https://doi.org/10.3311/PPee.16341

Periodica Polytechnica Electrical Engineering and Computer Science, 65(1), pp. 1–10, 2021

Higher Order Automatic Differentiation with Dual Numbers

László Szirmay-Kalos[1*]

[1] Department of Control Engineering and Information Technology, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, H-1521 Budapest, P. O. B. 91, Hungary
[*] Corresponding author, e-mail: szirmay@iit.bme.hu

Received: 28 April 2020, Accepted: 27 September 2020, Published online: 26 October 2020

Abstract

In engineering applications, we often need the derivatives of functions defined by a program. The approach chosen for derivative computation must be algebraic to allow computer implementation. A particular solution to obtain first derivatives is the application of dual numbers. This paper proposes simple and compact generalizations of this idea to obtain derivatives of arbitrary order for single or multi-variate functions and the automatic handling of 0/0 ambiguities in the calculations. We also provide the C++ code that takes advantage of operator overloading and recursion. The method is demonstrated by path animation, Gaussian curvature computation, and curve fairing.

Keywords

dual numbers, higher order automatic differentiation

1 Introduction

Many engineering tasks require the computation of derivatives of a function specified by a program segment. Although, the most essential approaches need only the first derivative, there are many problems requiring second or higher order derivation as well. For example, dynamics simulation uses the Newton's laws stating that the force is proportional to the second derivative of the path. The Frenet frame is based not only on the first but also on the second derivatives. Curvature calculation also needs the derivatives up to order two. Computational aesthetics, curve or surface fairing [1] use the assumption that a fair curve or surface uniformly distributes the curvature, which means that the third derivative of the parametric function should also be evaluated. Newton-Raphson methods attacking inverse problems use the Hessian i.e. the second derivative of the target function, and particular applications include reverse engineering [2, 3], regression methods [4], deep learning [5, 6] or medical imaging [7], etc. Second derivatives of parametric surfaces play an important role in non-photorealistic rendering as well to identify the principal curvature directions [8]. There are several options to implement derivative computation in a computer program:

- **Numerical differentiation** [9] approximates the derivatives of f at t by

$$f'(t) \approx \frac{f\left(t+\frac{\Delta}{2}\right)-f\left(t-\frac{\Delta}{2}\right)}{\Delta},$$

$$f''(t) \approx \frac{f(t+\Delta)-2f(t)+f(t-\Delta)}{\Delta^2},$$

$$f'''(t) \approx \frac{f\left(t+\frac{3\Delta}{2}\right)-3f\left(t+\frac{\Delta}{2}\right)}{\Delta^3} + \frac{3f\left(t-\frac{\Delta}{2}\right)-f\left(t-\frac{3\Delta}{2}\right)}{\Delta^3},$$

...,

  where Δ is a small offset. This is very simple, but suffers from severe problems. If Δ is too large, then the approximation is rather poor. However, when it is too small, the result is numerically unstable and is valid for very few digits (Fig. 1).
- **Symbolic differentiation** [10] mimics the manual differentiation process and generates the mathematical definition of the derived function. Symbolic differentiation handles the function as a whole, thus, it cannot solve conditionals and special cases.
- **Automatic differentiation** [5, 11, 12] determines the value and the derivative of a function defined by a

Cite this article as: Szirmay-Kalos, L. "Higher Order Automatic Differentiation with Dual Numbers", Periodica Polytechnica Electrical Engineering and Computer Science, 65(1), pp. 1–10, 2021. https://doi.org/10.3311/PPee.16341

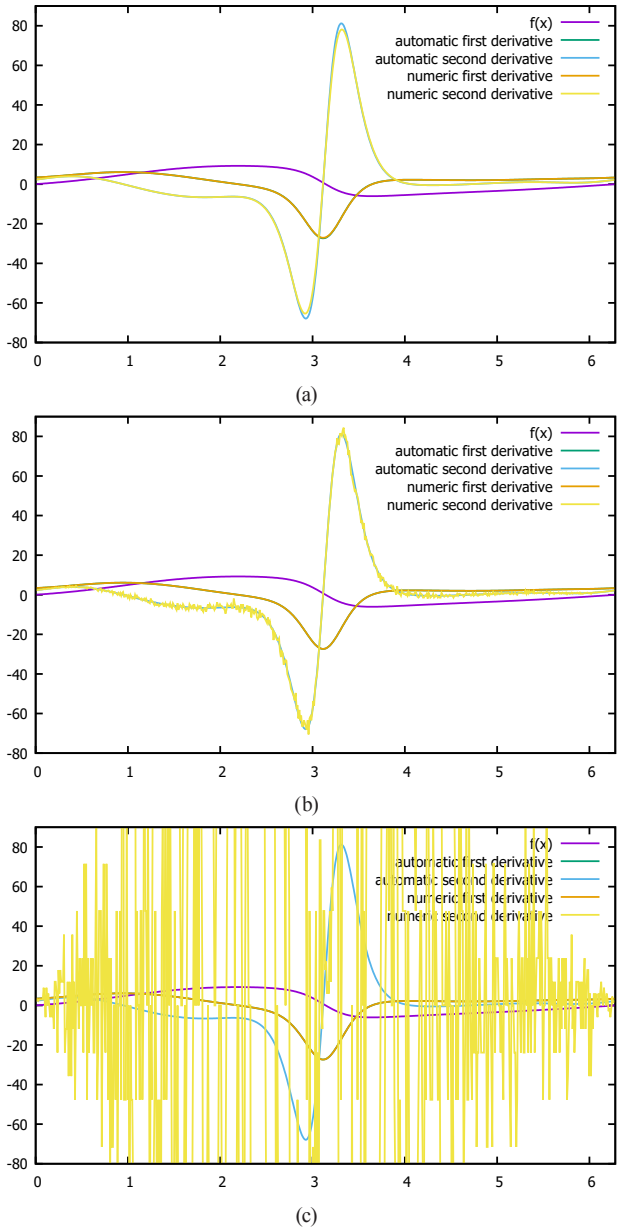


Fig. 1 Numerical differentiation of a function with different Δ values. The function is $X(t) = \sin(t)(\sin(t) + 3)4/(\tan(\cos(t)) + 2)$ (a) when $\Delta = 0.1$ the numerical derivative is not precise; (b) when $\Delta = 0.001$ the derivative becomes noisy; (c) when $\Delta = 0.0001$ the noise becomes intolerable. We used single precision floating point calculation.

program exactly up to the precision allowed by the finite digit representation of the computer. As this approach takes the function and the value where the function and the derivative should be determined, it is able to handle conditionals and to check and solve special cases.

Forward mode first order automatic differentiation is accomplished by replacing the algebra of real numbers by the algebra of dual numbers with arithmetic similar to the derivation rules. Concerning higher order differentiation, Fike and Alonso [13] proposed hyper-dual numbers that have the following form:

$$Z = x + y\epsilon_1 + z\epsilon_2 + w\epsilon_1\epsilon_2,$$

where the imaginary units satisfy $\epsilon_1^2 = \epsilon_2^2 = (\epsilon_1\epsilon_2)^2 = 0$. Hyper-dual numbers have three imaginary parts to mimic second order derivatives [14]. Following this construction, order N derivatives would need $2^N - 1$ imaginary units, making this approach prohibitively expensive for higher orders.

In this paper, we provide a simple and compact approach for higher order derivation. When dealing with derivatives up to order N, we store the function and the derivatives in an $N + 1$ dimensional vector, i.e. we use only the minimally required N imaginary units. To demonstrate the simplicity, we also present the C++ classes implementing the different solutions in this paper.

The structure of this paper is as follows. In Section 2 we first review the theory of dual numbers and their application in derivative computation. Section 3 presents our first result that generalizes dual numbers for multiple imaginary units and establishes the arithmetic rules for the computation of higher order derivatives. We also address the case of multi-variate functions and provide a solution for the computation of arbitrary derivatives with the application of recursive functions in Section 4.

Summarizing, the main contributions of this paper are:

- Generalization of dual numbers for multiple but minimal number of imaginary units to compute higher order derivatives.
- Generalization of dual numbers to handle higher order cross differentiation of multi-variate functions.
- Simple solution for the automatic higher order derivation with recursive functions.
- Automatic handling of 0/0 type ambiguities.

2 Dual numbers

Dual numbers are similar to ordinary complex numbers. Both of them are particular Clifford algebras or hyper-numbers of form $z = x + y\boldsymbol{i}$ where x and y are real numbers and $\boldsymbol{i}$ is the imaginary unit. We expect the existence of addition, multiplication and division with the properties of ordinary complex operations, like commutativity, distributivity, associativity. This means that when we compute the product of two such numbers, the result should have the same form, which necessitates the definition of $\boldsymbol{i}^2$ also in the form of $x + y\boldsymbol{i}$. The particular definition of $\boldsymbol{i}^2$ distinguishes different hyper-number types. For differentiation, we take *dual numbers* defined with the $\boldsymbol{i}^2 = 0$ fundamental

property, stating that $\boldsymbol{i}$ is an imaginary number, which does not belong to the real numbers, but its square can be replaced by zero. The arithmetic rules in this algebra are as follows. The addition or subtraction is

$$Z_1 \pm Z_2 = (x_1 + y_1\boldsymbol{i}) \pm (x_2 + y_2\boldsymbol{i}) = (x_1 \pm x_2) + (y_1 \pm y_2)\boldsymbol{i}. \quad (1)$$

The multiplication is

$$\begin{aligned} Z_1 Z_2 &= (x_1 + y_1\boldsymbol{i})(x_2 + y_2\boldsymbol{i}) \\ &= x_1x_2 + (y_1x_2 + x_1y_2)\boldsymbol{i} + y_1y_2\boldsymbol{i}^2 \\ &= x_1x_2 + (y_1x_2 + x_1y_2)\boldsymbol{i}. \end{aligned} \quad (2)$$

To establish the rule of division, both the numerator and the denominator are multiplied with the conjugate of the denominator to get rid of the imaginary part in the denominator:

$$\begin{aligned} \frac{Z_1}{Z_2} &= \frac{x_1 + y_1\boldsymbol{i}}{x_2 + y_2\boldsymbol{i}} = \frac{(x_1 + y_1\boldsymbol{i})(x_2 - y_2\boldsymbol{i})}{(x_2 + y_2\boldsymbol{i})(x_2 - y_2\boldsymbol{i})} \\ &= \frac{x_1x_2 + (y_1x_2 - x_1y_2)\boldsymbol{i} - y_1y_2\boldsymbol{i}^2}{(x_2^2 - y_2^2\boldsymbol{i}^2)} = \frac{x_1}{x_2} + \frac{y_1x_2 - x_1y_2}{x_2^2}\boldsymbol{i}. \end{aligned} \quad (3)$$

Examining Eqs. (1), (2), and (3), we can realize that the real part undergoes the same elementary operation as the dual number, while the imaginary part reflects the arithmetic rules of derivation for addition, multiplication and division. This means that considering function f and its derivative f' as the real and imaginary parts of a dual number $\mathcal{D}(f) = f + f'\boldsymbol{i}$, an arbitrary sequence of the four elementary operations results in the function value in the real part and the derivative in the imaginary part.

3 Generalizations of the dual numbers for higher order derivatives

As the dual number algebra provided a mechanism to compute function values and first derivatives, we aim at its generalization to cope with higher order derivatives. Suppose that we wish to compute the value of $f(t)$ for t, together with its derivatives $f'(t)$, $f''(t)$, ... up to order N. We use the notation of $f^{(j)}$ for the j^{th} derivative, thus $f^{(0)}(t) = f(t)$, $f^{(1)}(t) = f'(t)$, $f^{(2)}(t) = f''(t)$, etc.

To find an appropriate algebra, we should allow as many imaginary parts as many derivatives we wish to compute. The j^{th} imaginary unit is denoted by $\boldsymbol{i}_j$ and by convention we say that $\boldsymbol{i}_0 = 1$ to allow a uniform treatment for the real and imaginary parts. With these notations a generalized dual number representing a function and its derivatives has the following form:

$$\mathcal{D}(f) = \sum_{j=0}^{N} f^{(j)}\boldsymbol{i}_j.$$

Our goal is to define the arithmetic rules for such numbers, and find product $\boldsymbol{i}_j\boldsymbol{i}_k$ in particular, to make these rules similar to those of higher order derivation.

Derivatives of the sum or difference of two functions are just the sum or difference of the derivatives, respectively, so the generalized dual number rules of addition and subtraction are equivalent to the rules of derivation:

$$\mathcal{D}(f_1 \pm f_2) = \sum_{j=0}^{N} \left(f_1^{(j)} \pm f_2^{(j)}\right)\boldsymbol{i}_j = \mathcal{D}(f_1) \pm \mathcal{D}(f_2). \quad (4)$$

Let us consider multiplication. The product of two generalized dual numbers is

$$\mathcal{D}(f_1)\mathcal{D}(f_2) = \sum_{j=0}^{N} f_1^{(j)}\boldsymbol{i}_j \times \sum_{k=0}^{N} f_2^{(k)}\boldsymbol{i}_k = \sum_{j=0}^{N}\sum_{k=0}^{N} f_1^{(j)} f_2^{(k)}\boldsymbol{i}_j\boldsymbol{i}_k. \quad (5)$$

On the other hand, the dual number representation of product $f_1 f_2$ is

$$\mathcal{D}(f_1 f_2) = \sum_{n=0}^{N} (f_1 f_2)^{(n)}\,\boldsymbol{i}_n.$$

Substituting the formula of the n^{th} derivative of a product,

$$(f_1 f_2)^{(n)} = \sum_{m=0}^{n} f_1^{(m)} f_2^{(n-m)} \binom{n}{m}, \quad (6)$$

we get

$$\mathcal{D}(f_1 f_2) = \sum_{n=0}^{N}\sum_{m=0}^{n} f_1^{(m)} f_2^{(n-m)} \binom{n}{m}\boldsymbol{i}_n. \quad (7)$$

The derivatives can be computed with generalized dual number algebra if the two rules lead to identical results:

$$\mathcal{D}(f_1)\mathcal{D}(f_2) = \mathcal{D}(f_1 f_2).$$

Swapping the double summation in Eq. (7) and replacing m by j and $n - m$ by k, we obtain:

$$\mathcal{D}(f_1 f_2) = \sum_{j=0}^{N}\sum_{k=0}^{N-j} f_1^{(j)} f_2^{(k)} \binom{j+k}{k}\boldsymbol{i}_{j+k}. \quad (8)$$

Comparing this to Eq. (5) term by term, we obtain the following rule that makes the generalized dual number algebra equivalent to the higher order differentiation with respect to multiplication:

$$\boldsymbol{i}_j\boldsymbol{i}_k = \binom{j+k}{k}\boldsymbol{i}_{j+k} \text{ if } j + k \le N, \text{ and } 0 \text{ otherwise.} \quad (9)$$

Let us prove that this definition is consistent with the commutativity and associativity of multiplication. The commutativity is shown by

$$\boldsymbol{i}_j\boldsymbol{i}_k = \binom{j+k}{k}\boldsymbol{i}_{j+k} = \frac{(j+k)!}{j!k!}\boldsymbol{i}_{j+k} = \binom{j+k}{j}\boldsymbol{i}_{j+k} = \boldsymbol{i}_k\boldsymbol{i}_j. \quad (10)$$

Multiplication is also associative:

$$\left(\boldsymbol{i}_j\boldsymbol{i}_k\right)\boldsymbol{i}_l = \binom{j+k+l}{j+k}\binom{j+k}{k}\boldsymbol{i}_{j+k}\boldsymbol{i}_l = \frac{(j+k+l)!}{j!k!l!}\boldsymbol{i}_{j+k+l},$$

$$\boldsymbol{i}_j\left(\boldsymbol{i}_k\boldsymbol{i}_l\right) = \binom{j+k+l}{k+l}\binom{k+l}{l}\boldsymbol{i}_j\boldsymbol{i}_{k+l} = \frac{(j+k+l)!}{j!k!l!}\boldsymbol{i}_{j+k+l}.$$

The formulas developed for addition and multiplication guarantee that division also works since division is just a sequence of multiplying the numerator and the denominator with the conjugate of the denominator, and a division by a scalar when all imaginary units have disappeared from the denominator. Note that when we multiply denominator $f+\sum_{j=1}^{N} f^{(j)}\boldsymbol{i}_j$ with its conjugate $f-\sum_{j=1}^{N} f^{(j)}\boldsymbol{i}_j$, the result is

$$\left(f+\sum_{j=1}^{N} f^{(j)}\boldsymbol{i}_j\right)\left(f-\sum_{j=1}^{N} f^{(j)}\boldsymbol{i}_j\right) = f^2-\left(\sum_{j=1}^{N} f^{(j)}\boldsymbol{i}_j\right)^2.$$

In the imaginary part of the new denominator, products of imaginary units show up

$$\begin{aligned}\left(\sum_{j=1}^{N} f^{(j)}\boldsymbol{i}_j\right)^2 &= \sum_{j=1}^{N}\sum_{k=1}^{N-j} f^{(j)}f^{(k)}\boldsymbol{i}_j\boldsymbol{i}_k \\ &= \sum_{j=1}^{N}\sum_{k=1}^{N-j} f^{(j)}f^{(k)}\binom{j+k}{k}\boldsymbol{i}_{j+k}.\end{aligned} \tag{11}$$

If the minimal index j of non-zero imaginary part is $j_{\min}$ in the original denominator, then in the new denominator the minimal index of non-zero imaginary part is $2j_{\min}$. Since the indices of the imaginary units start with 1 and can be at most N, each multiplication with the current conjugate reduces the number of imaginary units, which will disappear in at most $\log_2(N) + 1$ steps.

3.1 Matrix representation for the case of $N = 2$

An intuitive way of imagining generalized dual numbers and imaginary units is to consider them as matrices. For the sake of simplicity, we discuss the $N = 2$ case, i.e. at most second derivatives are computed. A three-dimensional dual number $Z = x\boldsymbol{i}_0 + y\boldsymbol{i}_1 + z\boldsymbol{i}_2$ and units $\boldsymbol{i}_0, \boldsymbol{i}_1, \boldsymbol{i}_2$ can be regarded as 3 × 3 matrices:

$$Z = \begin{bmatrix} x & 2y & 2z \\ 0 & x & 2y \\ 0 & 0 & x \end{bmatrix},$$

$$\boldsymbol{i}_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},\ \boldsymbol{i}_1 = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix},\ \boldsymbol{i}_2 = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The matrices have been selected to meet the established formula of $\boldsymbol{i}_j\boldsymbol{i}_k = \binom{j+k}{k}\boldsymbol{i}_{j+k}$:

$$\boldsymbol{i}_0\boldsymbol{i}_j = \boldsymbol{i}_j\boldsymbol{i}_0 = \boldsymbol{i}_j,\ \boldsymbol{i}_1^2 = 2\boldsymbol{i}_2,\ \boldsymbol{i}_1\boldsymbol{i}_2 = \boldsymbol{i}_2\boldsymbol{i}_1 = 0.$$

Using matrices instead of generalized dual numbers is not practical but has theoretical advantages. The properties of arithmetic operations like the associativity and the distributivity of multiplication are inherited and thus need no further proof. Only the commutativity of multiplication has to be checked:

$$Z_1Z_2 = \begin{bmatrix} x_1x_2 & 2\left(x_1y_2+x_2y_1\right) & 2\left(x_1z_2+2y_1y_2+x_2z_1\right) \\ 0 & x_1x_2 & 2\left(x_1y_2+x_2y_1\right) \\ 0 & 0 & x_1x_2 \end{bmatrix}.$$

This is symmetric for the swapping of indices 1 and 2, thus commutativity of multiplication holds.

The second lesson learnt from the matrix analogy is that in this dual number algebra not only the division by zero is forbidden, but also the division by a pure imaginary dual number. The reason is that the matrices of $\boldsymbol{i}_1$ and $\boldsymbol{i}_2$ are singular, so they cannot be inverted.

3.2 C++ implementation

The C++ implementation of the generalized dual numbers of order N is in Algorithm 1 (addition and subtraction are trivial, and therefore not included). The solution should also prepare for nested functions, given also as a generalized dual number. Suppose that we wish to evaluate function f with derivatives f', f'', ... for an expression g of derivatives g', g'', ..., i.e. evaluate the value and the derivatives of $\left(f\left(g\right)\right)$. The dual number version is denoted by $\mathcal{D}\left(f\left(g\right)\right)$:

$$\begin{aligned}\mathcal{D}\left(f\left(g\right)\right) &= f\left(g\left(t\right)\right) + f'\left(g\left(t\right)\right)\times g'\left(t\right)\boldsymbol{i}_1 \\ &+\left(f''\left(g\left(t\right)\right)\times\left(g'\left(t\right)\right)^2 + f'\left(g\left(t\right)\right)\times g''\left(t\right)\right)\boldsymbol{i}_2 + \ldots.\end{aligned}$$

Algorithm 1 also contains the implementations of a constructor preparing for nested second derivatives and a few common mathematical functions with up to the second derivatives.

3.3 Multi-variate case

In the multi-variate case, the scalars defining the generalized dual number are also functions of other variables, associated with a completely different set of units. For the sake of notational simplicity, we consider the two-variate case. Let us denote the two variates of f by t_1 and t_2, and the $(j, l)^{\text{th}}$ cross derivative by

Algorithm 1 DnumN class and operations

```
#define foreach(j) for(int j=0; j<=N; j++)
class DnumN {
    float f[N+1];         // value and derivatives
public:
    DnumN(float v, float d = 0) {
        foreach(j) f[j] = 0;
        f[0] = v; f[1] = d;
    }
    float& operator()(int j) { return f[j]; }
    bool isReal() {
        foreach(j) if (j > 0 && f[j] != 0) return false;
        return true;
    }
    DnumN conjugate(){
        DnumN res;  foreach(j) res(j) = -f[j];
        res(0) = f[0];
        return res;
    }
};
DnumN operator/(DnumN f1, float f2) {
    DnumN res;  foreach(j) res(j) = f1(j) / f2;
    return res;
}
DnumN operator*(DnumN f1, DnumN f2) {
            DnumN res;
            foreach(j) foreach(k)
            if (j + k <= N) res(j+k) += f1(j) * f2(k) * choose(j+k, j);
            return res;
}
DnumN operator/(DnumN f1, DnumN f2) {
    if (f2.isReal()) return f1 / f2(0);
    else return (f1 * f2.conjugate()) / (f2 * f2.conjugate());
}
// v: value, d: first derivative, s: second derivative
DnumN::DnumN(float v, float d, float s, DnumN g) {
    f[0] = v; f[1] = d * g(1); f[2] = s * g(1) * g(1) + d * g(2);
}
DnumN Exp(DnumN g) {
    return DnumN(exp(g(0)), exp(g(0)), exp(g(0)), g);
}
DnumN Sin(DnumN g) {
    return DnumN(sin(g(0)), cos(g(0)), -sin(g(0)), g);
}
DnumN Pow(DnumN g, float n) {
    return DnumN(pow(g(0, 0), n), n * pow(g(0, 0), n - 1),
                                  n * (n - 1) * pow(g(0, 0), n - 2), g);
}
```

$$f^{(j,l)}\left(t_1,t_2\right)=\frac{\partial^{j+l} f}{\partial^j t_1 \partial^l t_2}.$$

We consider derivatives up to order N, i.e. $j + l \le N$. The dual number representation should also be multi-dimensional, where the units are $\boldsymbol{i}_{j,l}$:

$$\mathcal{D}(f)=\sum_{j=0}^{N}\sum_{l=0}^{N-j} f^{(j,l)}\boldsymbol{i}_{j,l}.$$

The addition/subtraction rule (Eq. (1)) remains valid, and the multiplication rule is a straightforward extension:

$$\boldsymbol{i}_{j,l}\boldsymbol{i}_{k,m}=\binom{j+k}{k}\binom{l+m}{l}\boldsymbol{i}_{j+k,l+m}$$

if $j + k + l + m \le N$ and zero otherwise.

The division is also similar to the one-variate case, we multiply the numerator and the denominator with the conjugate of the denominator until the denominator turns to be real. The complete C++ implementation is shown by Algorithm 2.

4 Recursive formulation of derivation

So far, we stored the function value in the real part of a generalized dual number and the derivatives were the imaginary parts. Alternatively, the function and derivatives can also be regarded as an $N + 1$ element vector $\boldsymbol{f}$, where $\boldsymbol{f}[0]$ is the function value and $\boldsymbol{f}[j]$ is the j^{th} derivative.

As the array contains increasing order derivatives, the derivative of the whole array is just a shift of array elements to left:

$$\frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}t}=\mathcal{D}\boldsymbol{f},$$

where *derivation operator* $\mathcal{D}$ copies element $j + 1$ into element j. Note that if $\boldsymbol{f}$ has $n + 1$ elements, then $\mathcal{D}\boldsymbol{f}$ can have only n valid elements. We also need an operation to reduce the number of elements without the application of derivation, which means simply ignoring the last element. For this,

Algorithm 2 DnumNxN class for two-variate functions

```
#define foreach(j,l) for(int j=0; j<=N; j++) for(int l=0; l<=N-j; l++)
class DnumNxN {
    float f[(N+1)*(N+2)/2];          // triangle matrix
public:
    DnumNxN(float v, float du = 0, float dv = 0) {
        foreach(j,l) (*this)(j,l) = 0;
        (*this)(0,0) = v; (*this)(1,0) = du; (*this)(0,1) = dv;
    }
    float& operator()(int j, int l) { return f[j*(N+1) - j*(j-1)/2 + l]; }
    bool isReal() {
        foreach(j, l) if ((j!=0 || l!=0) && (*this)(j,l)!=0) return false;
        return true;
    }
    DnumNxN conjugate() {
        DnumNxN res;  foreach(j,l) res(j,l) = -(*this)(j,l);
        res(0,0) = (*this)(0,0);
        return res;
    }
};
DnumNxN operator/(DnumNxN f1, float f2) {
    DnumNxN res;    foreach(j,l) res(j,l) = f1(j,l) / f2;
    return res;
}
DnumNxN operator*(DnumNxN f1, DnumNxN f2) {
    DnumNxN res;
    foreach(j, l) foreach(k, m)
        if (j+k+l+m <= N)  res(j+k,l+m) += f1(j,l) * f2(k,m) *
                                          choose(j+k, j) * choose(l+m, l);
    return res;
}
DnumNxN operator/(DnumNxN f1, DnumNxN f2) {
    if (f2.isReal()) return f1 / f2(0,0);
    else return (f1 * f2.conjugate()) / (f2 * f2.conjugate());
}
```

we introduce the *front operator* $\mathcal{F}$, i.e. $\mathcal{F}f$ is an n element array where the first n elements are identical to those of f.

As we remove elements both from the front and the end of the array, the appropriate data structure is the *deque*. We define two constructors, the default parametrization of the first one assumes that $f(t) = v$ and $f'(t) = 0$, which corresponds to constant v. For the derivation variable, i.e. for $f(t) = t$, the first derivative should be $f[1] = f'(t) = 1$, and all other derivatives are zero. The second constructor just merges the function value and all the derivatives (Algorithm 3). Global function Single also acts as a constructor that constructs a single element array from which recursion builds up the $N + 1$ element array.

The arithmetic operations are based on the recursive traversal of the respective computation trees. When two functions are added or subtracted, the same operation should be applied for the derivatives, which leads to a recursive implementation (Fig. 2).

To establish the rule of multiplication, we should recognize that the derivative of a product is the sum of two products:

$$\big(a(t)b(t)\big) = a'(t)b(t) + a(t)b'(t).$$

Algorithm 3 Derivation with recursion

```
struct Dvec : public deque<float> {
    Dvec(float v = 0, float d = 0, int n = N) : deque(n + 1, 0) {
        (*this)[0] = v;  if (n > 1) (*this)[1] = d;
    }
    Dvec(float v, Dvec d) : deque(d) { push_front(v); }
    bool isReal() { return (size() == 1); }
    Dvec F() {          // Front operator
        Dvec ffront(*this);  ffront.pop_back();  return ffront;
    }
    Dvec D() {          // Derivation operator
        Dvec fback(*this);  fback.pop_front();  return fback;
    }
};
Dvec Single(float e) { return Dvec(e, 0, 0); }
Dvec operator+(Dvec f1, Dvec f2) {
    return (f1.isReal() || f2.isReal()) ? Single(f1[0]+f2[0])
                                        : Dvec(f1[0]+f2[0], f1.D()+f2.D());
}
Dvec operator*(Dvec f1, Dvec f2) {
    return (f1.isReal() || f2.isReal()) ? Single(f1[0]*f2[0])
                        : Dvec(f1[0]*f2[0], f1.D()*f2.F()+f1.F()*f2.D());
}
Dvec operator/(Dvec f1, Dvec f2) {
    return (f1.isReal() || f2.isReal()) ? Single(f1[0]/f2[0])
                        : Dvec(f1[0]/f2[0], (f1.D()*f2-f1*f2.D())/(f2*f2));
}
Dvec Cos(Dvec g) {
    return g.isReal() ? Single(cos(g[0])) :
                        Dvec(cos(g[0]), -Sin(g.F())*g.D());
}
Dvec Sin(Dvec g) {
    return g.isReal() ? Single(sin(g[0])) :
                        Dvec(sin(g[0]), Cos(g.F())*g.D());
}
```

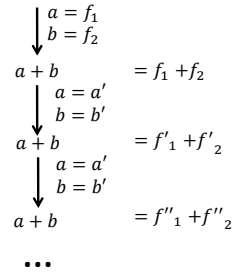


Fig. 2 Computation tree of the recursive addition.

Every term in the original sum spawns into two products, thus the computation up to the n^{th} derivative can be imagined as a binary tree where the root is the original product, and every level corresponds to a particular order of derivation. Every internal node represents a product of two functions $a(t)b(t)$ and has two children $a'(t)b(t)$ and $a(t)b'(t)$. The first child is a similar multiplication to its parent after $a \leftarrow a'$ substitution, and the second child is a similar after $b \leftarrow b'$ substitution (Fig. 3). The sum of terms on level j is the element j in vector $f_1 f_2$.

To attack division, we should recognize that the derivative of the ratio of two functions is the sum (or difference) of other ratios of functions:

$$\left(\frac{a(t)}{b(t)}\right) = \frac{a'(t)}{b(t)} + \frac{-a(t)b'(t)}{b^2(t)}.$$

In this computation tree, a ratio a/b also has two children performing the same operation. The left child gets the two functions after $a \leftarrow a'$ substitution, the right one gets them after the $a \leftarrow -ab'$ and $b \leftarrow b'$ substitutions (Fig. 4).

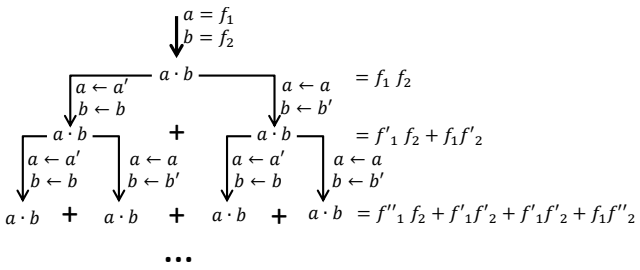


Fig. 3 Computation tree of the recursive multiplication.

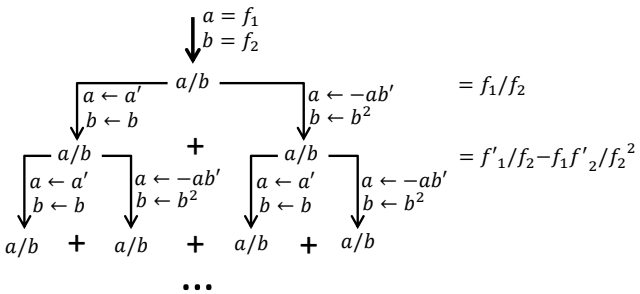


Fig. 4 Computation tree of the recursive division.

The chain rule for higher order derivatives can also be expressed in a recursive way.

Note that we do not have to specify all derivatives, just the first derivative, provided that it is also in the set of elementary functions with prepared derivatives.

5 Handling 0/0 type ambiguities

When we encounter a 0/0 type undefined division or the numerator and denominator are close to zero causing numerical instability, the l'Hospital rule can automatically be applied. If $f_1(t_0)$ and $f_2(t_0)$ are zero, then

$$\lim_{t\to t_0}\frac{f_1(t)}{f_2(t)}=\frac{f_1'(t_0)}{f_2'(t_0)},\qquad \left(\frac{f_1}{f_2}\right)'=\frac{f_1''f_2'-f_1'f_2''}{2(f_2')^2}.$$

If $f_1'(t_0)$ and $f_2'(t_0)$ are also zero, the l'Hospital rule can be applied recursively. Based on this, we modify the division operator, find the first derivatives of no 0/0 ambiguity, and use their ratio (Algorithm 4).

Fig. 5 depicts the $\sin(t)/t$ function and its first and second derivatives evaluated with the proposed method.

6 Applications

The presented methods can be applied in problems where higher order derivatives are needed or when 0/0 type ambiguities should also be solved. Here we present two particular applications belonging to dynamics simulation and differential geometry. Our first example uses the Dvec class, i.e. the recursive evaluation, the second takes advantage of the DnumNxN class, i.e. the two-variate generalized dual number.

Algorithm 4 Modified division with the l'Hospital rule

```
Dvec operator/(Dvec f1, Dvec f2) {
    int j = 0;
    if (fabs(f2[0]) < epsilon && fabs(f1[0]) < epsilon) {
        for (j = 0; j < f2.size(); j++) if (fabs(f2[j]) >= epsilon) break;
    }
    return (f1.isReal() || f2.isReal()) ? Single(f1[j]/f2[j])
                    : Dvec(f1[j]/f2[j], (f1.D()*f2-f1*f2.D())/(f2*f2));
}
```

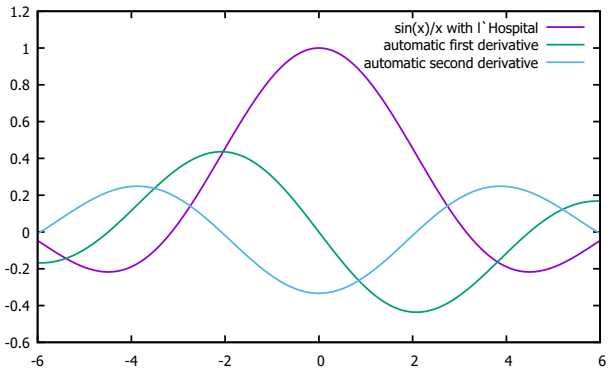


Fig. 5 Plot of the $\sin(t)/t$ function and of its derivatives evaluated with the proposed method and applying the l'Hospital rule.

6.1 Path animation and aesthetics of curves

Suppose the path of an object is defined by a time dependent function $\vec{r}(t)$, or alternatively, by the three scalar functions of Cartesian coordinates $X(t)$, $Y(t)$, $Z(t)$. Transformation of the object requires velocity $\vec{r}'(t)$ that defines the current heading direction, and acceleration $\vec{r}''(t)$, from which the Frenet frame can be calculated. The *curvature* is also obtained from these quantities:

$$\kappa(t)=\frac{|\vec{r}'\times\vec{r}''|}{|\vec{r}'|^3}. \tag{12}$$

Let us take the following example path ($X(t)$ with its automatic and numerical derivatives are shown by Fig. 1):

$$X(t)=\frac{\sin(t)(\sin(t)+3)0.4}{\tan(\cos(t))+2},$$

$$Y(t)=\frac{\cos(\sin(t)8+1)1.2+0.2}{(\sin(t)\sin(t))^3+2},$$

$$Z(t)=\exp\left(-\frac{(X^2(t)+Y^2(t))}{100}\right).$$

In fairing the uniformity of the curvature is maximized, thus the integral of the absolute value of the curvature derivative with respect to the path length should be minimized. This requires the computation of the following local quantity where s is the path length:

$$\left|\frac{\mathrm{d}\kappa(t(s))}{\mathrm{d}s}\right|=\left|\frac{\mathrm{d}\kappa(t)}{\mathrm{d}t}\right|\times\frac{1}{|\vec{r}'(t)|}. \tag{13}$$

We introduce the Vec3 class that contains three Dvec generalized dual number objects to represent the three Cartesian coordinates. Vector operations are implemented for this vector of functions, including cross product Cross and absolute value Abs (Algorithm 5).

Algorithm 5 Vec3 for vector derivation

```
struct Vec3 {
    Dvec X, Y, Z;
    Vec3(Dvec X0, Dvec Y0, Dvec Z0) : X(X0), Y(Y0), Z(Z0) { }
    Vec3 D() { return Vec3(X.D(), Y.D(), Z.D()); }
    vec3 operator()(int i) { return vec3(X[i], Y[i], Z[i]); }
};
Dvec Abs(Vec3 v) { return Pow(v.X*v.X + v.Y*v.Y + v.Z*v.Z, 0.5); }
Vec3 Cross(Vec3 v1, Vec3 v2) {
    return Vec3(v1.Y * v2.Z - v1.Z * v2.Y,
                v1.Z * v2.X - v1.X * v2.Z, v1.X * v2.Y - v1.Y * v2.X);
}
```

The program of Algorithm 6 takes parameter t and converts it to a function with derivatives T. As this is the derivation variable, its derivative is set to 1. The corresponding point of the path together with the derivatives up to order 3 are stored in R. The curvature and its derivative are in curv[0] and curv[1] after evaluating Eq. (12).

Fig. 6 shows the path and the axes of the Frenet frame, which are parallel with the velocity, centripetal acceleration, and binormal direction. The diffuse color of the curve in the left image depicts the curvature. We used the rainbow colors, blue corresponds to zero and red to the maximum value.

In the right image of Fig. 6, the aesthetic measure of Eq. (13) depending even on the third derivative is visualized. The aim of fairing would be to force this measure to go to zero, which would be visualized with blue diffuse color.

6.2 Gaussian curvature of parametric surfaces

The Gaussian curvature κ of a parametric surface can be computed as the ratio of the determinants of the second and the first fundamental forms. The first fundamental form contains the partial derivatives $\vec{r}'_u$ and $\vec{r}'_v$ of the surface function $\vec{r}(u,v)$ with respect to parameters u and v, the second fundamental form the second derivatives $\vec{r}''_{uu}$ and $\vec{r}''_{vv}$ and also the cross derivative $\vec{r}''_{uv}$. The Gaussian curvature is

$$\kappa = \frac{\left(\vec{n}\cdot\vec{r}''_{uu}\right)\left(\vec{n}\cdot\vec{r}''_{vv}\right)-\left(\vec{n}\cdot\vec{r}''_{uv}\right)^2}{\left(\vec{r}'_u\cdot\vec{r}'_u\right)\left(\vec{r}'_v\cdot\vec{r}'_v\right)-\left(\vec{r}'_u\cdot\vec{r}'_v\right)^2},$$

where

$$\vec{n} = \frac{\vec{r}'_u\times\vec{r}'_v}{\left|\vec{r}'_u\times\vec{r}'_v\right|}$$

is the unit normal.

In order to visualize the surface together with the curvature information, we evaluate the surface function $\vec{r}(u,v)$ together with up to second order derivatives. Suppose we have function eval(U, V, X, Y, Z) that gets the Dvec version U and V of parameters u and v, respectively, and computes not only the $\vec{r} = (X,Y,Z)$, but also the first and second derivatives.

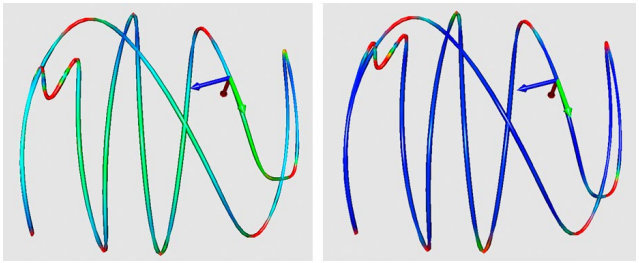
Note the conversion of u to U. The value is u, the derivative with respect to u is 1, and the derivative with respect to v is zero. The conversion of v is similar, but now the derivative with respect to u is 0, and the derivative with respect to v is 1.

Having called eval(U, V, X, Y, Z) all derivatives are available from which the normal vector, determinants of the fundamental forms, and the Gaussian curvature can be computed (Algorithm 7).

As the derivatives are automatically calculated, function eval(U, V, X, Y, Z) looks like only the surface function implementation. Tables 1 and 2 show the particular implementations for the surface types of Fig. 7.

Algorithm 6 Curvature and fairness computation

```
Dvec T(t, 1);                 // derivation variable
Vec3 R;                       // path point and derivatives
R.X = Sin(T)*(Sin(T)+3)*0.4 / (Tan(Cos(T))+2);
R.Y = (Cos(Sin(T)*8+1)*1.2+0.2) / (Pow(Sin(T)*Sin(T),3)+2);
R.Z = Exp((X*X + Y*Y) / (-100));
vec3 r = R(0), rt = R(1), rtt = R(2);      // point, velocity, acceleration
Dvec curv = Abs(Cross(R.D(),R.D().D()))/Pow(Abs(R.D()),3);
float curvature = curv[0];                 // Eq. (12)
float fairness = curv[1] / length(R(1));   // Eq. (13)
```



(a) (b)

Fig. 6 (a) Path animation with the visualization of the curvature and (b) the curvature derivative together with the Frenet frame.

7 Conclusions

This paper presented simple C++ classes that can solve the automatic derivation problem up to arbitrary order. We also considered the multi-variate case. To handle 0/0 ambiguities, the application of the l'Hospital rule is automated. The classes can be used in any program where higher order derivatives of analytical functions are needed. We presented two applications, the first addressed path animation and curve fairness, the second the Gaussian curvature of surfaces. The programs can be downloaded from [15].

Algorithm 7 Gaussian curvature of surfaces

```
// Computation of surface function and derivatives
DnumNxN U(u, 1, 0), V(v, 0, 1), X, Y, Z;
eval(U, V, X, Y, Z); // (U, V) -> X, Y, Z
// First fundamental form
vec3 ru = vec3(X(1, 0), Y(1, 0), Z(1, 0));
vec3 rv = vec3(X(0, 1), Y(0, 1), Z(0, 1));
vec3 n = normalize(cross(ru, rv));
float E = dot(ru,ru), F = dot(ru,rv), G = dot(rv,rv);
// Second fundamental form
vec3 ruu = vec3(X(2, 0), Y(2, 0), Z(2, 0));
vec3 ruv = vec3(X(1, 1), Y(1, 1), Z(1, 1));
vec3 rvv = vec3(X(0, 2), Y(0, 2), Z(0, 2));
float L = dot(n,ruu), M = dot(n,ruv), N = dot(n,rvv);
// Curvature is the ratio of the determinants
float curvature = (L * N - M * M) / (E * G - F * F);
```

Table 1 Surface evaluation functions eval(U, V, X, Y, Z) of the sphere, tractricoid, cylinder, torus, and Möbius strip.

Sphere	Tractricoid	Cylinder	Torus	Möbius
U=U*2*M_PI;	U=U*2*M_PI;	U=U*2*M_PI;	U=U*2*M_PI;	U=U*M_PI;
V=V*M_PI;	V=V*h;	V=V*h;	V=V*2*M_PI;	V=(V-0.5)*w;
X=Cos(U)*Sin(V);	X=Cos(U)/Cosh(V);	X=Cos(U);	X=(Cos(U)*r+R)*Cos(V);	X=(Cos(U)*V+R)*Cos(U*2);
Y=Sin(U)*Sin(V);	Y=Sin(U)/Cosh(V);	Y=Sin(U);	Y=(Cos(U)*r+R)*Sin(V);	Y=(Cos(U)*V+R)*Sin(U*2);
Z=Cos(V);	Z=V-Tanh(V);	Z=V;	Z=Sin(U)*r;	Z=Sin(U)*V;

Table 2 Surface evaluation functions eval(U, V, X, Y, Z) of the Klein bottle and the Boy surface.

Klein bottle	Boy surface
U = U*M_PI*2;	U = (U-0.5)*M_PI;
V = V*M_PI*2;	V = V * M_PI;
DnumNxN a = Cos(U)*(Sin(U)+1)*0.3;	float r2 = sqrt(2);
DnumNxN b = Sin(U)*0.8;	DnumNxN denom = (Sin(U*3)*Sin(V*2)*(-3/r2)+3)*1.2;
DnumNxN c = (Cos(U)*(-0.1)+0.2);	DnumNxN CosV2 = Cos(V)*Cos(V);
X = a+c*((U(0,0)>M_PI) ? Cos(V+M_PI) : Cos(U)*Cos(V));	X = (Cos(U*2)*CosV2*r2+Cos(U)*Sin(V*2)) / denom;
Y = b+((U(0,0)>M_PI) ? 0 : c*Sin(U)*Cos(V));	Y = (Sin(U*2)*CosV2*r2-Sin(U)*Sin(V*2)) / denom;
Z = c*Sin(V);	Z = (CosV2*3) / denom;

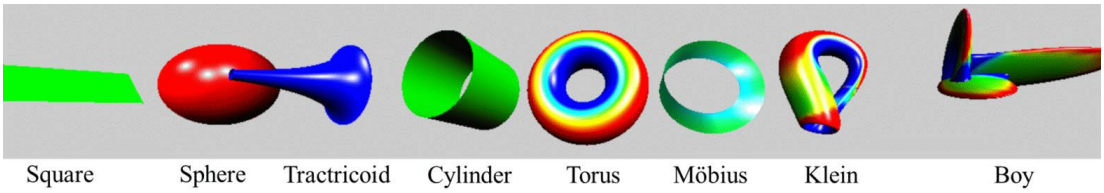


Fig. 7 Rendering of parametric surfaces with diffuse color hues defined by the Gaussian curvature. Green depicts zero curvature. From blue to green, the curvature is negative. From green to red, the curvature is positive. The square and the cylinder have zero curvature everywhere. The sphere has constant positive curvature, the tractricoid has constant negative curvature. The curvature of other surfaces depends on the location.

Acknowledgement

This work has been supported by OTKA K–124124.

References

[1] Sapidis, N., Farin, G. "Automatic fairing algorithm for B-spline curves", Computer-Aided Design, 22(2), pp. 121–129, 1990. https://doi.org/10.1016/0010-4485(90)90006-X

[2] Vaitkus, M., Várady, T. "Parameterizing and extending trimmed regions for tensor-product surface fitting", Computer-Aided Design, 104, pp. 125–140, 2018. https://doi.org/10.1016/j.cad.2017.11.008

[3] Várady, T., Martin, R. "Chapter 26 - Reverse Engineering", In: Farin, G., Hoschek, J., Kim, M. S. (eds.) Handbook of Computer Aided Geometric Design, Elsevier, Amsterdam, The Netherlands, 2002, pp. 651–681. https://doi.org/10.1016/B978-044451104-1/50027-7

[4] Szirmay-Kalos, L. "Filtering and Gradient Estimation for Distance Fields by Quadratic Regression", Periodica Polytechnica Electrical Engineering and Computer Science, 59(4), pp. 175–180, 2015. https://doi.org/10.3311/PPee.8529

[5] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., Siskind, J. M. "Automatic Differentiation in Machine Learning: a Survey", Journal of Machine Learning Research, 18, pp. 1–43, 2018. [online] Available at: http://jmlr.org/papers/v18/17-468.html [Accessed: 09 November 2019]

[6] Würfl, T., Hoffmann, M., Christlein, V., Breininger, K., Huang, Y., Unberath, M., Maier, A. K. "Deep Learning Computed Tomography: Learning Projection-Domain Weights From Image Domain in Limited Angle Problems", IEEE Transactions on Medical Imaging, 37(6), pp. 1454–1463, 2018. https://doi.org/10.1109/TMI.2018.2833499

[7] Szirmay-Kalos, L., Kacsó, Á., Magdics, M., Tóth, B. "Dynamic PET Reconstruction on the GPU", Periodica Polytechnica Electrical Engineering and Computer Science, 62(4), pp. 134–143, 2018. https://doi.org/10.3311/PPee.11739

[8] Kalnins, R. D., Markosian, L., Meier, B. J., Kowalski, M. A., Lee, J. C., Davidson, P. L., Webb, M., Hughes, J. F., Finkelstein, A. "WYSIWYG NPR: drawing strokes directly on 3D models", ACM Transactions on Graphics, 21(3), pp. 755–762, 2002. https://doi.org/10.1145/566654.566648

[9] Fornberg, B. "Numerical Differentiation of Analytic Functions", ACM Transactions on Mathematical Software, 7(4), pp. 512–526, 1981. https://doi.org/10.1145/355972.355979

[10] Davenport, J. H., Siret, Y., Tournier, E. "Computer algebra: systems and algorithms for algebraic computation", Academic Press, London, UK, 1988.

[11] Hoffmann, P. H. W. "A Hitchhiker's Guide to Automatic Differentiation", Numerical Algorithms, 72(3), pp. 775–811, 2016. https://doi.org/10.1007/s11075-015-0067-6

[12] Naumann, U. "The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation", Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2012. https://doi.org/10.1137/1.9781611972078

[13] Fike, J. A., Alonso, J. J. "The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations", In: 49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, Orlando, FL, USA, 2011, Article Number: AIAA 2011-886. https://doi.org/10.2514/6.2011-886

[14] Neuenhofen, M. "Review of theory and implementation of hyper-dual numbers for first and second order automatic differentiation", [cs.MS], arXiv:1801.03614, Cornell University, Ithaca, NY, USA, 2018, [online] Available at: https://arxiv.org/abs/1801.03614 [Accessed: 09 November 2019]

[15] Szirmay-Kalos, L. "AutomaticDiff.zip", [computer program], Available at: http://cg.iit.bme.hu/~szirmay/AutomaticDiff.zip [Accessed: 05 February 2020]