Arbogast: Higher order AD 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

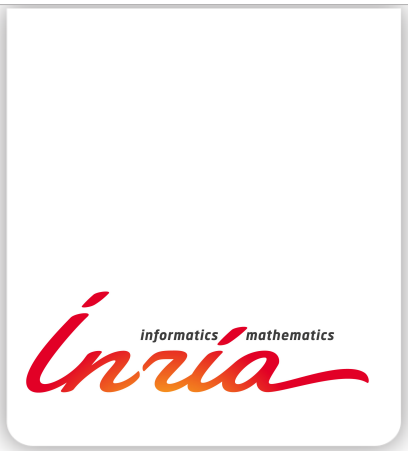
RESEARCH REPORT

N° 8907

April 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 — April 2016 — 7 pages

Abstract: We introduce a new tool for automatic differentiation, `arbogast`. It builds on a recent extension of the C programming language, *Modular C*, that eases code reuse and code unrolling. Although it is type generic (for the 6 different floating point types), code differentiated with *arbogast* contains no dynamic type dependencies and compiles into efficient executables. For a typical test case the re-implementation of some differential operator by means of `arbogast` has proven to be about 30% faster than a reference code in Fortran.

Key-words: automatic differentiation, Modular C, special functions

* ICube, CNRS and Université de Strasbourg, France
† Inria, France

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

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

Résumé : Nous introduisons le nouvel outil de différentiation automatique `arbogast`. Il s'appuie sur une récente extension du langage de programmation C, *Modular C*, qui facilite la réutilisation et le déploiement de code. Bien qu'il soit type-générique (pour les 6 différents types à virgule flottante), un code différentié avec *arbogast* ne contient aucune dépendance dynamique de type et sa compilation produit des exécutables efficaces. Pour un cas test particulier, le code avec ré-implantation de certains opérateurs différentiels au moyen d'`arbogast` est environ 30% plus rapide que le code Fortran de référence.

Mots-clés : différentiation automatique, Modular C, fonction spéciale

This toolbox for higher order AD is named after L. F. A. Arbogast (1759–1803), a French mathematician from Strasbourg (Alsace), for his pioneering work in derivation calculus [1, 2]. He is also responsible for the law introducing the metric system in the French Republic.

1 Introduction

Special functions and their derivatives play a crucial in research fields of physics and mathematical analysis. Considerable research efforts have been directed at implementing special functions in numerical libraries, some of them are proposed in the GNU Scientific Library, without having developed genuine activities within the context of AD.

Many special functions, Tab. 1, are solutions of the general second order ordinary differential equation (ODE)

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

where functions $\alpha(z)$, $\beta(z)$, $\gamma(z)$ determine the mathematical function $\varphi(z)$. In their classical definitions [3], the input z is either a real or a complex variable. Assuming $z = z(t)$ is a function depending on some variable t, this ODE yields the general formulation (2),

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

allowing for the higher-order differentiation of the compound function $v(t) = \varphi \circ z(t)$ [4], the implementation of which may be of quadratic complexity [5]. The case $z^{(1)} = 0$ is discussed in [4]. As usual, the derivative and the Taylor coeffficient of the function z at order k are denoted by $z^{(k)}$ and z_k, respectively.

One of the goals of this paper here is to show that equations (2) allow for the automatic generation of a higher-order automatic differentiation library comprising special functions satisfying (1). Only seeds $\varphi^{(0)} = \varphi$ and $\varphi^{(1)}$, and generating functions $\alpha(z)$, $\beta(z)$, $\gamma(z)$ have to be provided.

Function	Seeds		Generating functions		
	$\varphi\big(z^{(0)}\big)$	$\varphi^{(1)}(z^{(0)})$	$\alpha(z)$	$\beta(z)$	$\gamma(z)$
Faddeeva	$w\big(z^{(0)}\big)$	$-2z^{(0)}w\big(z^{(0)}\big) + 2i/\sqrt{\pi}$	1	$2z$	2
Bessel	$J_\nu\big(z^{(0)}\big)$	$-J_{\nu+1}\big(z^{(0)}\big) + \nu/z^{(0)}J_\nu\big(z^{(0)}\big)$	z^2	z	$(z^2-\nu^2)$
Modified Bessel	$I_\nu\big(z^{(0)}\big)$	$I_{\nu+1}\big(z^{(0)}\big) + \nu/z^{(0)}I_\nu\big(z^{(0)}\big)$	z^2	z	$-(z^2+\nu^2)$
Hypergeometric	${}_2F_1\big(a,b;c;z^{(0)}\big)$	$(ab/c){}_2F_1(a+1,b+1;c+1;z^{(0)})$	$z(1-z)$	$c-(a+b+1)z$	$-ab$

Table 1: Seeds and generating functions for some mathematical functions satisfying (1), where a, b, c and ν are parameters.

2 Modular C as a tool for automatic code generation

Since decades, C is one of most widely used programming languages, see [6], and is used successfully for large software projects that are ubiquitous in modern computing devices of all scales. 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, **long double**, type generic mathematical functions (all in C99), programmable type generic interfaces, choosable alignment and unicode support (in C11), see [7].

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. Prominent among these shortcomings, is a lack of two closely related features: modularity and reusability. C misses to encapsulate different translation units (TU) properly: all symbols that are part of the interface of a software unit such as functions are shared between all TU that are linked together into an executable. A common practice to overcome that difficulty is the introduction of naming conventions. Usually software units (referred as *modules* in the following) are attributed a name prefix that is used for all data types and functions that constitute the programmable interface (API). Often such naming conventions (and more generally coding styles) are 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.

To cope with these difficulties we recently proposed an extension to the C standard called *Modular C*, see [8]. It consists in the addition of a handful of directives and a naming scheme transforming traditional TU

Listing 1: The solver for the second order ODE in (1), code excerpt. On the left the *Modular C* specific directives that describe the parameterization of the code. On the right 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 φ⁰ = { Tf a, b; b = φ⁰(a); }
#pragma CMOD slot φ¹ = { Tf a, b; b = φ¹(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 z¹ = deriv(z);
  TP z² = deriv(z¹);

  // Mathematical function φ
  Tf r⁰ = φ⁰(z.coeff[0]);
  Tf r¹ = φ¹(z.coeff[0]);
  TP v⁰ = INITIALIZER(v⁰, -1,
            [0] = r⁰, [1] = r¹*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)*(z¹*z¹):);
  TP const βα   = (:z²/z¹-(B/A)*z¹:);

  /* resolve by equating coefficients */
  ...
  return v⁰;
}
```

into *modules*. The change to the C language is minimal since we only add one feature, composed identifiers, to the core language[1]. Other features of Modular C are implemented through **CMOD** *directives*. For our discussion here the important among these are:

import Our modules can import other modules as long as the import relation remains acyclic and a module can refer to its own identifiers and those of the imported modules through freely chosen abbreviations. Other than traditional C **include**, our **import** directive ensures complete encapsulation between modules. The abbreviation scheme allows to seamlessly replace an imported module by another one with equivalent interface.

snippet In addition to the export of symbols, we provide parameterized code injection through the import of "*snippets*". This implements a mechanism that allows for code reuse, similar to so-called X macros or templates as other programming languages implement them. Listing 1 shows an excerpt of a **snippet** definition. The related **slot** directives are used to specify the parameters of a snippet. Then, **fill** directives are used to specialize these slots for a particular instantiation of the snippet, Listing 2.

foreach When implementing libraries that have to deal with similar code for different base types (here C's six real and complex floating point types) usually almost identical code is just replicated and then adapted for the target type. 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.

Additional features of our proposal are a simple 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.

3 The example of the 2nd order ODE

An excerpt of code for the second order ODE method (2) can be seen in Listing 1. This **snippet** code is then used in Listing 2 to provide the differential operator $w \circ f_{t_0}$ (DO) for the Faddeeva function, where f_{t_0} 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 $J_\nu \circ f_{t_0}$, $I_\nu \circ f_{t_0}$, $T_n \circ f_{t_0}$ and ${}_2F_1^{a,b;c} \circ f_{t_0}$.

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.

[1] In Listing 1 these correspond to the identifiers segmented by the character ■. Such a character may be chosen for each module.

Listing 2: The Faddeeva DO for _**Complex double** (cd) implemented through the ODE in (1), code excerpt. On the left the implementation of the interfaces that are needed for the use of the **snippet** of Listing 1. On right the *Modular C* specific directives that fill the slots of that **snippet**.

```
#define Faddeeva(Z)                 \   | #pragma CMOD import wofz-cd = arbogast■snippet■ODE2nd
  _Generic((Z),                     \   | /* The Taylor and floating types used */
           types■tcf: func-cf,      \   | /* in the snippet                      */
           default: func-cd)(Z)         | #pragma CMOD fill   wofz-cd■TP = tcd
                                        | #pragma CMOD fill   wofz-cd■Tα = scd
#define wofz support■cerf■w_of_z        | #pragma CMOD fill   wofz-cd■Tβ = tcd
inline scd φ¹-cd(scd z₀) {              | #pragma CMOD fill   wofz-cd■Tγ = scd
  scd r⁰ = wofz(z₀);                    | #pragma CMOD fill   wofz-cd■Tf = scd
  return -2*z₀*r⁰ + I*factor-scd;       |
}                                       | /* The five functional parameters     */
/* Two of the "functions" are  */       | #pragma CMOD fill   wofz-cd■φ⁰ = wofz
/* actually just constants     */       | #pragma CMOD fill   wofz-cd■φ¹ = φ¹-cd
#define α 1                             | #pragma CMOD fill   wofz-cd■α  = α
#define γ 2                             | #pragma CMOD fill   wofz-cd■β  = β-cd
inline tcd β-cd(tcd z) {                | #pragma CMOD fill   wofz-cd■γ  = γ
  return (:2*z:);                       | /* The name of the resulting function */
}                                       | #pragma CMOD fill   wofz-cd■func = func-cd
```

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, **Φ**, 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 right half of Listing 1 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 **Γ** are the result of the compositions of α, β and γ with `z(t)`. For instance, `A` is the Taylor polynomial in $\mathtt{t}_0$ of the composition $\alpha \circ z(t)$.

The next two auxiliary variables $\gamma\alpha$ and $\beta\alpha$ correspond to an algebraic reformulation of (2). As C does not allow to overload its operators, internally we use a "functional" notation for the operators that act on Taylor polynomials. The context of AD operations is therefore marked with special bracketing[2]. This construct then just rewrites an expression with operators `*`, `/`, `+` ... into the functional notation. Here in for the special case of the DO, the reformulation is done such that the problem parameters `A`, `B` and **Γ** 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 2 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 left hand side shows the few C code that we have to provide for this implementation. The very first macro that is given at the beginning of Listing 2 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 trough a name (`wofz-cd`), an additional import of the same snippet with another name, `wofz-cf`, to implement `func-cf` is omitted.

The five types of the import are chosen to be scalar complex double (`scd`) for `Tα`, `Tβ` and `Tf` and as Taylor polynomial for the two others, `TP` and `Tγ` (tcd). Observe, that thereby the division **div**`(B, A)` is just a division of two scalars and **div**`(`**Γ**`, A)` divides Taylor polynomial **Γ** 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.

[2]As for the ■ character, the syntax for this is choseable, here we use `(:` and `:)` for starting and ending an AD expression, respectively.

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`.

4 Properties of `arbogast`

- All DOs are implemented as functions (versus procedures or **void** functions) returning a Taylor polynomial. Thereby nested mathematical expressions in user code that are to be differentiated can be reused in their natural form without rewriting.
- The implementation of the core of the library and all derived functions are written in C. There is no need to program core parts in some language that is closer to the compilers, such as AST, see [9].
- C's natural assignment works flawlessly for our purpose. There is no need to overload the assignment operator.
- All code is readable and verifiable.
- Code can be re-used by means of the **snippet** directive:
  - The same code can be instantiated in three different versions. First, it can be a usual mathematical function (taking a **double**) or an AD function (taking a Taylor polynomial). For the latter, we can choose between the convergence criterion on the value (as for the **double** function), or a convergence criterion on the Taylor coefficients [10].
  - Internals of the library (*e.g.* the **div** DO) are implemented just once for all floating types.
  - User code can re-use predefined DOs such as our example `ODE2nd`.
- Interfaces can be programmed *type generic*. Several choices of floating point types can be maintained simultaneously.
- Interfaces can be programmed *case generic*, that is parameters with special properties (constants versus functions) are detected automatically and the generated code takes advantage of such a special case.
- Code is easily encapsulated through the **module** directive. Functions that are accessed through a local name (*e.g* `wofz`) can easily be replaced by another implementation.
- Each module can have an **entry** function that acts as its unit test for continuous integration of the code base.
- All DOs have at most quadratic complexity and are implemented efficiently. In particular, we are using fast convolution for the product formula, [11]. When the `arbogast` C code is compiled with FORTRAN's relaxed floating point model (gcc option `-fcx-fortran-rules`) the code is up to 30% faster than the FORTRAN reference code from which we started.

4.1 Conclusions and further work

Conclusions. `arbogast` provides a highlevel toolbox for the calculus with Taylor polynomials. It is entirely 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* or *reinterpretation* of code, because it permits the programmer to place his code in different types of contexts (usual math, AD, ...) to reinterpret it as usual C function or as a DO. Because of the type generic features of modern C, all specialization (e.g multiplication by a constant) can be delegated to the compiler.

Future work. C as base language does not provide the opportunity for operator overloading. Therefore, at the time of this submission, the use of `arbogast` on existing code needs to rewrite expressions as function calls. This rewriting completely remains at a syntactical level. Currently, there is no distinction for different types of contexts for rewriting, but we are evaluating the possibility of such a distinction. Also, we will consider this rewriting feature in the general context of *Modular C*, not only `arbogast`.

References

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

[2] Isabelle Charpentier, Jean-Pierre Friedelmeyer, and Jens Gustedt. `arbogast` — genèse d'un outil de différentiation automatique. in preparation, 2016.

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

[4] 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.

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

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

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

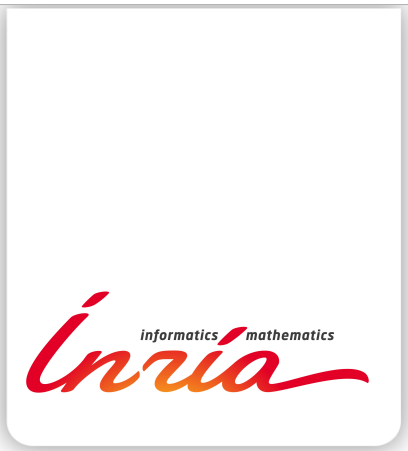
[8] Jens Gustedt. Modular C. Research Report RR-8751, INRIA, June 2015.

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

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

[11] Anatoli A. Karatsuba and Yuri P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics-Doklady*, 7:595596, 1963.

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



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

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

ISSN 0249-6399