The Journal of Open Source Software

cppduals: a nestable vectorized templated dual number library for C++11

Michael Tesch[1]

1 Department of Chemistry, Technische Universität München, 85747 Garching, Germany

DOI: 10.21105/joss.01487

Software

- Review
- Repository
- Archive

Submitted: 13 May 2019
Published: 05 November 2019

License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC-BY).

Summary

Mathematical algorithms in the field of optimization often require the simultaneous computation of a function and its derivative. The derivative of many functions can be found automatically, a process referred to as automatic differentiation. Dual numbers, close relatives of the complex numbers, are of particular use in automatic differentiation. This library provides an extremely fast implementation of dual numbers for C++, `duals::dual<>`, which, when replacing scalar types, can be used to automatically calculate a derivative.

A real function's value can be made to carry the derivative of the function with respect to a real argument by replacing the real argument with a dual number having a unit dual part. This property is recursive: replacing the real part of a dual number with more dual numbers results in the dual part's dual part holding the function's second derivative. The `dual<>` type in this library allows this nesting (although we note here that it may not be the fastest solution for calculating higher order derivatives.)

There are a large number of automatic differentiation libraries and classes for C++: adolc (Walther, 2012), FAD (Aubert & Di Césaré, 2002), autodiff (Leal, 2019), ceres (Agarwal, Mierle, & others, n.d.), AuDi (Izzo, Biscani, Sánchez, Müller, & Heddes, 2019), to name a few, with another 30-some listed at (autodiff.org Bücker & Hovland, 2019). But there were no simple single-file stand-alone header libraries that were explicitly vectorized. *Cppduals* can be copied and used as a single header file `duals/dual` with no dependencies other than the standard library, and the vectorization support is contained in a small number of additional auxiliary files. The interface generally follows the conventions of the C++11 standard library's `std::complex` type, and has very liberal licensing.

In order to fully utilize the computing resources of modern CPUs it is often necessary to use their special data-parallel capabilities. However, compilers often struggle to find the best sequence of instructions to achieve maximum performance due to difficulties in automatically detecting parallel data operations in programs and then mapping those parallel operations onto the CPU's data-parallel instructions. Some intervention from the programmer to exploit these special data-parallel instructions, combined with cache-optimized algorithms, can improve program execution speed significantly. This is done by *cppduals* through providing template specializations for selected C++ data types.

Template specializations that map the dual arithmetic directly to vector machine language are provided for the types typically used to calculate first-order derivatives: `dual<float>`, `dual<double>`, `std::complex<dual<float>>`, and `std::complex<dual<double>>`. These types and their algebras are expressed in hand-coded assembly for use with the Eigen (Guennebaud, Jacob, & others, 2010) matrix library. The vector operations for the `dual<>` type are then picked up automatically by Eigen's cache-optimized algorithms. The integration lets us achieve extremely fast performance for automatic forward, first-order differentiation of matrix expressions. Furthermore, by piggy-backing on Eigen's algorithms, the library inherits

future improvements and adaptations to other hardware platforms. The vector specializations are currently available for the x86_64 SSE and AVX instruction sets, though support for other vector machine architectures (such as GPUs) can easily be added.

Mathematics

A dual number has a *real part* and a *dual part*, the *dual part* is multiplied by the dual unit ϵ, with the property that $\epsilon^2 = 0$. Formally the dual space, $\mathbb{D} = \{x = a + b\epsilon \mid a, b \in \mathbb{R}, \epsilon^2 = 0\}$.

The property can be used to differentiate functions of real values by expanding $f(x)$ as its Taylor series at a:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + ...$$

Setting $x = a + b\epsilon$ in this expansion gives

$$\begin{aligned} f(a+b\epsilon) &= f(a) + f'(a)(b\epsilon) + \frac{f''(a)}{2!}(b\epsilon)^2 + \frac{f'''(a)}{3!}(b\epsilon)^3 + ... \\ &= f(a) + f'(a)(b\epsilon) \end{aligned}$$

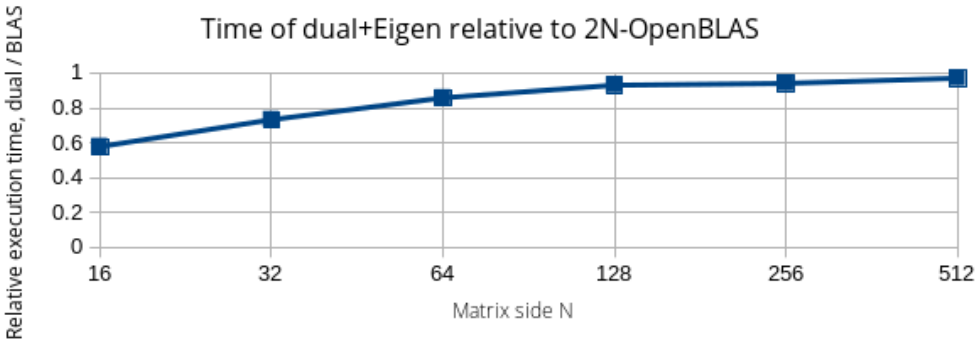
Because the ϵ^2 terms are zero, the Taylor expansion truncates after the f' term. Thus to evaluate $f'(a)$: set $x = a + b\epsilon$ with $b = 1$, and take the dual part of the evaluated function $\mathfrak{Du}(f(x))$.

A matrix representation of the dual numbers is $\left(\begin{smallmatrix} a & b \\ 0 & a \end{smallmatrix}\right)$. This form can be used to calculate Fréchet derivatives of matrix functions $D_B f(\mathbf{A})$ by doubling the size of the calculation and embedding the parameter (A) and with-regard-to parts (B) into this doubled matrix, zeroing the lower left quadrant: $D_B f(\mathbf{A}) = C$ where $f\left(\begin{smallmatrix} A & B \\ 0 & A \end{smallmatrix}\right) = \left(\begin{smallmatrix} F & C \\ 0 & F \end{smallmatrix}\right)$.

Benchmarks

The above blocking method of computing matrix function derivatives requires twice as much memory as using dual numbers, and furthermore does not take advantage of the relative sparsity of dual number arithmetic. However, the blocking method *does* permit the use of highly optimized complex-valued BLAS libraries, which are often significantly faster than "regularly" compiled code, being hand-tuned for a specific CPU architecture and memory heirarchy. We compared using `dual<>` to calculate the derivative of the matrix-matrix product with using the blocking method implemented with OpenBLAS (Wang, Zhang, Zhang, & Yi, 2013). All benchmarks were performed single-threaded on an *Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz* running *Fedora 30*, using *Eigen v3.3.7*, *OpenBLAS v0.3.6*, and compiled using *clang v8.0.0*.

The *cppduals* vectorized type reduced the time of a 32x32 matrix-multiplication derivative by 40% over the 2Nx2N (in this case 64x64) multiplication performed by OpenBLAS (even with a suspected performance bug in Eigen's GEMM algorithm, described below). In the current implementation, this advantage diminishes as the matrix size grows, as shown in Figure 1



we suspect this is due to a bug in Eigen's cache optimization for non-scalar valued matrix-matrix multiplication. We note that the Eigen scalar-valued matrix multiplications are roughly as fast as OpenBLAS, demonstrating the validity of Eigen's approach to optimization, but complex-valued multiplications take roughly twice as much time as their OpenBLAS equivalents, indicating a performance bug in Eigen's current optimizations, though only for complex-valued (and consequently, dual-valued) matrices.

We hope to achieve further speed improvements with tuning and more debugging of the integration with Eigen. In general, dual-valued operations should be marginally faster than corresponding complex-valued operations, as they require slightly fewer floating point operations.

Usage

Cppduals is used in the SpinDrops (Tesch, Glaser, & Glaser, 2019) NMR and quantum computing software for optimization of pulse quality functions.

Acknowledgments

We acknowledge first of all the Eigen project and its contributors, upon which the vectorization was based. Some funding was provided by the TUM School of Education.

References

Agarwal, S., Mierle, K., & others. (n.d.). Ceres solver. http://ceres-solver.org.

Aubert, P., & Di Césaré, N. (2002). Expression templates and forward mode automatic differentiation. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, & U. Naumann (Eds.), *Automatic differentiation of algorithms: From simulation to optimization*, Computer and information science (pp. 311–315). New York, NY: Springer. doi:10.1007/978-1-4613-0075-5

Bücker, M., & Hovland, P. (2019). Autodiff.org. http://www.autodiff.org.

Guennebaud, G., Jacob, B., & others. (2010). Eigen v3. http://eigen.tuxfamily.org.

Izzo, D., Biscani, F., Sánchez, C., Müller, J., & Heddes, M. (2019, May). Darioizzo/audi: Update third party dependencies. Zenodo. doi:10.5281/zenodo.2677671

Leal, A. (2019). Autodiff. https://github.com/autodiff/autodiff.

Tesch, M., Glaser, N., & Glaser, S. J. (2019). SpinDrops 2.0. https://spindrops.org/.

Walther, A. (2012). Getting started with ADOL-C. In U. Naumann & O. Schenk (Eds.), *Combinatorial scientific computing*. Chapman-Hall CRC Computational Science.

Wang, Q., Zhang, X., Zhang, Y., & Yi, Q. (2013). AUGEM: Automatically generate high performance dense linear algebra kernels on X86 CPUs. In *Proceedings of the international conference for high performance computing, networking, storage and analysis (SC13)* (pp. 1–12). Denver, Colorado: ACM Press. doi:10.1145/2503210.2503219