁴SymForce has been deployed on production robots produced by Skydio, an American drone manufacturer (Martiros, 2022).

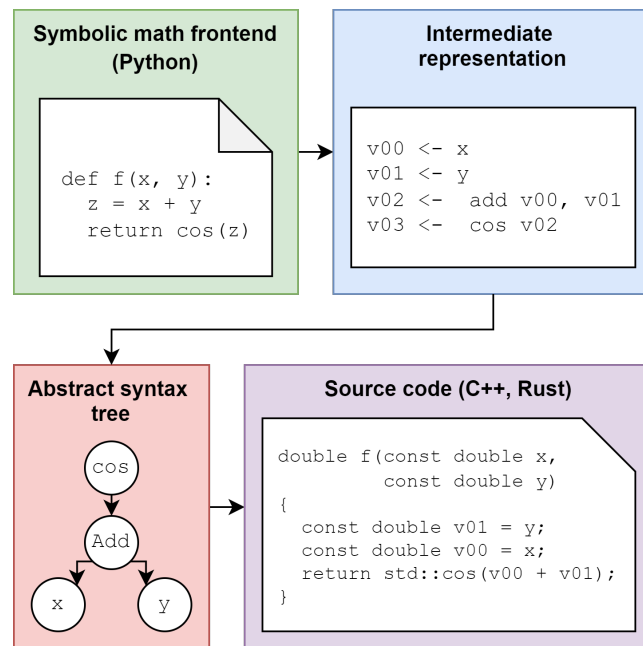


Figure 1: wrenfold system architecture

One of our design goals is to avoid burdening developers with long code generation times. To that end, we implement the core library in C++. The memoization pattern is employed throughout the symbolic backend in order to further improve performance. The fast backend allows wrenfold to scale to larger expression trees than a plain Python implementation (e.g. SymPy) would allow. wrenfold code generation times are also 20-100x faster than SymForce on representative functions.⁵

The user-facing Python API is implemented using pybind11 (Jakob et al., 2016). While code generators are written in C++, they can be inherited and overridden in Python. This facilitates easy adaptation of the generated code to a particular project - for instance:

- Injecting custom types into the signature of generated methods. For example, emitting code that interfaces with externally-provided geometry types like `gtsam::Pose3` (Dellaert & Contributors, 2022).
- Overriding a core math function (eg. `sin` or `cosh`) in order to invoke a custom implementation.
- Customizing syntax in order to suit a particular compiler or language version.
- Adding an entirely new programming language to the list of possible targets.

Examples of the above behaviors can be found in the wrenfold project repository.

Usage Example

We proceed with a usage example illustrating how a developer might interact with wrenfold. For the sake of brevity, this example is relatively simple. More realistic demonstrations can be found in the project repository. The following code was generated with wrenfold v0.2.2 - the newest version at the time of this writing. A symbolic function is created by writing a type-annotated Python function:

```
from wrenfold import code_generation, sym, type_annotations
```

⁵Based on a comparison of end-to-end code generation times of sample functions taken from <https://github.com/wrenfold/wrenfold-benchmarks>. We compared wrenfold v0.2.2 and SymForce 0.9.0.

```
def rotate_point(
    p: type_annotations.Vector2,
    theta: type_annotations.FloatScalar
):
    """Rotate a 2D point by the angle `theta`."""
    c, s = sym.cos(theta), sym.sin(theta)
    R = sym.matrix([(c, -s), (s, c)])
    p_rotated = R * p
    # Produce the rotated point, and the 2x1 Jacobian with respect to `theta`.
    return (
        code_generation.OutputArg(p_rotated, name="p_out"),
        code_generation.OutputArg(
            sym.jacobian(p_rotated, [theta]), name="p_out_D_theta", is_optional=True
        )
    )

# Generate C++ code:
code = code_generation.generate_function(
    rotate_point, generator=code_generation.CppGenerator()
```

Our example function accepts a 2D vector p , and rotates it by the angle θ . The outputs consist of the rotated point, and the Jacobian with respect to θ . The type annotations `Vector2` and `FloatScalar` specify the numeric types and dimensions of the input arguments. We also annotate the output values to indicate how they should be returned. The equivalent generated C++ function for `rotate_point` is:

```
#include <wrenfold/span.h>

template <typename Scalar, typename T0, typename T2, typename T3>
void rotate_point(const T0 &p, const Scalar theta, T2 &&p_out,
                  T3 &&p_out_D_theta) {
    auto _p = wf::make_input_span<2, 1>(p);
    auto _p_out = wf::make_output_span<2, 1>(p_out);
    auto _p_out_D_theta = wf::make_optional_output_span<2, 1>(p_out_D_theta);

    const Scalar v002 = theta;
    const Scalar v003 = std::sin(v002);
    const Scalar v001 = _p(1, 0);
    const Scalar v007 = std::cos(v002);
    const Scalar v006 = _p(0, 0);
    const Scalar v012 = v003 * v006 + v001 * v007;
    const Scalar v009 = v006 * v007 + -(v001 * v003);
    if (static_cast<bool>(_p_out_D_theta)) {
        _p_out_D_theta(0, 0) = -v012;
        _p_out_D_theta(1, 0) = v009;
    }
    _p_out(0, 0) = v009;
    _p_out(1, 0) = v012;
}
```

The generated C++ is intended to be maximally type agnostic. Vectors and matrices are passed as generic types, such that a wide variety of linear algebra frameworks can be supported at runtime.⁶ Note that common sub-expressions `v009` and `v012` have been extracted into variables so that they may be reused.

⁶We provide instructions for adding support for third-party matrix libraries at: https://wrenfold.org/reference/integrating_code. Support for Eigen is provided by default.

Future Work

wrenfold is designed to be extensible. Future avenues for improvement include:

- Adding new target languages, for example shader code (GLSL/HLSL) or CUDA.
- Extending the list of built-in core math functions.
- Generalizing the symbolic expression tree to reason about matrix expressions (thereby enabling vectorization of output code in some instances).

Resources

- GitHub repository: <https://github.com/wrenfold/wrenfold>
- Website: <https://wrenfold.org>

Acknowledgements

We acknowledge code contributions and feedback from: Himel Mondal, Anurag Makineni, Rowland O’Flaherty, and Chao Qu.

Agarwal, S., Mierle, K., & Team, T. C. S. (2023). *Ceres Solver* (Version 2.2). <https://github.com/ceres-solver/ceres-solver>

Barfoot, T. D. (2024). *State estimation for robotics*. Cambridge University Press. <https://doi.org/10.1017/9781316671528>

Dellaert, F., & Contributors, G. (2022). *Borglab/gtsam* (Version 4.2a8). Georgia Tech Borg Lab. <https://doi.org/10.5281/zenodo.5794541>

Fernando, I. et. al. (2024). *SymEngine*. <https://github.com/symengine/symengine>

Hereid, A., & Ames, A. D. (2017, September). FROST: Fast robot optimization and simulation toolkit. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. <https://doi.org/10.1109/iros.2017.8202230>

Holtz, H. M. A. A. M. A. N. B. A. B. S. A. R. K. A. J. Z. A. T. D. A. D. P. A. H. Z. A. T. T. A. P. H. A. G. C. A. J. V. A. A. S. A. S. W. A. K. (2022). SymForce: Symbolic Computation and Code Generation for Robotics. *Proceedings of Robotics: Science and Systems*. <https://doi.org/10.15607/RSS.2022.XVIII.041>

Jakob, W., Rhineland, J., & Moldovan, D. (2016). *pybind11 — seamless operability between c++11 and python*.

Lynch, K., & Park, F. C. (2021). *Modern robotics: Mechanics, planning, and control*. Cambridge University Press. <https://doi.org/10.1017/9781316661239>

Martiros, H. (2022). Open-sourcing SymForce. In *Skydio Blog RSS*. Skydio. <https://www.skydio.com/blog/open-sourcing-symforce/>