Arbogast: Higher order automatic differentiation for special functions with Modular C

Isabelle Charpentier, Jens Gustedt

Arbogast
Higher order AD for special functions with Modular C

Isabelle Charpentier and Jens Gustedt

RESEARCH REPORT
N° 8907
Apr. 2016
Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--8907--FR+ENG

`Arbogast` Higher order AD for special functions with Modular C[*]

Isabelle Charpentier[†] and Jens Gustedt[‡]

Project-Team Camus

Research Report n° 8907 — version 4 — initial version Apr. 2016 — revised version Jan. 2018 — 21 pages

Abstract: This high-level toolbox for the calculus with Taylor polynomials is named after L.F.A. Arbogast (1759–1803), a French mathematician from Strasbourg (Alsace), for his pioneering work in derivation calculus.
`Arbogast` is based on a well-defined extension of the C programming language, Modular C, and places itself between tools that proceed by operator overloading on one side and by rewriting, on the other. The approach is best described as contextualization of C code because it permits the programmer to place his code in different contexts – usual math or AD – to reinterpret it as a usual C function or as a differential operator. Because of the type generic features of modern C, all specializations can be delegated to the compiler. The HOAD with `arbogast` is exemplified on families of functions of mathematical physics and on models for complex dielectric functions used in optics.

Key-words: automatic differentiation, differential operators, modular programming, C, contextualization, functions of mathematical physics

[*] Accepted for publication in *Optimization Methods and Software*
[†] ICube, CNRS and Université de Strasbourg, France
[‡] Inria, France

`Arbogast` DA d'ordre élevé avec Modular C pour des fonctions spéciales

Résumé :

Cette boite à outil pour le calcul avec les polynômes de Taylor est nommé aprs L.F.A. Arbogast (1759–1803), mathématicien français de Strasbourg, Alsace, pour son travail pionnier sur le calcul des dérivations.

`Arbogast` est basé sur une extension du langage de programmation C, Modular C, et se place entre des outil travaillant avec la surcharge d'opérateurs et ceux faisant de la réécriture. L'approche est mieux décrit en tant que contextualisation de code C, car il permet au programmeur de placer son code en contextes différents – habituellement mathématique ou DA – pour le réinterpréter comme fonction C usuelle ou comme opérateur différentiel. Due au caractéristiques de généricité de types du C moderne, toute spécialisation peut être déléguée au compilateur. La différentiation automatique à haut dégrée avec `arbogast` est exemplifiée avec des familles de fonction de physique mathématiques et avec des modèles de fonctions diélectriques complexes utilisées en optique.

Mots-clés : différentiation automatique, opérateurs différentielles, programmation modulaire, C, contextualisation, fonctions spéciales

1 Introduction and Overview

From the time that L.F.A. Arbogast [1, 2, 3] wrote the "Calcul des derivations", the higher-order derivation of compound mathematical functions has been extensively studied [4, 5]. Nowadays, see [6] for instance, the higher-order automatic differentiation (HOAD) of computer codes representing complex compound mathematical functions mainly relies on operator overloading as a technique for attaching well-known recurrence formulas to arithmetic operations and intrinsic functions of programming languages such as C++, FORTRAN 90 or Matlab. A list of packages that allow for the differentiation of C++ codes is provided on the AutoDiff site. Beyond that, the differentiation of linear solvers or linear transformations, fixed-point methods [7], nonlinear solvers [8, 9] and special functions of mathematical physics [10] requires a careful study to be accurate and efficient. Although general developments were proposed and automated for most of these issues, they are not systematically included in the existing AD tools.

Since decades, C is one of the most widely used programming languages [11] and is used successfully for large software projects that are ubiquitous in modern computing devices of all scales. To the best of our knowledge, the AutoDiff site only references possible usages of C++ operator overloading libraries on C codes that do not contain C-specific features, and the source transformation tool ADIC [12, 13] for differentiating C codes[1] up to the second order. In this paper, we discuss the implementation and validation of a modular HOAD library called `arbogast` dedicated to modern ISO C that includes second order operators for the HOAD of classical functions of mathematical physics [14, 15] and has the potential to integrate other of the specialized algorithms, eventually.

C is undergoing a continued process of standardization and improvement and, over the years, has added features that are important in the context of this study: complex numbers, variable length arrays (VLA), `long double`, the `restrict` keyword, type generic mathematical functions (all in C99), programmable type generic interfaces (`_Generic`), choosable alignment and Unicode support (in C11), see [16]. Contrary to common belief, C is not a subset of C++. Features such as VLA, `restrict` and `_Generic` that make C interesting for numerical calculus do not translate to C++. Moreover, its static type system, fixed at compile time, and its ability to manage pointer aliasing make C particularly interesting for performance critical code. These are properties that are not met by C++, where dynamic types, indirections and opaque overloading of operators can be a severe impediment for compiler optimization. Unfortunately, these advantages of C are met with some shortcomings. Prominent among these is the lack of two closely related features, modularity and reusability, that are highly desirable in the context of automatic differentiation.

To propose a HOAD tool for C, we consider an extension to the C standard called *Modular C* [17] that enables us to cope with the identified lack of modularity and reusability. *Modular C* consists in the addition to C of a handful of directives and a naming scheme transforming traditional translation units (TU) into *modules*. The goal of this paper is to prove that this extension allows us to implement efficient, modular, extensible and maintainable code for automatic differentiation, while preserving the properties of C that we appreciate for numerical code. Named `arbogast`, the resulting modular AD tool we present provides a high-level toolbox for univariate calculus with Taylor polynomials. For a method to extend this approach to multivariate tensors the interested reader is refered to [18, 10]. It places itself between tools that proceed by operator overloading on one side, and by rewriting on the other. The approach is better described as *contextualization* or *reinterpretation* of code. The HOAD with `arbogast` is exemplified on models for complex dielectric functions, one of them comprising a function of mathematical physics.

This paper is organized as follows. The aims and abilities of Modular C are introduced in Section 2 and illustrated on an implementation of a generalized Heron's method to compute ν^{th} roots. The new AD library, `arbogast`, is described in Section 3. Subsequently we demonstrate the usefulness of our HOAD tool by designing a differential operator (DO) devoted to the solution for the general 2nd order ODE satisfied by most of the functions of mathematical physics in Section 4, then an application to optics in Section 5. All these code examples are accompanied by benchmarks that prove the efficiency of our approach. Section 6 presents some conclusions and outlooks.

2 *Modular C*

For many programmers, software projects and commercial enterprises C has advantages – relative simplicity, faithfulness to modern architectures, backward and forward compatibility – that largely outweigh its shortcomings. Among these shortcomings is a lack of modularity and reusability due to the fact that C misses to encapsulate different translation units. All symbols of all used interfaces are shared and may clash when they are linked together into an executable. A common practice to cope with that difficulty is the use of naming conventions. Usually software units (*modules*) are attributed a name prefix that is used for all data types and functions that constitute the programmable interface (API). Such naming conventions (and more generally

[1]ADIC only handles the historic C89 version, often referred to as "ANSI C".

coding styles) are often perceived as a burden. They require a lot of self-discipline and experience, and C is missing features that would support or ease their application.

2.1 Modularity for AD and Taylor polynomials

Because of its requirement for code efficiency, AD can profit substantially from code written in C. Unfortunately though, an AD tool has a strong need for genericity and modularity since it has to be able to deal with numerical codes that possibly involve 6 different floating point types (`float`, `double`, `long double` and their complex variants). Then, 3 binary operations on Taylor polynomials have to be implemented covering the case where both operands are polynomials, but also for the case that one of them is a scalar. In a complete support library for AD we would have to produce similar code for the combination of all 6 floating types and 3 combinations of operands, so 18 replicas in total. This growing code complexity gets even worse when we want to special-case certain types of operations, for instance to take advantage of polynomials with different degrees.

Other programming languages than plain C are able to cope better with such a combinatorial explosion of cases. In particular, *C++* offers implicit type conversion and template programming that can (and have been) used to implement operators generically. A typical C++ implementation would first implement Taylor-to-Taylor operators as template. This would give rise to only 6 instantiations. When such an operator is called by user code, the arguments, Taylor or scalar, would be converted to a common “super” type `T`, then the appropriate operator for `T` would be called.

Whereas this quickly provides a solution for the problem, such code is generally not as efficient as it could be. A lot of intermediate values (Taylor polynomials) will be produced during execution, all arithmetic will always be performed in the wider and more expensive version, and if objects are passed by reference (or pointer) aliasing restrictions can undermine optimization. Moreover, when using templates, C++ typically would defer the compilation of a particular instantiation of a template to the compilation that *uses* it, leading to prohibitive compilation times of user code. Although partial solutions exist to circumvent these problems, their general effect may result in a degradation of the modularity properties that had motivated the choice for C++ in the first place.

Listing 1: Heron’s method for the approximation of $\sqrt[\nu]{x}$. This C version is backwards compatible with *Modular C*.

```c
#include <stddef.h>                                // standard definitions
#include "powk.h"                                  // powk, abs2, epsilon and imax

double phi(double x, unsigned nu) {
  if (nu == 1) return x;                           // special case
  double const chi = 1/x;
  double rho = (1+x)/nu;                           // initial estimation
  for (size_t i = 0; i < imax; ++i) {
    double rhonu = powk(rho, nu-1);                // powk(x, k-1) = x^{k-1}, k integer
    if (abs2(1-rho*rhonu*chi) < epsilon) break;// abs2(x) = powk(x, 2) = x^2
    rho = ((nu-1)*rho + x/rhonu)/nu;               // next Heron iteration value
  }
  return rho;
}
```

Below, we exemplify the transition from C to *Modular C*, then to modular AD with `arbogast` with a small toy example, an implementation of a generalized Heron’s method to compute ν^{th} roots. Listing 1 shows concise C code that implements this method. It uses two external functions `powk` and `abs2` to compute integral powers and $\|.\|^2$. The latter is used as a convergence measure to compare against a bound `epsilon`.

2.2 Features of *Modular C*

Modular C is an add-on to the C programming language that targets to add modularity, reusability and encapsulation. Thereby it provides an improved support for software projects that have to deal with parameterized types and functions, and that have to specialize these for a large variety of base types. Since it is not a completely new programming language, code written within *Modular C* remains compatible with C and compiled libraries can easily be called from C or from other programming languages such as FORTRAN.

Modular C’s main tool are composed identifiers, the only addition of *Modular C* to the core language. All other features of *Modular C* then aim to support and ease the implementation of a hierarchical module structure for software projects that uses and enforces this naming scheme. These features of Modular C are implemented through `CMOD` *directives*. Particularly important for our discussion are three directives, namely `import`, `snippet` and foreach. We will discuss and exemplify them below in more detail. Another feature that is particularly interesting for AD is *contextualization*. It provides a replacement for operator overloading as we will see when we discuss arbogast in Section 3.

Additional features of *Modular C* are a dynamic module initialization scheme, a structured approach to the C library, a migration path for existing software projects and, last but not least, complete Unicode integration.

2.2.1 Composed identifiers

Composed identifiers are segmented by a user-chosable character[2]. The basic rules for interpretation of such an identifier are straightforward, the segmented prefix of an identifier corresponds to the module (translation unit) where that identifier can be found. For instance, C::io::printf refers to the printf function in the module C::io and arbogast::trd refers to the Taylor polynomial type trd (Taylor real double) in the arbogast module. As long as segmented identifiers are used in this *long* form, they can be used freely anywhere they make sense. All necessary information is encoded in that name and no #include or import directive is needed.

2.2.2 Import directive

Modules can import other modules as long as the import relation remains acyclic. As we already have mentioned above, such an import can be implicit if a *long* segmented identifier is used, or it can be explicit by means of an import directive. Other than traditional #include, import ensures complete encapsulation between modules.

The advantage of using the directive over implicit import is the abbreviation scheme. It allows to refer to all identifiers of another module with a *short* prefix, and it also allows to seamlessly replace an imported module by another one with equivalent interface.

Code as in Listing 1 is valid for *Modular C*, too, but we would not gain much by this. Using C's #include still propagates all names from the headers into our translation unit. Listing 2 shows a first implementation

Listing 2: Heron's method for the approximation of $\sqrt[\nu]{x}$. Using *Modular C* features, only.

```
#pragma CMOD module phi   = heron::droot        // name it
#pragma CMOD import powk  = heron::powk         // for powk and abs2

#pragma CMOD definition
double phi(double x, unsigned nu) {
  if (nu == 1) return x;
  double const chi = 1/x;
  double rho = (1+x)/nu;
  for (C::size i = 0; i < heron::imax; ++i) {
    double rhonu = powk(rho, nu-1);
    if (powk::abs2(1-rho*rhonu*chi) < heron::epsilon) break;
    rho = ((nu-1)*rho + x/rhonu)/nu;
  }
  return rho;
}
```

that uses only *Modular C* features: segmented identifiers and abbreviations to refer to standard features (C::size) or to features defined in other modules (heron::epsilon). Because we establish an abbreviation powk for heron::powk we can use the short form powk::abs2 for heron::powk::abs2.

2.2.3 Snippets

The snippet directive implements a mechanism that allows for code reuse, similar to so-called X macros or templates available in other programming languages. Listing 3 shows an excerpt of a snippet definition. The related slot directives are used to specify the parameters of a snippet. Then, fill directives will be used to instantiate these slots for a particular specialization of the snippet, see Listings 4 and 5.

Note that, in contrast to C++' template mechanism, a function in a snippet is compiled once per specialization only, *i.e.* when it is explicitly imported into another module with an import directive. The syntax is unambiguous and compilation can effectively use all type information that is available at the importing side.

As an example, Listing 3 shows a more sophisticated implementation of Heron's method as a snippet with slots SRC and φ. The first represents a floating point type, and the second the external name for the defined function. That snippet is specialized explicitly by importing it into another translation unit and by filling the slots, see Listing 4. The three directives, there, generate a function rtf that implements Heron's method for float.

[2]In the following we use the character ::, but such a character may be chosen separately for each module.

Listing 3: Generic method (real or complex) for one branch of $\sqrt[\nu]{x}$. From the snippet directive onward code is injected directly into importers. It reserves two *slots* that have to be *filled* by importers.

```
#pragma CMOD module root   = heron::snippet::root
#pragma CMOD import powk   = heron::powk        // for generic powk and abs²

#pragma CMOD snippet none                       // code only seen in importer
#pragma CMOD slot SRC = complete                // real or complex scalar type
#pragma CMOD slot φ   = extern SRC φ(SRC x, unsigned ν); // reserve name

SRC φ(SRC x, unsigned ν) {                      // greek names to greek symbols
  if (ν == 1) return x;
  SRC const χ = 1/x;
  SRC ρ = (1+x)/ν;
  for (C::size i = 0; i < heron::imax; ++i) {
    SRC ρν = powk(ρ, ν-1);                      // powk is a type generic function
    if (powk::abs²(1-ρ*ρν*χ) < heron::ε) break; // abs² is a type generic function
    ρ = ((ν-1)*ρ + x/ρν)/ν;
  }
  return ρ;
}
```

Listing 4: A specialization of the snippet in Listing 3. Slots are referred to by a name prefixed with the import abbreviation.

```
#pragma CMOD import rootf      = heron::snippet::rt  // name this import
#pragma CMOD fill   rootf::SRC = float               // assign a type to SRC
#pragma CMOD fill   rootf::φ   = rtf                 // name the new function
```

The second feature that we see in Listings 3 and 4, is the use of Unicode characters for most purpose, in particular as identifiers and in strings. As most modern platforms, *Modular C* uses UTF-8 for the source encoding. In the context of mathematics, and HOAD in particular, it allows to use notations that are much closer to the domain specifics than most C or FORTRAN platforms. Here, we are for example able to replace the spelling of Greek characters for variables by the characters themselves (*e.g* φ) and use superscripts where we think it is appropriate (*e.g* abs²).

2.2.4 Code replication

When implementing libraries that have to deal with similar code for different base types usually almost identical code is just replicated and adapted for the target type. In the context of AD, this would be C's six real and complex floating point types. Obviously, such a procedure is prone to subtle copy errors that are difficult to find by inspection. Modular C's `foreach` directive avoids these problems by allowing parameterized code replication that is completely resolved at compile time. This replication of code equally applies to statements, type or function declarations as to other *Modular C* constructs. Listing 5 shows how a specific code section is replicated four times. Here, identifiers in the replacement list for `T` are types acronyms.

In the example, four functions `heron::rt::φ‾srd`, `heron::rt::φ‾scd`, `heron::rt::φ‾srf` and `heron::rt::φ‾scf` are specialized for types `srd`, `scd`, `srf` and `scf`, respectively.

Observe that the two type generic functions `heron::powk` and `heron::powk::abs²` used in the snippet of Listing 3 can be defined to return values and types for the `complex` cases as one would expect: `heron::powk(z, κ)` is $z^\kappa \in \mathbb{C}$, `heron::powk::abs²(z)` is $\|z\|^2 \in \mathbb{R}$, the square of the complex norm.

These four functions are then assembled into one type generic macro `root` (short for `heron::root`) that uses C11's `_Generic` to choose the function that corresponds to the first argument. This feature, as all replacements by *Modular C*, guarantees that the corresponding function is chosen at compile time. No dynamic resolution or indirection occurs during execution.

3 Arbogast

Typically AD libraries are projects that are much in need of modularity, reusability and encapsulation. In addition, they have quite pressing demands about the efficiency of the resulting code. Our *Modular C* toolkit **arbogast** aims to realize an easy-to-use interface for AD that, at the same time, also results in highly efficient binary code. It provides support for all six C floating point types and all standard operations.

Listing 5: Four specializations of the generic method for one branch of $\sqrt[\varphi]{x}$ and a type generic interface.

```
#pragma CMOD module   root          = heron::root
#pragma CMOD composer ▀

#pragma CMOD declaration
typedef double          srd;
typedef _Complex double scd;
typedef float           srf;
typedef _Complex float  scf;

#pragma CMOD foreach T = srd scd srf scf
#pragma CMOD import  root▀${T}      = heron::snippet::root
#pragma CMOD fill    root▀${T}::SRC = ${T}
#pragma CMOD fill    root▀${T}::φ   = φ▀${T}
#pragma CMOD done

#define root(X, N)                                \
_Generic((X),                                     \
         srd: φ▀srd, scd: φ▀scd,                  \
         srf: φ▀srf, scf: φ▀scf)                  \
 ((X), (N))
```

3.1 Taylor polynomials and operations

As a major feature, `arbogast` implements 6 Taylor polynomial types and corresponding operations and functions [6, 303–308]. It is important to note here that these types are implemented as structure types that directly contain the coefficient array without any pointer indirection. We made this choice to accommodate modern compiler optimization that is able to track such array elements individually. As a consequence, our Taylor polynomials have a maximum degree.

The six types correspond to C's 6 floating point types that are used as coefficients for the polynomial. By default the maximal degree N is 31, which largely covers the practical needs for HOAD. For efficiency reasons, the Taylor types also maintain an actual degree of the instance they represent.

The principal implemented algebraic operations are polynomial addition $P(t)+Q(t)$, subtraction $P(t)-Q(t)$, product $P(t)\cdot Q(t)$, derivative $\frac{dP(t)}{dt}$, indefinite integral $\int P(t)dt$ and polynomial evaluation $P(t_0)$. Note that the division operator, $\frac{P(t)}{Q(t)}$, is not among these algebraic operations because the exact mathematical solution is in general only a rational function and not a polynomial.

These algebraic operation are not exact but only approximations. First, obviously, the coefficients are computed with the corresponding precision of the underlying C floating point type. But more importantly, the Taylor expansion is cut off at a maximum degree N. Of the operations that are listed above these are $P(t)\cdot Q(t)$ and $\int P(t)dt$ which are operations that result in polynomials of higher order than their input.

In addition to the basic arithmetic operators, `arbogast` also implements composition operators $f\circ P(t)$ for major numerical functions $f(x)$ that verify simple 1st-order differential equations (ODE), that can be expressed with the algebraic operations as dissussed above. Examples with simple ODEs are e^x, $1/x$, $\sqrt{x}$. Their implementations are based on the following well-known principles [1, 6]. For polynomials $P(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_N t^N$ and $f\circ P(t) \approx b_0 + b_1 t + b_2 t^2 + \cdots + b_N t^N$, the additive constant b_0, called *seed* in the sequel, is determined by the function value $b_0 = f(a_0)$. Then, a defining 1st-order differential equation for f together with the chain rule is used to subsequently compute $b_1, b_2, \ldots, b_N$ by equating coefficients. For example for $f(x) = e^x$ we know that $\frac{df(x)}{dx} = f(x)$ and thus that

$$b_1 + 2b_2 t + \cdots + N b_N t^{N-1} \quad\approx\quad (a_1 + 2a_2 t + \cdots + N a_N t^{N-1})\cdot(b_0 + b_1 t + b_2 t^2 + \cdots + b_N t^N)$$

Setting $a'_k = k a_k$ for $k > 0$ and expanding the product on the right hand side then leads to the solvable sequence of equations:

$$b_0 = e^{a_0}, \quad b_j = \frac{1}{j}\sum_{k=0}^{j-1} b_k a'_{j-k} \qquad \text{for } j = 1,\ldots N \tag{1}$$

Other functions come in pairs that are linked by ODEs such that we can develop the coefficients for the corresponding pair of operators simultaneously. These are $\sin x$ and $\cos x$, $\sinh x$, and $\cosh x$. Yet others operators for operations or functions such as y/x, $\log x$, x^ν, $\tan x$ or $\tanh x$ can then be combined algebraically from the above.

Observe that a formula such as (1) in general requires $\frac{N(N-1)}{2}$ multiplications, $\frac{(N-1)(N-2)}{2}$ additions, and $N-1$ divisions, so $N(N-1)$ floating point operations in total. For our case of $N = 31$ this corresponds to an overhead of 930 floating point operations per Taylor operator. As a consequence we can expect a "Taylor" program $\mathfrak{F}'$ that is derived from a conventional program $\mathfrak{F}$ by replacing floating point variables by Taylor polynomials to be several orders of magnitude slower than $\mathfrak{F}$.

But fortunately this overhead for AD is only additive, that is, any closed expression that uses n of the algebraic or numerical floating point operations from above should only encounter n times this overhead when transformed into an operator for Taylor polynomials. Therefore we will use the complexity of the most commonly used operators, namely the division operator, as a baseline for performance discussions, below.

3.2 Contextualization

The snippet in Listing 3 is type generic and could almost serve as a base to be used with `arbogast` for AD, just by "filling" with one of `arbogast`'s Taylor polynomial types instead of a floating point type. As C does not provide the possibility of overloading arithmetic operations, the type generic programming described above is only possible through functional notation.

One way to differentiate the numerical code in Listing 3 would be to manually rewrite the snippet such that all necessary arithmetic operators are replaced by function calls. For instance, we could replace each division that potentially could be a polynomial division by a call to `arbogast::div`.

Listing 6: A generic method for an approximation of one branch of $\sqrt[\nu]{x}$ or $\sqrt[\nu]{x_0 + x_1 t + x_2 t^2 + \cdots}$. The specialized context for arithmetic with the parameter type `RC` is coded inside `(:␣:)` brackets.

```
1  #pragma CMOD module          heron::snippet::rt
2  #pragma CMOD import   powk   = heron::powk          // scalars or Taylor types
3
4  #pragma CMOD snippet none
5  #pragma CMOD context  AD     = (:␣␣:)               // overload some expressions
6  #pragma CMOD slot     RC     = complete             // scalar or Taylor type
7  #pragma CMOD slot     φ      = extern RC φ(RC x, unsigned ν);
8
9  RC φ(RC x, unsigned ν) {
10   if (ν == 1) return x;
11   RC const χ = (:1/x:);                            // may be polynomial division
12   RC ρ = (:(1+x)/ν:);
13   for (C::size i = 0; i < heron::imax; ++i) {      // conventional operations
14     RC ρν = powk(ρ, ν-1);                          // no context needed
15     if ((:powk::abs²(1-ρ*ρν*χ)␣<␣heron::ε:)) break; // compare to a real value
16     ρ = (:((ν-1)*ρ␣+␣x/ρν)/ν:);
17   }
18   return ρ;
19 }
```

Modular C offers a easier way to do this: the `context` directive. This allows to choose opening and closing "*parenthesis*" that mark a special context in an expression and replace all occurrences of arithmetic operators by proper function calls. In Listing 6 we define a context, locally named `AD`, that starts with a `(:` and ends with a `:)`. Then we mark all places that might involve arithmetic with Taylor polynomials by these characters. For instance, the division in Line 11 could be either scalar arithmetic if `x` is a scalar, or polynomial division if `x` is a Taylor polynomial. Note that using the AD context for the convergence criterion in Line 15 is mainly for demonstrative purpose. The computation here only uses the 0^{th} Taylor coefficients.

Modular C allows to define several contexts inside the same module. The strings for opening or closing parenthesis can be chosen quite freely. Listing 7 shows a specialization of the snippet with `arbogast`'s six

Listing 7: Six specializations of the generic method for one branch of $\sqrt[\nu]{x_0 + x_1 t + x_2 t^2 + \cdots}$.

```
#pragma CMOD module     heron::inst::rt::AD
#pragma CMOD import     arbogast::taylor
#pragma CMOD composer ‾

#pragma CMOD foreach TRC             = trf tcf trd tcd trl tcl
#pragma CMOD import  rt‾${TRC}       = heron::snippet::rt
#pragma CMOD fill    rt‾${TRC}::AD   = arbogast::context
#pragma CMOD fill    rt‾${TRC}::RC   = ${TRC}
#pragma CMOD fill    rt‾${TRC}::φ    = φ‾${TRC}
#pragma CMOD done
```

Table 1: Complexity and run time for some linear operators and a vector length of N. "Memory" denotes the number of memory accesses or integer to floating point conversions, the next two columns are the numbers of floating point arithmetic, without and with fast `fma` instruction, and "total" is the total number of these primitives in case of presence of fast `fma`. "min ||" denotes a parallel lower bound for the arithmetic of the operation.

	operation	memory	$+\cdot$	$+\cdot$ `fma`	total	$N=32$	min $\|\|$	sec	cycles
assign	$C_i = A_i$	$2N$	0	0	0	64	0	7.2E-9	21.7
dot	$c \mathrel{+}= A_i \cdot B_i$	$2N$	$2N$	N	$3N$	96	$\log N$	3.1E-8	93.3
add	$C_i = A_i + B_i$	$3N$	N	N	$4N$	128	1	2.3E-8	70.2
smul	$C_i = c \cdot A_i$	$2N$	N	N	$3N$	96	1	2.0E-8	60.0
sma	$C_i \mathrel{+}= c \cdot A_i$	$3N$	$2N$	N	$4N$	128	1	2.2E-8	67.1
deriv	$C_i = (i+1) \cdot A_{i+1}$	$3N$	N	N	$4N$	128	1	3.3E-8	98.5
indef	$C_i = A_{i-1}/i$	$3N$	N	N	$4N$	128	1	2.6E-8	77.9
eval	$c = c \cdot x + A_i$	N	$2N$	N	$2N$	96	N/k	2.5E-8	73.5

Taylor types. The types are referred to by identifiers that are imported from `arbogast::taylor`, the context `AD` is filled with polynomial arithmetic from `arbogast::context`. Another module `arbogast::inst::rt::scalar` (not shown) can be used to instantiate functions for scalar arguments in a similar way: we just have to use `arbogast::scalar` for the types, and fill the context `AD` with `C::mod::trivial`. All twelve specialized function interfaces can then be grouped together in a type generic interface similar to the one in Listing 5, as promoted by C11.

3.3 Implementation

`Arbogast` is itself implemented in *Modular C* and takes much advantage of its features for composing operators from basic primitives. In addition to the algebraic operations on Taylor polynomials, there are primitives for the dot product, the vector reversion, the scalar multiply, and scalar multiply-add-in-place. Table 1 summarizes the vector operations with linear work that are at the heart of `arbogast`'s operators.

3.3.1 Experimental Setup

All run times that are given in the sequel are effected on commodity hardware comprising an Intel processor at a maximal frequency of 3.0 GHz with a builtin `AVX2` vector unit. The code is compiled with a `gcc 5.4` including `OpenMp SIMD` support[3].

To reliably measure operations that perform within a low number of CPU cycles is an art of its own. For instance, advanced compiler optimization makes it difficult to measure individual operations in vitro, because the compiler eliminates operations for which the results are never used in the sequel. We carefully avoided such issues by qualifying variables for results of operations as `volatile` and/or by chaining operations, that is using the output of one test iteration in the next. We carefully checked the intermediate assembler code to ensure that none of the operations that we tried to measure has been optimized out.

To avoid perturbation of the measurements by the timing itself, each experiment performs a tight loop that repeats a given operation at least 1000 times. Time measurements are taken outside of that loop. These experiments are then repeated 40 times to ensure a low variance. The times are then broken down to an average per operation or function. In addition to measuring the code for Taylor polynomials, we always measured the performance of "normal" mathematical operations. These proved the good general performance of the tool chain that we used: standard mathematical operations on `double` such as `+` or `*` contribute to 1 to 2 CPU cycles, standard mathematical functions such as `cos` to about 40 cycles.

3.3.2 Linear vector primitives

Much care has been taken to implement these primitives such that they perform very efficiently on modern architectures. In fact, modern CPUs are complex devices that provide a lot of low level parallelism. For instance, most architectures nowadays allow indirect addressing of assembler operands. Thus one assembler addition can perform an address operation, a load or a store and the addition itself. That does not mean that all of this is done in one CPU cycle, but that each CPU cycle can start an instruction pipeline for such a complex operation. Even more parallelism can be achieved with *vector units* as they can be found in most commodity hardware. In particular, modern Intel and AMD CPUs provide SIMD units (called SSE or AVX) that are able to work on 4 `float` or 2 `double` values simultaneously in one instruction. These vector units also have special

[3]This mode allows to have compiler support for SIMD vectorization, without refering to OpenMp's thread parallelization.

fast *fused multiply-add* (`fma`) instructions that in the case of `float` can perform 4 such operations, that is 8 FLOPs, in a single CPU instruction[4].

Table 1 shows the vector operations that are at the basis of Taylor polynomial operations as they are implemented by `arbogast` and their average execution times on a recent commodity platform. They have varying efficiency and in particular we see that the dot product operation is much slower than a scalar-multiply-add-in-place operation "sma". This is due to the fact that the former has a logarithmic parallel overhead for the computation of the final sum and that the latter can profit from a fast `fma` operation. Below we will see how this performance difference is used to speedup `arbogast` substantially compared to a direct implementation of formulas such as (1).

For the evaluation of polynomials at a given point x we use Estrin's [19] variation of Horner's method [20]. This method allows for vector parallelization and proves to be very efficient. Its performance is comparable to polynomial addition. In particular, for a given special function the time for an evaluation of the Taylor polynomial can be faster than the evaluation of the function itself.

3.3.3 Product formulas

The first non-trivial Taylor operator to implement is the product $P(t) \cdot Q(t)$ of two polynomials. The generic formula for such a product is similar to formula (1), but with the difference that all the terms are independent:

$$c_k = \sum_{i+j=k} a_i \cdot b_j = \sum_{i=0}^{k} a_i \cdot b_{k-i} \tag{2}$$

As already discussed, a direct implementation of such a formula for $N = 31$ needs to perform at least 930 floating point operations. But we cannot expect all the vectors and partial results to fit in CPU registers during the whole computation, so values generally must be stored to and reloaded from memory. Thereby a direct implementation of these formulas without vector unit and without indirect addressing will normally account from 2 to 3 times the number of CPU cycles. So typically, a HOAD operator will use several thousand CPU cycles more than the scalar operation that it replaces.

A challenge to implement such formulas efficiently when using a vector unit stems from the fact that the vector of the second term in the sum (b in (2)) is accessed in reverse order. Such downward accesses have a performance hit on vector units. We can avoid these by reversing the vector b first, then using a sequence of dot products to perform the summation. With the performance measures that we have seen above this would amount to lower bound of a cycle count for `float` of

$$T_{\text{div}} + T_{revert} + \frac{N}{2} T_{dot} = 70 + 20 + 16 \cdot 65 = 1130.$$

We can do much better when implementing (1) differently. In fact, as soon as a particular coefficient b_i is computed, its scalar multiple with the vector a' can be computed and added to the partial sums of the coefficients $b_{i+1}, \dots, b_N$. When using scalar-multiply-add-in-place operations (5th line in Table 1) this leads to a lower bound on the instruction count of about $70 + 16 \cdot 46 = 806$.

As long as there are no dependencies between the coefficients, the *convolution* formulas (1) and (2) also have algorithms that run in $O(N \log N)$. Unfortunately the constants of proportionality for these algorithms can be large and we have to chose one that performs well for small N. We use a variant of Karatsuba's algorithm, see [21], for $P(t) \cdot Q(t)$ and $P^2(t)$. This algorithm may or may not be more efficient than a quadratic implementation. But, as we can see in Table 2 it is more efficient for the important cases of `float` or `double` coefficients, so we use fast convolution for these and a quadratic convolution for the others.

Lines $P(t) \cdot Q(t)$ and $P^2(t)$ give the times for the operations as they are mapped to either fast or quadratic convolution. It shows that for these operations with `float` and `double` data, `arbogast`'s total cycle count, including memory access, is even below the total arithmetic operation count of 930 that we computed for equation (1). We see that the squaring algorithm then has also an improved running time for the `float` and `double` polynomials.

3.3.4 Equating coefficients

The next group of Taylor operators that `arbogast` provides are those that implement quadratic formulas similar to (1), *e.g* `div`, `cos`, `sqrt` and `exp`. Table 3 shows the measurements for some of the different basic operators that `arbogast` implements. Complex `float` and `double` operations are about 2 to 3 times more expensive than the real valued ones.

The tables also show the performance of `arbogast` when we use real or complex `long double`. This data type is less normalized between platforms. On our platform with a standard Intel processor, this data type is a

[4]The C standard also guarantees that an `fma` call also has better rounding properties than the composition of multiply and add operations.

Table 2: Taylor product operators, measured in CPU cycles, for $N = 31$.

	real	real	real	complex	complex	complex
	`float`	`double`	`long double`	`float`	`double`	`long double`
quadradic conv.	809.0	897.7	2585.2	2312.1	2818.3	11204.5
fast conv.	607.5	753.8	5053.2	2695.0	3996.9	17968.8
$P(t) \cdot Q(t)$	634.8	792.4	2605.6	2325.0	2900.5	11356.2
$P^2(t)$	516.1	627.8	2557.6	2316.2	2852.5	11245.5

Table 3: Quadratic Taylor operations of standard mathematical functions or operations, measured in CPU cycles, for $N = 31$. The functions of the first group are implemented directly, those of the second are implemented as compositions of other operators.

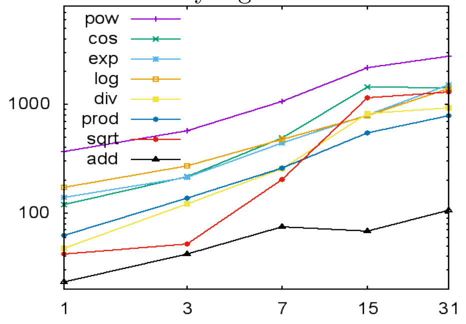
	real	real	real	complex	complex	complex
	`float`	`double`	`long double`	`float`	`double`	`long double`
`div`	974.2	922.8	2718.8	2416.8	3235.8	11711.9
`cos`	1176.0	1424.6	9516.4	4062.2	5065.0	20469.7
`sqrt`	1556.4	1300.2	1281.8	2064.4	2029.6	6748.3
`exp`	1260.3	1498.1	2816.3	2606.1	3393.4	11047.8
`log`	1345.0	1389.4	3248.3	2919.4	3885.0	13367.2
`pow`	2641.0	2748.2	6765.9	5666.4	7015.7	23713.6

Table 4: Quadratic Taylor operations of standard mathematical functions or operations, measured in `div` operations. The functions of the first group are implemented directly, those of the second are implemented as compositions of other operators.

	real	real	real	complex	complex	complex
	`float`	`double`	`long double`	`float`	`double`	`long double`
`div`	1.0	1.0	1.0	1.0	1.0	1.0
`cos`	1.2	1.5	3.5	1.7	1.6	1.7
`sqrt`	1.6	1.4	0.5	0.9	0.6	0.6
`exp`	1.3	1.6	1.0	1.1	1.0	0.9
`log`	1.4	1.5	1.2	1.2	1.2	1.1
`pow`	2.7	3.0	2.6	2.4	2.5	2.1

Table 5: Average number of cycles per operation for different degrees of Taylor polynomials with `double` coefficients, `arbogast`. The associated plot shows these numbers on a doubly logarithmic scale.

	1	3	7	15	31
`add`	23.4	42.3	75.4	68.3	106.1
`prod`	62.1	138.2	259.4	544.3	791.0
`div`	47.3	120.7	256.4	820.0	922.8
`cos`	119.0	215.8	493.5	1448.2	1424.6
`sqrt`	41.8	52.4	202.6	1146.8	1300.2
`exp`	138.9	214.1	444.0	795.9	1498.1
`log`	172.2	270.8	477.9	781.7	1389.4
`pow`	366.0	574.8	1056.1	2155.6	2748.2



good example for computations without a vector unit. Here, it uses a precision of 64 bits, 80 bits in total, but stores them in 16 bytes. It has no vector operations for the data type, only load and store instructions from or to memory, and the CPU only has 8 hardware registers that are organized as a stack. As we have to expect, besides for `sqrt`[5], the operators for `long double` are much slower than for `double`. Using this data type only makes sense when very high precision is mandatory and the extra computational cost can be afforded.

3.3.5 Combining operators

Table 3 also shows run-times for operators that are implemented as combinations of other operators. Listing 8 exemplifies implementations of $\ln(x)$ and x^ν operators. By convention, the `indef` operator, abbreviated as $\int$, sets the affine coefficient of the result to 0 and thus it composes well with `+` in the case of `log` and `*` for `pow`.

Listing 8: An example of implementing $\ln(x)$ and x^ν operators as combinations of other operators. Abbreviations `trd` (Taylor real double), ^{0} (the affine coefficient) and $\int$ (the indefinite integral) are imported from the `arbogast‾symbols` module.

```
inline trd log(trd x) {
  double y₀ = C::math::log(⁰(x));
  return (:y₀␣+␣∫(1/x):);
}
inline trd pow(trd x, double ν) {
    double y₀ = C::math::pow(⁰(x), ν);
    return (:y₀*exp(∫(ν/x)):);
}
```

Other operators are implemented with similar integral formulas, *e.g* `acosh` is implemented as

$$y_0 + \int(^2\sqrt{}(1/(^2(x) - 1))),$$

`tan` as

$$y_0 + \int(1/(1 + {}^2(x))),$$

where $^2\sqrt{}(x)$ and $^2(x)$ are abbreviations for the square root and square functions, respectively.

Table 4 shows that the implementation is as efficient as we may expect. The times for `log` are close to `div`, those of `pow` are close to the sum of the times for `div` and `exp`.

3.3.6 Other Taylor degrees

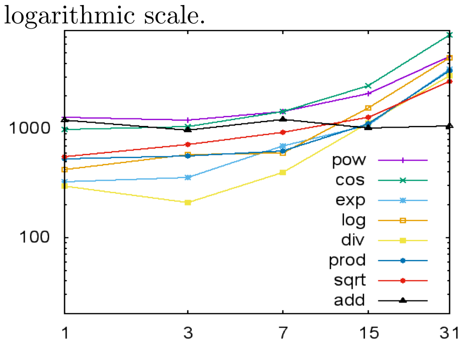
Per default `arbogast` maintains Taylor coefficients $0, \ldots, 31$. For projects that need less degree, `arbogast` can be compiled with a maximal degree of 1, 3, 7 or 15, instead.[6]

Thereby the run time can much be improved, see Table 5 and the associated figure. We can observe that the performance of these primitives of `arbogast` is quite regular. In particular, we see that the dependency of the length of the coefficient vector is even sublinear for most of them. For example, the behavior of `prod` is similar to $62N^{3/4}$. This is due to the fact that the SIMD features need a certain vector length to take effect.

[5] `sqrt` has a highly dependent formula that inhibits vectorization but which has the advantage of exploiting symmetries.
[6] To maintain the efficiency of the convolution algorithm, the number of coefficients must always be a power of 2.

Table 6: Average number of cycles per operation for different degrees of Taylor polynomials with `double` coefficients, ADOL-C. The first column "trace" shows the times for trace execution within ADOL-C. The other columns show the accumulated time for this trace execution plus the "forward mode" for the required degree. The associated plot shows these accumulated times on a doubly logarithmic scale.

	trace	1	3	7	15	31
`add`	928.4	1195.0	972.4	1226.5	1017.2	1065.8
`prod`	497.3	533.5	559.2	630.4	1120.4	3421.7
`div`	144.2	298.8	208.8	399.9	1122.1	3055.6
`cos`	748.9	981.3	1046.8	1430.5	2478.8	7292.7
`sqrt`	494.8	555.0	716.8	925.0	1282.0	2728.9
`exp`	274.0	323.6	355.6	694.7	1078.3	3529.1
`log`	264.9	420.9	583.9	598.8	1545.9	4493.3
`pow`	710.1	1273.4	1206.6	1439.6	2109.0	4649.0



3.4 A comparison to ADOL-C

We compare `arbogast` to ADOL-C because it is in relatively wide use and it is similar in several aspects that concern the programming interface. For both, the user has to identify the variables in the code that will be differentiated, and sections of the code in which the differentiation will be performed have to be determined.[7]

Nevertheless, there are many differences. ADOL-C is a package for forward and reverse differentiation. It is able to handle mixed derivatives in multiple directions. Other than the name suggests, ADOL-C is written in C++ (and not in C) and thereby it can serve C projects only in the intersection of the two languages. ADOL-C has no support for complex arithmetic or for complex special functions. It is important to note that these are not advantages or disadvantages per se, but depend a lot on the context in which they are applied and purpose the corresponding program is meant to serve.

A usage of ADOL-C for HOAD as we present proceeds in two steps.

1. In a "trace" phase the annotated code is executed. This phase computes the original floating point operations, and in addition keeps a trace of all operations for future use.

2. A second "forward" phase uses that trace and performs the veritable differentiation. This differentiation iteratively computes each coefficient of the resulting Taylor polynomial up to the desired degree.

Thus the computational method of ADOL-C is somewhat orthogonal to ours. On the one hand, `arbogast` follows the application flow of computation only once and computes all Taylor coefficients along with each operation. On the other hand, to the best of our understanding, ADOL-C follows this flow a first time to generate a dynamic trace of that flow. Then, during the forward phase this trace is followed dynamically, again, to compute the Taylor coefficients.

On a point of view of software engineering and maintenance the two approaches are also mostly orthogonal. Both require modification of an existing numerical code, in particular to mark variables that will be subject to differentiation. ADOL-C then requires important changes in the control flow of the application code to separate the *trace* and *forward* phase. Code that is modified in that way only serves as AD code, any connection with the original numerical code is lost. Thus, maintaining a numerical code together with its AD code may become tedious: a development or improvement of the numerical code imposes a *manual* transposal of the modifications into the AD code.

In contrast to that, `arbogast` imposes an annotation of all expressions that are to be differentiated, which can also be quite a burden for a pre-existing numerical application. Other than for ADOL-C, that modified code can then still be used as direct numerical code. All replacement of the `(:␣:)` expressions is done at compile time such that it has no impact whatsoever when a `(:␣:)` is interpreted as an expression composed of standard arithmetic operations. Thereby, once the transition to AD with `arbogast` is achieved, any modification of the numerical code is automatically transposed into AD.

Table 6 shows the average runtimes that we obtained for the same primitives as we have shown for `arbogast` in Table 5. The experimental setup is the same as described in Section 3.3.1, in particular we present average times of several thousand runs, and we applied the same careful investigations to ensure that no optimization phase (compiler or interpreter of the trace) can shortcut the measurement of operators or functions. ADOL-C' functions `trace_on` and `trace_off` are placed outside the inner loop of at least 1000 runs, so their overhead for initiating and terminating the trace can be neglected. For example the line for `cos` shows the average run time per function call of a sequence of 40 probes of 640000 chained applications `y = cos(y)`.

[7]Here we only discuss the part of ADOL-C that is able to perform HO AD.

Table 7: Parametric functions used as seeds, second order equations and validation formulas for some orthogonal polynomials.

		Seed [14, 22.8]			Second order ODE [15, 18.8.1]			Validation [15, 10.6.1]		
Name	φ_ν	a_ν	b_ν	g_ν	α_ν	β_ν	γ_ν	A_ν	B_ν	G_ν
Ultraspherical	$C_\nu^{(\lambda)}$	$1-z^2$	$-\nu z$	$\nu+2\lambda-1$	$1-z^2$	$-(2\lambda+1)z$	$\nu(\nu+2\lambda)$	1	$\frac{2(\nu+\lambda)}{\nu+1}z$	$\frac{\nu+2\lambda-1}{\nu+1}$
Chebyshev	T_ν	$1-z^2$	$-\nu z$	ν	$1-z^2$	$-z$	ν^2	1	$(2-\delta_{\nu,0})z$	1
Chebyshev	U_ν	$1-z^2$	$-\nu z$	$\nu+1$	$1-z^2$	$-3z$	$\nu(\nu+2)$	1	$2z$	1
Legendre	P_ν	$1-z^2$	$-\nu z$	ν	$1-z^2$	$-2z$	$\nu(\nu+1)$	1	$\frac{2\nu+1}{\nu+1}z$	$\frac{\nu}{\nu+1}$
Gen. Laguerre	$L_\nu^{(\lambda)}$	z	ν	$-(\nu+\lambda)$	z	$\lambda+1-z$	ν	1	$-\frac{z+2\nu+\lambda+1}{\nu+1}$	$\frac{\nu+\lambda}{\nu+1}$
Hermite	H_ν	1	0	2ν	1	$-2z$	2ν	1	$2z$	2ν
Hermite	He_ν	1	0	ν	1	$-z$	ν	1	z	ν

The first column shows the average time that these operations need for the trace, before any differentiation is effected. The other columns then show the overall average time for the whole operation in question, such that the numbers can directly be compared with those for `arbogast`.

The resulting times for ADOL-C have a very high variance and have to be taken with a lot of reserve. A thorough micro-benchmarking and review would have probably been convenient, but the C++ code of ADOL-C is so complex that we found it beyond our possibilities to investigate plausible causes for such an erratic behavior. Consequently, we have no explanation to offer why the addition operation in ADOL-C has such a high overhead for the trace phase, and we don't know if the running times for `exp` and `log` are significantly distinct.

Nevertheless after taking into account these uncertainties, we see that the trace phase (not the initiation or termination of the phase) has an important impact on the runtime of each of the operations and that it clearly forms a bottleneck of the computation[8]. We attribute this to the fact that tracing tracks the sequence of operations dynamically, resulting in a lot of dynamic allocations, indirections, cache misses, pipeline stalls and far jumps through the binary executable.

In our opinion, the lack of performance of ADOL-C is due to an unfortunate combination of two strategies that makes the code optimization by the compiler very difficult. First, the trace code uses a lot of indirections that are inherent to the operator overloading technique. They are quite challenging to any optimizing compiler, because of their side effects and their aliasing and because they interrupt the "natural" control flow of the application. But second, during the forward phase the sequence of operations is not directly visible to the compiler, but interpreted dynamically at run time from the trace. Thus, optimization opportunities that could profit from compile-time knowledge about a whole sequence of operations may be missed.

For the considered use case of HOAD, `arbogast`'s runtime is significantly better than ADOL-C in most of the cases. ADOL-C operates and performs similar to interpreted languages. Therefore a completely compiled approach in C as provided by `arbogast` may outperform it to the extent that we see in our measurements.

4 Operators for the HOAD of special functions

Special functions and their derivatives play a crucial role in research fields of physics and mathematical analysis. As reported in [14, 15], many of these functions are solutions of the general second order ordinary differential equation (ODE)

$$\alpha(z)\varphi^{(2)}(z) + \beta(z)\varphi^{(1)}(z) + \gamma(z)\varphi(z) = 0, \tag{3}$$

where the input z is either a real or a complex variable, and functions $\alpha(z)$, $\beta(z)$, $\gamma(z)$ determine the mathematical function $\varphi(z)$. In other words, seeds $\varphi^{(0)}(z) = \varphi(z)$ and $\varphi^{(1)}(z) = \frac{d\varphi}{dz}(z)$, and functions $\alpha(z)$, $\beta(z)$, $\gamma(z)$ may be used to evaluate the second order derivative $\varphi^{(2)}(z) = \frac{d^2\varphi}{dz^2}(z)$, then higher-order derivatives. For the sake of generality in the presentation, we also adopt the same general seed formula and notation for the orthogonal polynomials, Tab. 7, the Bessel functions, Tab. 8, and the hypergeometric functions, Tab. 9. Note that other formulas are available and can be used to deal with special cases, see for instance [22].

In the past, considerable research efforts have been directed at implementing special functions in numerical libraries (some of them are proposed in the GNU Scientific Library), without, however, having developed genuine activities within the context of AD. This paper builds on [10, 23, 24] to propose formulas and operators that allow for the automatic generation of the higher-order automatic differentiation library `arbogast` for mathematical functions satisfying (3), then for its validation.

[8]ADOL-C has a special mode for first order differentiation that does not need the trace phase.

Table 8: Parametric functions used as seeds, second order equations and validation formulas for representative Bessel functions.

		Seed				Second order ODE				Validation			
Bessel	φ_ν	a_ν	b_ν	g_ν	Ref.	α_ν	β_ν	γ_ν	Ref.	A_ν	B_ν	G_ν	Ref.
Bessel	J_ν	1	$-\frac{\nu}{z}$	1	(10.6.1)	z^2	z	$(z^2-\nu^2)$	(10.2.1)	1	$\frac{2\nu}{z}$	1	(10.6)
Modified	I_ν	1	$-\frac{\nu}{z}$	1	(10.29.1)	z^2	z	$-(z^2+\nu^2)$	(10.25.1)	1	$-\frac{2\nu}{z}$	-1	(10.29)
Spherical	j_ν	1	$-\frac{\nu+1}{z}$	1	(10.51.1)	z^2	$2z$	$z^2-\nu(\nu+1)$	(10.47.1)	1	$\frac{2\nu+1}{z}$	1	(10.51)

Table 9: Parametric functions used as second order equations and validation formulas for some hypergeometric functions.

		Second order ODE				Validation			
Hyp. Geo.	φ	$\alpha(z)$	$\beta(z)$	$\gamma(z)$	Ref.	$A(\nu)$	$B(z)$	$G(z)$	Ref.
Kummer	M	z	$\mu-z$	$-\nu$	(13.2.1)	ν	$2\nu-\mu+z$	$\mu-\nu$	(13.3.1)
Whitaker	U	z	$\mu-z$	$-\nu$	(13.2.1)	$\nu(\nu-\mu+1)$	$\mu-2\nu-z$	1	(13.3.7)
Gauss	${}_2F_1$	$z-z^2$	$\xi-(\nu+\mu+1)z$	$-\nu\mu$	(15.10.1)	$\nu(z-1)$	$\xi-2\nu-(\mu-\nu)z$	$\xi-\nu$	(15.5.11)

4.1 Second order ODE

The key difference between [14] and AD is in the definition of z. AD considers z as a function $z(t)$ depending on some variable t and implements it as a Taylor polynomial, the coefficients of which are classically denoted by z_k, $k=0,...,N$, and satisfy $z_k = \frac{1}{k!}\frac{\partial^k z}{\partial t^k}$. The second order differentiation of the compound function $v(t) = \varphi \circ z(t)$ [25] is here carried out by applying the chain rule to (3). This yields the general formulation (4),

$$\begin{aligned} v^{(0)} &= \varphi^{(0)}\big(z^{(0)}\big), \quad v^{(1)} = \varphi^{(1)}\big(z^{(0)}\big)z^{(1)}, \\ v^{(2)} &= \frac{-\gamma v^{(0)}\big(z^{(1)}\big)^3 - \beta v^{(1)}\big(z^{(1)}\big)^2 + \alpha v^{(1)} z^{(2)}}{\alpha z^{(1)}}, \text{ for } z^{(1)} \neq 0. \end{aligned} \tag{4}$$

This can be overloaded for the higher-order differentiation of $v(t) = \varphi \circ z(t)$ [25]. Equating coefficients leads to an implementation of $v(t)$ which is of quadratic worst-case complexity [24], $O(N^2)$, where N is the maximal degree of a Taylor polynomial. The special case of $z^{(1)} = 0$ is discussed in [25].

4.1.1 Seeds

Equation (4) requires the evaluation of the function φ and its derivative at the point of development, that can then be used as *seeds* for equating coefficients. Whereas an implementation of φ is usually available, an implementation of its derivative might not. But many parameterized families of functions provide recurrence relations that relate functions and their derivatives. With `arbogast` we are able to use these recurrence relations to actually compute the derivative where necessary.

The families of orthogonal polynomials, Bessel functions, and hypergeometric functions are constituted into groups of parameterized functions with specific names and very similar relationships. To avoid any confusion with the differentiation order k, we use the same leading index ν to denote either the degree in an orthogonal polynomial sequence, or the "main" parameter in the other functions. The interested reader is referred to [14, 15] for parameter ranges.

Under this convention, these three families of functions meet the general first order differential relation,

$$a_\nu(z)\varphi_\nu^{(1)}(z) = b_\nu(z)\varphi_\nu(z) + g_\nu(z)\varphi_{\nu-1}(z), \tag{5}$$

that defines the seed $\varphi_\nu^{(1)}$ by means of functions $a_\nu(z)$, $b_\nu(z)$ and $g_\nu(z)$.

For instance, the first derivative of the hypergeometric function $\varphi(\nu,\mu;\xi;z)$, the parameters of which are here denoted by ν, μ, and possibly ξ, satisfies

$$a_{[\nu,\mu;\xi]}(z)\varphi^{(1)}(\nu,\mu;\xi;z) = b_{[\nu,\mu;\xi]}(z)\varphi(\nu,\mu;\xi;z) + g_{[\nu,\mu;\xi]}(z)\varphi(\nu-1,\mu;\xi;z). \tag{6}$$

Functions $a_{[\nu,\mu;\xi]}(z)$, $b_{[\nu,\mu;\xi]}(z)$ and $g_{[\nu,\mu;\xi]}(z)$ related to the confluent hypergeometric functions $M(\nu,\mu;z)$ and $U(\nu,\mu;z)$, and the hypergeometric function ${}_2F_1(\nu,\mu;\xi;z)$ are reported in Tab. 10. Additionally, these families of functions have recurrence relations for all two (respectively three) parameters. This leads to formulas that only refer to $\varphi(\nu+1,\mu+1;\xi+1;z)$ multiplied by a scalar $d_{[\nu,\mu;\xi]}$

$$\varphi^{(1)}(\nu,\mu;\xi;z) = d_{[\nu,\mu;\xi]}\varphi(\nu+1,\mu+1;\xi+1;z). \tag{7}$$

Since it also avoids a polynomial division by the $a_{[\nu,\mu;\xi]}$ term, `arbogast` uses Equation (7) to implement $\varphi^{(1)}$ for the hypergeometric functions.

Table 10: Possible implementation for the seeds of some hypergeometric functions.

		Seed (6)				Seed (7)	
Hypergeom.	φ	$a_{[\nu,\mu;\xi]}$	$b_{[\nu,\mu;\xi]}$	$g_{[\nu,\mu;\xi]}$	Ref.	$d_{[\nu,\mu;\xi]}$	Ref.
Kummer	M	z	$\nu-\mu+z$	$\mu-\nu$	[14, (13.4.11)]	$-\frac{\nu}{\mu}$	(13.3.15)
Whitaker	U	z	$\nu-\mu+z$	1	[14, (13.4.26)]	$-\nu$	(13.3.22)
Gauss	${}_2F_1$	$z(1-z)$	$\nu-\xi+\mu z$	$\xi-\nu$	(15.5.19)	$\frac{\nu\mu}{\xi}$	(15.5.1)

4.1.2 Validation

The recurrence relation with respect to the index ν,

$$A_\nu(z)\varphi_{\nu+1}(z) = B_\nu(z)\varphi_\nu(z) - G_\nu\varphi_{\nu-1}(z), \tag{8}$$

is used for the validation of the HOAD of the orthogonal polynomials. Functions $A_\nu(z)$, $B_\nu(z)$ and $G_\nu(z)$ depending on ν are defined in Tab. 7. This formula also allows for the validation of the HOAD of the Bessel and hypergeometric functions. Corresponding $A_\nu(z)$, $B_\nu(z)$ and $G_\nu(z)$ are reported in Tabs. 8 and 9. Note that, for the ${}_2F_1$ function, any formula based on two of the contiguous functions $\varphi(\nu\pm1,\mu;\xi;z)$, $\varphi(\nu,\mu\pm1;\xi;z)$ or $\varphi(\nu,\mu;\xi\pm1;z)$ of $\varphi(\nu,\mu;\xi;z)$ can be considered [14].

For all special functions that have such validation formulas, `arbogast` implements automatic unit tests that ensure the correctness of the functions. For the implementation of these tests we can use the same techniques as for the implementation of the HOAD operators, so we will omit further details in the sequel.

4.1.3 Faddeeva function

With regards to the application reported in Section 5, we also consider the so-called Faddeeva function $w(z)$. This scaled complex complementary error function used in many fields of physics [24],

$$w(z) = e^{-z^2}\left(1+\frac{2i}{\sqrt{\pi}}\int_0^z e^{t^2}dt\right) = e^{-z^2}\operatorname{erfc}(-iz). \tag{9}$$

satisfies the recurrence formula [15, (7.10.3)]

$$w^{(k+2)}(z) + 2zw^{(k+1)}(z) + 2(k+1)w^{(k)}(z) = 0. \tag{10}$$

At $k=0$, it provides a second order ODE that agrees with (3) for $\alpha(z)=1$, $\beta(z)=2z$ and $\gamma(z)=2$. Moreover, this equation may be used for validation purposes for $k>2$. Taylor coefficients are computed from the seed [15, (7.10.2)]:

$$w^{(1)}(z) = -2zw(z) + (2i/\sqrt{\pi}). \tag{11}$$

4.2 Implementation

An excerpt of code for the second order ODE method (4) can be seen in Listing 9. This `snippet` code is then used in Listing 10 to provide the differential operator (DO) $w\circ z$ for the Faddeeva function, where z is a real or complex function that is represented by its Taylor polynomial at point t_0. Another `snippet` (not shown) is used for families of functions that are themselves parameterized, such as $J_\nu\circ z$, $I_\nu\circ z$, $T_\nu\circ z$ and ${}_2F_1^{\nu,\mu;\xi}\circ z$.

We see that this `snippet` has 11 slots corresponding to identifiers that have to be specialized by each user of the snippet. The first five correspond to type parameters, *complete types* in the C jargon.

The next five are the functions that will give the specifics of the DO that is to be implemented. In fact, the constraints on the right hand side of the `=` sign specify that φ^0 and φ^1 must not necessarily be proper functions, they only have to allow for the evaluation of `b=`φ^0`(a)` at compile time. So, φ^0 and φ^1 can *e.g* be functions or type generic macros.

The last slot, $\mathbf{\Phi}$, names the function that this snippet will produce, namely an `extern` function that receives and returns a value of the Taylor polynomial type `TP`.

The bottom half of Listing 9 shows the initial part of the C code of the DO itself. The first five variables (`z`^{1}, `z`^{2}, `r`^{0}, `r`^{1} and `v`^{0}) represent the Taylor polynomial of the first and second derivative $z^{(1)}$ and $z^{(2)}$, the evaluation of φ^0 and φ^1 at $z(t_0)$, and the initialization of the return value for the first two Taylor coefficients.

Then, `A`, `B` and $\mathbf{\Gamma}$ are the result of the compositions of α, β and γ with `z(t)` calculated at point `t`$_0$ of the composition $\alpha\circ z(t)$.

Listing 9: The solver for the second order ODE in (3), code excerpt. First, the *Modular C* specific directives that describe the parameterization of the code. Below, the generic implementation of the differential operator.

```
#pragma CMOD module arbogast::snippet::ODE2nd
#pragma CMOD import arbogast::symbols

#pragma CMOD snippet none
/* The Taylor and floating types used in the snippet. */
#pragma CMOD slot TP = complete
#pragma CMOD slot Tα = complete
#pragma CMOD slot Tβ = complete
#pragma CMOD slot Tγ = complete
#pragma CMOD slot Tf = complete
/* The five functional parameters */
#pragma CMOD slot φ^0 = { Tf a, b; b = φ^0(a); }
#pragma CMOD slot φ^1 = { Tf a, b; b = φ^1(a); }
#pragma CMOD slot α  = none
#pragma CMOD slot β  = none
#pragma CMOD slot γ  = none
/* The name of the resulting function */
#pragma CMOD slot Φ  = extern TP Φ(TP);

/* Generic code of DO Φ */
TP Φ(TP z) {
  // Initialization
  TP z^1 = deriv(z);
  TP z^2 = deriv(z^1);

  // Mathematical function φ
  Tf r^0 = φ^0(z.coeff[0]);
  Tf r^1 = φ^1(z.coeff[0]);
  TP v^0 = INIT−MAX([0] = r^0,
                    [1] = r^1*z.coeff[1]);
  Tα const A    = arbogast::operate(α, z);
  Tβ const B    = arbogast::operate(β, z);
  Tγ const Γ    = arbogast::operate(γ, z);

  // Auxiliary variables
  TP const γα     = (:-(Γ/A)*(z^1*z^1):);
  TP const βα     = (:z^2/z^1-(B/A)*z^1:);

  /* resolve by equating coefficients */
  ...
  return v^0;
}
```

The next two auxiliary variables $\gamma\alpha$ and $\beta\alpha$ correspond to an algebraic reformulation of (4). As described in Section 3.2 we use a specialized "context" for AD operations that is marked with special bracketing[9]. This construct then just rewrites an expression with operators `*`, `/`, `+` ... into functional notation. Here, in the special case of the DO, the reformulation is done such that the problem parameters `A`, `B` and $\mathbf{\Gamma}$ only occur in the divisions `(:B/A:)` and `(:Γ/A:)`. Thereby, the system can take advantage of specific properties of these arguments and can avoid the division of polynomials, if possible. Special cases are detected at compile time when any of these is a constant function, or if `B` is even the 0-function.

Listing 10 shows the user side for the snippet, here in particular the implementation of the Faddeeva function, `arbogast::Faddeeva::func−cd`. The suffix `−cd` stands for complex double precision argument functions.

The top half shows the few C code that we have to provide for this implementation. The very first macro given at the beginning of Listing 10 provides a *type generic* interface: `_Generic` is C11's new keyword for type based choices. Here it chooses either `func−cf` or `func−cd` according to `Z`'s type. This function is then applied to the same argument `Z`.

To specify the functions, first we use an implementation of the Faddeeva function from the `libcerf` library, and look up the derivative of the `wofz` function. Then, we observe that "functions" α and γ are actually constant and can be implemented as simple macros. Function β, specialized as β`−cd`, is just $2 \cdot z(t)$.

These specializations are then fed to the `import` of the snippet code, on the right hand side. This import is identified through a name (`wofz−cd`), an additional import of the same snippet with another name, `wofz−cf`, to implement `func−cf` is omitted.

[9]As for the :: character, the syntax for this is choosable, here we use `(:` and `:)` for starting and ending an AD expression, respectively, see Section 3.2.

Listing 10: The Faddeeva DO for `_Complex double` (cd) implemented through the ODE in (3), code excerpt. First, the implementation of the interfaces that are needed for the use of the `snippet` of Listing 9. Below, the *Modular C* specific directives that fill the slots of that `snippet`.

```
#define Faddeeva(Z) _Generic((Z), types::tcf: func-cf, default: func-cd)(Z)

#define wofz support::cerf::w_of_z
inline scd φ¹-cd(scd z₀) {
  scd r⁰ = wofz(z₀);
  return -2*z₀*r⁰ + 2*I/√π;
}
/* Two of the "functions" are actually just constants. */
#define α 1
#define γ 2
inline tcd β-cd(tcd z) {
  return (:2*z:);
}

#pragma CMOD import wofz-cd = arbogast::snippet::ODE2nd
/* The Taylor and floating types used in the snippet.  */
#pragma CMOD fill   wofz-cd::TP = tcd
#pragma CMOD fill   wofz-cd::Tα = scd
#pragma CMOD fill   wofz-cd::Tβ = tcd
#pragma CMOD fill   wofz-cd::Tγ = scd
#pragma CMOD fill   wofz-cd::Tf = scd

/* The five functional parameters:                     */
#pragma CMOD fill   wofz-cd::φ⁰ = wofz
#pragma CMOD fill   wofz-cd::φ¹ = φ¹-cd
#pragma CMOD fill   wofz-cd::α  = α
#pragma CMOD fill   wofz-cd::β  = β-cd
#pragma CMOD fill   wofz-cd::γ  = γ
/* The name of the resulting function:                 */
#pragma CMOD fill   wofz-cd::func = func-cd
```

The five types of the import are chosen to be scalar complex double (`scd`) for `T`α, `T`β and `Tf` and as Taylor polynomial for the two others, `TP` and `T`γ (tcd). Observe, that thereby in the instantiation of the snippet of Listing 9 can be optimized substantially at compile time: the division `(:B/A:)` is just a division of two scalars and `(:`$\mathbf{\Gamma}$`/A:)` divides Taylor polynomial $\mathbf{\Gamma}$ by a scalar.

We can easily estimate the complexity of the whole. The only parts with quadratic complexity are the products and division for the computation of $\gamma\alpha$ and $\beta\alpha$ and the part for equating coefficients which is not shown.

With Modular C's `foreach` directive, we can instantiate versions for other types. We just have to surround the above code by `#pragma CMOD foreach TYPE = cf cd` and `#pragma CMOD done` and do some adjustments to the naming. Then the code is repeated twice. In the first copy, all occurrences of the pattern `${TYPE}` will be replaced by `cf`, in the second by `cd`.

5 A case study in optics

In light propagation, a plausible sequence for the computation of the so-called *propagation constant* β in an optic device is

$$\lambda \to \omega = c/\lambda \xrightarrow{model} \begin{matrix}\varepsilon(\omega)\\ \text{or}\\ \tilde{n}(\omega)\end{matrix} \xrightarrow{PDE} \beta, \tag{12}$$

for a given wavelength λ or frequency $\omega = c/\lambda$ (c is the speed of light). The complex dielectric function $\varepsilon(\omega)$ and the refractive index $\tilde{n}(\omega)$ are interrelated complex nonlinear functions, the parameters of which are identified from experimental data to reproduce the behavior of a given medium in a range of frequencies. A nonlinear eigenvalue problem (PDE) may be then set according to the geometry and the materials to compute β, see [26, 27] for instance. The interested reader is referred to [28] for a higher-order solver for nonlinear complex eigenvalue problem and to [29, 30] for exhaustive illustrations in beam physics.

The full account for the dependence on wavelength/frequency is crucial for the evaluated properties such as the modal delay per unit length, the dispersion and the dispersion slope. These are mixed derivatives, up to the order three, of the propagation constant with respect to the wavelength and the frequency. More generally [31], the nonlinear effects of light dispersion may be studied using a Taylor expansion for the modeling of the propagation constant. Higher-order methods ranging from analytical differentiation [32] to finite difference schemes up to order six [33] were proposed for the evaluation of derivatives of the propagation constant, with applications to optimal design [34].

In the following, we present three classical nonlinear models of growing complexity, to propose an AD solution to these higher-order derivative computations and to make test with `arbogast` on complex variables and the Faddeeva function used in many fields of physics [24].

5.1 Refractive index and complex dielectric functions

The complex nonlinear index $\tilde{n}(\omega) = n(\omega) - i\kappa(\omega)$ is a parameterized function that depends on the frequency ω (or the wavelength) and the material. It combines the real refractive index denoted by $n(\omega)$ (or $n(\lambda)$) and possibly a pure imaginary absorption function denoted by $i\kappa(\omega)$. Real refractive indexes $n(\lambda)$ are in use for glass optical fibers while complex dielectric functions are preferred in the modeling of fiber lasers or fiber optic sensors that comprise metals or rare earth elements. These interrelated functions may be deduced from each other following

$$\begin{aligned} n &= \tfrac{1}{2}\left[(Re(\varepsilon)^2 + Im(\varepsilon)^2)^{1/2} + Re(\varepsilon)\right]^{\frac{1}{2}}, \\ \kappa &= \tfrac{1}{2}\left[(Re(\varepsilon)^2 + Im(\varepsilon)^2)^{1/2} - Re(\varepsilon)\right]^{\frac{1}{2}}, \end{aligned} \tag{13}$$

equations in which the dependence in ω is omitted.

5.1.1 Sellmeier equation

The classical 3-term Sellmeier equation is presented for sake of completeness. The real-valued empirical formula models the dispersion of light in a transparent medium by linking the refractive index to the wavelength [35]

$$n^2(\lambda) = 1 + \sum_{j=1}^{3} A_j \frac{\lambda^2}{\lambda^2 - \lambda_j^2}, \quad \text{for } \lambda \in (\lambda_{min}, \lambda_{max}), \tag{14}$$

where the coefficients A_j and the resonant wavelengths λ_j ($j = 1,..3$) are identified from refractive index measurements in the range $(\lambda_{min}, \lambda_{max})$.

5.1.2 Lorentz-Drude model

The complex dielectric function is classically split into intraband and interband contributions. The intraband contribution $\varepsilon^f(\omega)$ is computed by the Drude model [36]

$$\varepsilon^f(\omega) = 1 - \frac{f_0\omega_p^2}{\omega(\omega - i\Gamma_0)}. \tag{15}$$

It involves a unique oscillator with strength f_0, a damping coefficient Γ_0, and the plasma frequency ω_p. The interband contribution $\varepsilon^b(\omega)$ comprises a finite number J of oscillators

$$\varepsilon^b(\omega) = \sum_{j=1}^{J} \frac{f_j\omega_p^2}{(\omega_j^2 - \omega^2) + i\omega\Gamma_j}. \tag{16}$$

The frequency ω_j, the strength f_j, and the lifetime $1/\Gamma_j$ are parameters that describe the oscillators. Note that in the lossless case, $\Gamma_j = 0$ for $j = 0,..,J$, the Lorentz-Drude model is closely related to the Sellmeier equation (14). Parameter values for classical metals used in optics and optoelectronics devices are reported in [37].

5.1.3 Brendel-Bormann model

The interband contribution may be also modeled using a finite superposition,

$$\varepsilon^b(\omega) = \sum_{j=1}^{J} \chi_j(\omega), \tag{17}$$

of Brendel-Bormann oscillators $\chi_j(\omega)$, for $j = 1,..,J$ [37]. Each of them comprises an infinite number of oscillators modeled using the Faddeeva function w described in Subsection 4.1.3

$$\chi_j(\omega) = \frac{i\sqrt{\pi} f_j\omega_p^2}{2\sqrt{2}\sqrt{\omega^2 - i\omega\Gamma_j}\sigma_j}\chi_j(\omega)\left(w\left(\frac{\sqrt{\omega^2 - i\omega\Gamma_j} - \omega_j}{\sqrt{2}\sigma_j}\right) + w\left(\frac{\sqrt{\omega^2 - i\omega\Gamma_j} + \omega_j}{\sqrt{2}\sigma_j}\right)\right), \tag{18}$$

The parameters $\{\sigma_j\}_{j=1,..,J}$ allow for the definition of models ranging from a purely Lorentzian model for $\{\sigma_j\}_{j=1,..,J} \simeq 0$, to a nearly Gaussian model for $\{\Gamma_j\}_{j=1,..,J} \simeq 0$. Parameters values for the Brendel-Bormann model are also reported in [37].

Table 11: Run time of implementations of the models for optics with `arbogast`. Floating point types are `complex double`.

	degree	1		3		7		15		31	
function	model	cycles	`*div`	cycles	`*div`	cycles	`*div`	cycles	`*div`	cycles	`*div`
	Drude	375	7.9	682	5.6	1211	4.7	2144	1.5	5430	5.9
ε	Lorentz-Drude	1725	36.5	2802	23.2	14208	55.4	18341	13.1	23661	25.7
	Brendel-Bormann	38732	818.7	61691	511.0	84843	330.5	112751	80.7	240944	261.5
ñ		355	0.6	601	2.7	1368	2.9	2656	1.1	6421	2.2

Table 12: Run time of a real-valued implementation of the Lorentz-Drude model for optics with ADOL-C. All floating point types are `double`.

	degree	1	3	7	15	31
ε	Lorentz-Drude	13277	15275	19066	24168	53390

5.2 Implementation

An AD implementation of the three models (Drude, Lorentz-Drude and Brendel-Bormann) has been undertaken with `arbogast`. The runtimes of the different functions for different Taylor degrees are shown in Table 11.

The implementation of Brendel-Bormann is not possible as easily with other AD tools such as ADOL-C, because they do not provide an operator for the Faddeeva function. We thus implemented a real-valued version of the Lorentz-Drude algorithm in ADOL-C. As in Section 3.4, we see that `arbogast` clearly outperforms ADOL-C for the considered use case of "forward" HOAD, see Table 12.

6 Conclusion

We have presented `arbogast`, a toolbox for HOAD based on Modular C. It provides a comfortable interface to differentiate existing C programs and to implement new differential operators from scratch. It provides full support for major features of C that target the efficient implementation of numerical methods, such as the use of all real and complex floating point types, type generic interfaces and variable length arrays as function parameters. Up to our knowledge, `arbogast` is unique with these features, no other HOAD tool offers such a complete integration within C.

We have validated this tool by several means. First, a detailed campaign of micro-benchmarks shows that our implementation is very efficient and that we are able to use the vector processing features that are present on modern commodity hardware to a large extent. Second, we were able to implement operators for special functions that obey a certain type of second order differential equations. Again, up to our knowledge, `arbogast` is the only tool that implements such DO for real and complex valued functions systematically and efficiently. Third, we have instantiated the DO for one of these functions, the Faddeeva w function needed for the computation of the complex dielectric functions used for the modeling of fiber lasers or fiber optic sensors that comprise metals or rare earth elements.

References

[1] Louis François Antoine Arbogast. *Du calcul des dérivations.* Imprimerie de Levrault frères, Strasbourg, An VIII (1800).

[2] Jean-Pierre Friedelmeyer. Arbogast, de l'Institut national de France, June 2016.

[3] Isabelle Charpentier, Jean-Pierre Friedelmeyer, and Jens Gustedt. Arbogast – Origine d'un outil de dérivation automatique. Research Report RR-8911, INRIA, May 2016.

[4] Warren P. Johnson. The curious history of fa di brunos formula. *Amer. Math. Monthly*, 109:217–234, 2002.

[5] Alex D. D. Craik. Prehistory of Faà Di Bruno's Formula. *Amer. Math. Monthly*, 112:119–130, 2005.

[6] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Society for Industrial Mathematics, 2nd edition, November 2008.

[7] Bruce Christianson. Reverse accumulation and attractive fixed points. *Optim. Methods Softw.*, 3:311–326, 1994.

[8] I. Charpentier. On higher-order differentiation in nonlinear mechanics. *Optim. Method. Softw.*, 27(2):221–232, 2012.

[9] I. Charpentier and B. Cochelin. Towards a full higher-order AD continuation and bifurcation framework. *Optim. Method. Softw. (submitted)*, 2017.

[10] Isabelle Charpentier and Jean Utke. Fast higher-order derivative tensors with Rapsodia. *Optim. Method. Softw.*, 24:1–14, 2009.

[11] Tiobe Software BV, 2015. monthly since 2000.

[12] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[13] Jason Abate, Christian Bischof, Lucas Roh, and Alan Carle. Algorithms and design for a second-order automatic differentiation module. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, ISSAC '97, pages 149–155, New York, NY, USA, 1997. ACM.

[14] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover Publications, New York, 1964.

[15] F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, and B. V. Saunders, editors. *NIST Digital Library of Mathematical Functions.* 1.0.13 edition, September 2016.

[16] JTC1/SC22/WG14, editor. *Programming languages - C.* Number ISO/IEC 9899. ISO, cor. 1:2012 edition, 2011.

[17] Jens Gustedt. Modular C. Research Report RR-8751, INRIA, June 2015.

[18] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate taylor series. *Mathematics of Computation*, 69(231):1117–1130, 2000.

[19] Gerald Estrin. Organization of computer systems – the fixed plus variable structure computer. In *Proc. Western Joint Comput. Conf.*, pages 33–40, May 1960.

[20] William George Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions. Royal Society of London*, pages 308–335, July 1819.

[21] Anatoli A. Karatsuba and Yuri P. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys. Dokl.*, 7:595596, 1963.

[22] Richard Culham. ME755 special functions. Engineering Courses online, Univ. Waterloo, 2004.

[23] Isabelle Charpentier and Claude Dal Cappello. Higher-order automatic differentiation of mathematical functions. *Comput. Phys. Commun.*, 189:66–71, 2015.

[24] Isabelle Charpentier. Optimized higher-order automatic differentiation for the Faddeeva function. *Comput. Phys. Commun.*, 2016. to appear.

[25] Isabelle Charpentier, Claude Dal Cappello, and Jean Utke. Efficient higher-order derivatives of the hypergeometric function. In Christian H. Bischof et al., editors, *Advances in Automatic Differentiation*, pages 127–137. Springer, 2008.

[26] T. A. Lenahan. Calculation of modes in an optical fiber using the finite element method and EISPACK. *The Bell System Technical Journal*, 62(9):2663–2694, November 1983.

[27] Linda Kaufman. Eigenvalue problems in fiber optic design. *SIAM J. Matrix Anal. A.*, 28:105–117, 2006.

[28] M. Bilasse, I. Charpentier, E. Daya, and Y. Koutsawa. A generic approach for the solution of nonlinear residual equations. Part II: Homotopy and complex nonlinear eigenvalue method. *Comput. Method. Appl. M.*, 198:3999–4004, 2009.

[29] Martin Berz. *Modern Map Methods in Particle Beam Physics.* Academic Press, 1999.

[30] Kyoko Makino and Martin Berz. COSY INFINITY version 9. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 558(1):346–350, 3 2006.

[31] G. Agrawal. *Nonlinear Fiber Optics.* Academic Press, SanDiego, fifth edition, 2012.

[32] Linda Kaufman. Calculating dispersion derivatives in fiber-optic design. *J. Lightwave Technol.*, 25:811–819, 2007.

[33] Jr. Mores, J.A., G.N. Malheiros-Silveira, H.L. Fragnito, and H.E. Hernández-Figueroa. Efficient calculation of higher-order optical waveguide dispersion. *Opt. Express*, 18:19522–19531, 2010.

[34] Linda Kaufman, Seok-Min Bang, Brian Heacook, William Landon, Daniel Savacool, and Nick Woronekin. Designing optical fibers: Fitting the derivatives of a nonlinear PDE-eigenvalue problem. *American Journal of Computational Mathematics*, 02(04):321–330, 2012.

[35] Wilhelm Sellmeier. Zur Erklärung der abnormen Farbenfolge im Spectrum einiger Substanzen. *Ann. Phys. (Berlin)*, 219:272–282, 1871.

[36] H. Ehrenreich and H. R. Philipp. Optical properties of Ag and Cu. *Phys. Rev.*, 128:1622–1629, Nov 1962.

[37] A.D. Rakić, A.B. Djurišic, J.M. Elazar, and M.L. Majewski. Optical properties of metallic films for vertical-cavity optoelectronic devices. *Appl. Opt.*, 37:5271–5283, 1998.

[38] Harley Flanders. Automatic differentiation: Origin and references, 2002.

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399