Automatic Evaluations of Cross-Derivatives

Andreas Griewank* Lutz Lehmann Hernan Leovey Marat Zilberman†

version of October 9, 2009

Abstract

Cross–derivatives are mixed partial derivatives that are obtained by differentiating at most once in every coordinate direction. They are a computational tool in combinatorics and high–dimensional integration. Here we present two methods of computing exact values of all cross–derivatives at a given point both following the general philosophy of automatic differentiation. Implementation details are discussed and numerical results given.

1 Introduction and Motivation

With the term *cross–derivatives* we will refer to those mixed partial derivatives where differentiation w.r.t. each variable is done at most once. If $f:\mathbb{R}^n \to \mathbb{R}$ is a n times continuously differentiable function, we can associate with each subset $\mathbf{i} \subset D = \{1,2,\ldots,n\}$ of size $|\mathbf{i}|$ the cross-derivative $f_{\mathbf{i}}$ where the partial derivatives are w.r.t. the variables x_j with index $j \in \mathbf{i}$, that is

$$f_{\mathbf{i}}(x) = \left(\prod_{j\in\mathbf{i}} \tfrac{\partial}{\partial x_j} f\right)(x) = \frac{\partial^k f}{\partial x_{i_1}\ldots\partial x_{i_k}}(x), \qquad \mathbf{i} = \{i_1, i_2, \ldots, i_k\}\,.$$

Just as there are 2^n subsets of D, there also are 2^n cross–derivatives. The cross–derivative corresponding to the empty set is the original function, $f_\emptyset(x) = f(x)$. Throughout the paper we use the realistic assumption that f is evaluated by a procedure that can be interpreted as a sequence of elementary arithmetic operations and intrinsic functions as is customary in automatic, or algorithmic differentiation [GW08]. If u and w are functions in n variables that appear as intermediate result in the evaluation of f, then the cross–derivatives of the result of elementary operations $g(u,w)$ or functions $h(u)$ can be expressed in terms of the cross–derivatives of u and w and the partial derivatives of the functions g resp. h. For most elementary operations and functions these latter partial derivatives have a structure that allows a fast implementation.

In Section 2 we develop such a direct implementation with a complexity of $O(3^n)$ per elementary operation. In Section 3 we consider an alternative method based on interpolation of univariate Taylor polynomials with an overall complexity of $O(n^2 2^n)$ per elementary operation. However, as we see in

*partially supported by the DFG research center ”MATHEON, Mathematics for the key technologies” in Berlin
†IAESTE internship supported by grant of DAAD

the numerical results Section 4 the resulting accuracy is less due to cancellation errors. In the remainder of the introduction we sketch a few applications.

Applications of cross–derivatives are known in the computation of combinatorial numbers in graph theory and in statistics for the estimation of the effective dimension of functions over high–dimensional domains leading to efficient integration methods.

1.1 Cross–derivatives applied to Combinatorics

We present here examples which demonstrates the use of evaluation of high order derivatives in the field of combinatorics. Both examples, the computation of the number of Hamiltonian cycles and the evaluation of the permanent of a matrix, are #P–complete [GJ79], that is, they are enumeration problems that have associated decision problems, that are NP–complete. The existence of polynomial time evaluation algorithms would imply that P=NP. Thus we can expect an exponentially growing runtime for any algorithm computing this solution. The best available algorithm to compute the permanent has indeed a complexity of $O(n2^n)$ arithmetic operations for $(n \times n)$–matrices [Rys63].

Kubota in [Kub08] presents transformations of both problems to the determination of certain coefficients of suitably constructed multivariate polynomials. These coefficients are of the maximally mixed terms of degree n polynomials in n variables. Thus, they are also obtainable as highest order cross–derivatives. Kubota evaluates them using the transformation

$$g(t) = f\left(t, t^2, t^4, \dots, t^{2^{n-1}}\right)$$

to univariate polynomials with $\deg g \leq n2^{n-1}$, and obtains the required coefficient as the degree $(2^n - 1)$ coefficient of g. Using FFT-based methods for efficient evaluation of g results in operations counts of $O(n2^n)$ times the basic operations count of evaluating f, which is $O(n^2)$ for the permanent and $O(n^3)$ for the Hamiltonian cycles. Using our second method to evaluate the full set of cross–derivatives we arrive at similar complexity that differs only in the power of n.

1.1.1 Hamiltonian cycles

Given a simple directed graph $G = (V, E)$, $E \subset V \times V$, a Hamiltonian cycle is a cycle of edges in E that visits each vertex in V exactly once (except the initial and final vertex that is visited twice). We can count the total number of Hamiltonian cycles of graph V by evaluation of cross–derivatives. First, define an adjacency matrix A to represent the edges of the graph, that is $A_{ij} = 1$ if there exists an edge from the vertex v_i to the vertex v_j, otherwise $A_{ij} = 0$. Then, we define a multivariate diagonal matrix

$$X(x_1, x_2, \dots, x_n) = \operatorname{diag}(x_1, x_2, \dots, x_n) = \begin{pmatrix} x_1 & 0 & \dots & 0 \\ 0 & x_2 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & x_n \end{pmatrix}$$

And finally, we construct the multivariate matrix $H(x_1, x_2, \dots, x_n) = (X \cdot A)^n$. Note that $(X \cdot A)^k$ is nonzero at position (i, j) if there is a directed path from edge v_i to edge v_j of length k. The monomials of that entry actually trace that path, if there is a directed path $v_i = v_{i_1} \to v_{i_2} \to \cdots \to v_{i_k} = v_j$, then the monomial $x_{i_1}x_{i_2}\dots x_{i_{k-1}}$ will be present. In consequence, each Hamiltonian cycle will contribute a monomial $x_1x_2\dots x_n$ to each diagonal entry of $H=(X\cdot A)^n$. Moreover, the number of Hamiltonian cycles $c(G)$ can be obtained as the coefficient of that monomial, which in turn can be computed as the highest order cross–derivative of the first diagonal entry by the following equation:

$$c(G)=\frac{\partial H_{1,1}}{\partial x_1\partial x_2\dots\partial x_n}(x_1,x_2,\dots,x_n)$$

1.1.2 Computation of the permanent of a matrix

Given a matrix A, its permanent is defined as:

$$\mathrm{per}(A)=\sum_{\sigma\in\mathscr{S}^n}a_{1\sigma(1)}a_{2\sigma(2)}\cdots a_{n\sigma(n)}$$

where $\sigma\in\mathscr{S}^n$ runs over all the permutations of $\{1,2,\dots,n\}$. One way to calculate the permanent is to define an n–variate polynomial f:

$$f(x_1,x_2,\dots,x_n)=\prod_{i=1}^{n}(a_{i1}x_1+a_{i2}x_2+\cdots+a_{in}x_n)$$

and to compute its n–th order cross–derivative:

$$\mathrm{per}(A)=\frac{\partial^n f}{\partial x_1\partial x_2\dots\partial x_n}(x_1,x_2,\dots,x_n)$$

1.2 High–dimensional Quasi-Monte Carlo integration

Quasi-Monte Carlo methods are deterministic methods for approximating the integral:

$$I(f)=\int_{C^n}f(\boldsymbol{x})d\boldsymbol{x},\quad \boldsymbol{x}=(x_1,\dots,x_n)$$

where $C^n=[0,1]^n$ is the unit cube in $\mathbb{R}^n$. Such integrals over a high–dimensional cube result frequently from discretizations of path integrals, especially in mathematical finance, electric power utilities and statistical physics.

The quasi-Monte Carlo methods are efficient equal–weight quadrature rules for these integrals of the form

$$Q_{N,n}(f)=\frac{1}{N}\sum_{i=0}^{N-1}f(\boldsymbol{z}_i),$$

where $\boldsymbol{z}_0,\dots,\boldsymbol{z}_{N-1}$ are the first N points of a quasi–random (such as a *low discrepancy sequence*) in $[0,1]^n$ [KS05], for instance a Sobol sequence.

1.2.1 Error estimates

Zaremba [Zar68] provides an expression for the error of quasi-Monte Carlo quadrature rules in terms of integrals over cross–derivatives:

Proposition 1.1 (Zaremba identity) *Let f be a C^n function over the unit cube $[0,1]^n$, $\boldsymbol{z} = (\boldsymbol{z}_k)_{k\leq N}$ be a finite sequence of points in the interior of $[0,1]^n$, and define, using the boxes $[0,\boldsymbol{x}) := [0,x_1)\times\cdots\times[0,x_n) \subset C^n$,*

$$disc_{\boldsymbol{z}}(\boldsymbol{x}) := \frac{1}{N}\sum_{k=1}^{N} 1_{[0,\boldsymbol{x})}(\boldsymbol{z}_k) \,-\, x_1\cdots x_n$$

as the local discrepancy function over $[0,1]^n$. Then the Zaremba identity states:

$$\frac{1}{N}\sum_{k=1}^{N} f(\boldsymbol{z}_k) - \int_{[0,1]^n} f(\boldsymbol{x})d\boldsymbol{x} \,=\, \sum_{\emptyset\neq u\subset D} (-1)^{|u|}\int_{[0,1]^{|u|}} disc_{\boldsymbol{z}}(\boldsymbol{x}_u,\mathbf{1})\,\frac{\partial^{|u|}}{\partial \boldsymbol{x}_u} f(\boldsymbol{x}_u,\mathbf{1})\,d\boldsymbol{x}_u.$$

Here $D = \{1,\ldots,n\}$ is the set of coordinates indices, $|u|$ is the cardinality of a subset $u\subset D$, by $(\boldsymbol{x}_u,\boldsymbol{1})$ we denote the vector $\boldsymbol{x}\in[0,1]^n$ with all components whose indices are not in u replaced by 1.

This identity results directly from the formula for integration by parts for multidimensional Stieltjes integrals. Applying the Cauchy–Schwarz or more generally the Hölder inequalities to this decomposition, various error estimates result. The Koksma–Hlawka [Hla61] inequality applies the Hölder inequality for the L^1 norm to this decomposition to obtain a simple expression for the integration error through quasi-Monte Carlo quadrature rules. Let f be a function with bounded Hardy and Krause variation $V^1(f) < \infty$ over the cube $[0,1]^n$ defined by

$$V^1(f) := \sum_{\emptyset\neq u\subset D} V^1_u(f)\,;\quad \text{where}\quad V^1_u(f) := \int_{[0,1]^{|u|}} \left|\frac{\partial^{|u|}}{\partial \boldsymbol{x}_u} f(\boldsymbol{x}_u,1)\right| d\boldsymbol{x}_u\,.$$

Then the Koksma-Hlawka inequality states:

$$\left|\frac{1}{N}\sum_{i=1}^{N} f(\boldsymbol{z}_i) - \int_{[0,1]^n} f(\boldsymbol{x})d\boldsymbol{x}\right| \leq D^\star_N(\boldsymbol{z})\cdot V^1(f)$$

where

$$D^\star_N(\boldsymbol{z}) := \sup_{\boldsymbol{x}\in[0,1]^n} |disc_{\boldsymbol{z}}(\boldsymbol{x})|\,.$$

1.2.2 Effective dimension and function decomposition

X. Wang and K.T. Fang [WF03] have investigated the concept of effective dimensions in high dimensional integrations, proposing to use this as the real dimension of the problem in some specific sense. Typically, the definition of effective dimensions is based on an ANOVA (analysis of variance, see [Sob01]) decomposition of the given function f.

The *ANOVA decomposition* is a way of decomposing a function into a sum of simpler functions. Let $D = \{1,\ldots,n\}$. For any subset $\mathbf{i}\subset D$, let $|\mathbf{i}|$ denote its cardinality and $(D-\mathbf{i})$ be its complementary set in D. Let $\boldsymbol{x}_{\mathbf{i}} = (x_j : j \in \mathbf{i})$ be the $|\mathbf{i}|-$dimensional vector containing the coordinates of $\boldsymbol{x}$ with indices in $\mathbf{i}$. Let $C^{\mathbf{i}}$ denote the $|\mathbf{i}|-$dimensional unit cube involving the coordinates in $\mathbf{i}$. Now assume that f is a square integrable function. Then we can write f as the sum of 2^n ANOVA terms:

$$f(\boldsymbol{x}) = \sum_{\mathbf{i} \subset D} f^{\mathbf{i}}(\boldsymbol{x})$$

where the ANOVA terms $f^{\mathbf{i}}(\boldsymbol{x})$ are defined recursively by

$$f^{\mathbf{i}}(\boldsymbol{x}) := \int_{C^{D-\mathbf{i}}} f(\boldsymbol{x}) d\boldsymbol{x}_{D-\mathbf{i}} - \sum_{\mathbf{j} \subsetneq \mathbf{i}} f^{\mathbf{j}}(\boldsymbol{x})$$

and $f^{\emptyset} := I(f)$. The sum of the RHS is over strict subsets $\mathbf{j} \neq \mathbf{i}$, and we use the convention $\int_{C^{\emptyset}} f(\boldsymbol{x}) d\boldsymbol{x}_{\emptyset} := f(\boldsymbol{x})$. The ANOVA terms enjoy the following interesting properties:

1. $\int_0^1 f^{\mathbf{i}}(\boldsymbol{x}) dx_j = 0$ for $j \in \mathbf{i}$.
2. The decomposition is orthogonal, in that $\int_{C^d} f^{\mathbf{i}}(\boldsymbol{x}) f^{\mathbf{j}}(\boldsymbol{x}) d\boldsymbol{x} = 0$ whenever $\mathbf{i} \neq \mathbf{j}$.
3. Let $\sigma^2 := \int_{C^d} f(\boldsymbol{x})^2 d\boldsymbol{x} - (I(f))^2$ be the variance of f, then we have:

$$\sigma^2 = \sum_{\mathbf{i} \subset D} \sigma_{\mathbf{i}}^2(f), \quad \text{where} \quad \sigma_{\mathbf{i}}^2(f) := \int_{C^d} f^{\mathbf{i}}(\boldsymbol{x})^2 d\boldsymbol{x}$$

   for $|\mathbf{i}| > 0$ is the variance of $f^{\mathbf{i}}$ and $\sigma_{\emptyset}^2(f) := 0$.

Let us assume from now on that $f \in C^n([0,1]^n)$. If some cross–derivative $f_{\mathbf{i}}$ vanishes identically throughout the domain, the corresponding ANOVA terms $f^{\mathbf{j}}$ with $\mathbf{j} \supset \mathbf{i}$ do vanish too and the function is *partially separable* in the sense of Griewank and Toint [GT81]. Using the size of the cross–derivatives we can now estimate how close this function is to a partially separable function of low order. To that end we define effective dimensions of a function in a differentiable sense:

Definition 1.2 *Given an $0 < \varepsilon < 1$, we will say that f is of $(V_{HK}, S)-$effective dimension k, if*

$$\sum_{u\,:\,|u| \geq k} V_u^1(f) \leq \varepsilon V^1(f).$$

and similarly that f is of $(V_{HK}, T)-$effective dimension k, if there exists $i_1, \ldots, i_k$ such that

$$\sum_{\emptyset \neq u \subset \{i_1, \ldots, i_k\}} V_u^1 \geq (1-\varepsilon) V^p(f).$$

S and T are referred to superposition and truncation dimension respectively, based on the Hardy and Krause (HK) variation of a function.

Assuming that the function we wish to integrate has low effective dimension $k << n$, it has a good approximation by the partial sum of ANOVA-terms depending on k or less variables. Then the integration over the full cube may be reduced to integration over low–dimensional faces of the cube, incurring only a small error. In the literature it has usually been assumed that, for the determination of the effective dimension, the higher order derivatives or at least their order of magnitude can somehow be guessed. Here we consider two methods for evaluating them by automatic differentiation.

2 Direct propagation of cross derivatives

This section provides a description of the direct approach for evaluating cross–derivatives. Given an evaluation procedure for a function that is a succession of elementary operations and intrinsic functions, treat the input variables as linear functions and assign cross–derivatives correspondingly. Thereafter the cross–derivatives are propagated step by step to all intermediate values, and in the end to the function values.

We now only need to be concerned with the propogation through single steps. For that, note that each of the intermediate values is a function of the input variables. Given any two arbitrary functions u and w with their value and all the cross–derivatives, we now have to outline how to propagate cross–derivatives to the results of simple arithmetic operations such as multiplication $u \cdot w$, division u/w etc. Moreover, we need to propagate cross–derivatives through simple elementary functions such as $\exp(u)$, $\sin(u)$, $\sqrt{u}$ and other functions from the `math.h` C–library.

For that purpose, we will suggest a data structure which contains all the cross–derivatives of an arbitrary function, describe rules for initialization, provide equations for further propagations and demonstrate the general approach for constructing such rules. We use C as an informal programming language to specify our algorithms, of course, other implementations are possible.

2.1 Allocation in a cube

Due to the commutativity of partial differentiation, the set of higher order partial derivatives of a given order is highly symmetric. It is still an open problem to organize those derivatives in a data structure providing both efficient access to individual derivatives and containing none of the derivatives twice. However, in the case of the subset of all cross–derivatives, a natural data structure with easy and fast access exists.

This data structure organizes all the 2^n cross–derivatives of a function u in a flat array with 2^n entries. We call such data structure an n–dimensional cube. Each entry of that cube contains one cross–derivative according to the following rule: consider the binary representation of $0 \leqslant k < 2^n$, $k = \sum_{j \in \mathbf{i}} 2^{j-1}$, it has n bits, $\mathbf{i} \subset \{1, 2, \ldots, n\}$ – regard each bit as corresponding to one of the independent variables; we differentiate with respect to variables x_j whose corresponding bits are $1's$, that is where $j \in \mathbf{i}$, and get the k–th entry of the cube. In a geometric interpretation the number k corresponds to the vertex of the hypercube $[0,1]^n$ represented by the same 0–1–pattern, that is $\sum_{j \in \mathbf{i}} e_j$.

With the above binary representation property, we notice that the second half of the cube has the same internal structure as the first half, but differentiation is done also with respect to the last variable. This property allows us to create simple recursive functions for evaluating cross–derivatives.

2.2 Arithmetic operations

For a function u we will denote by `U` its cube. We start with simple *initialization* operations at a given point x. For a constant function $u(x) = c$ we would set `U[0]=c` and all the remaining cube entries are initialized to zero. For a coordinate function resp. input variable $u(x) = x_j$ we would initialize its cube by setting `U[0]=`x_j and `U[`2^j`]=1`, the rest of the entries are set to zero.

For *addition and substraction* operations, $v = u \pm w$, the corresponding propagation rule is: `V[i]=U[i]`$\pm$ `W[i]` for all $0 \leqslant i < 2^n$. And for scalar multiplication $v(x) = cu(x)$ the propagation rule is: `V[i]=c*U[i]`. This stems directly from the linearity of differentiation. A special case of addition (substraction) is that one of the operands is a constant scalar. In this case we do not need to allocate a whole 2^n cube for the scalar then apply the addition propagation, we can just add (substract) this constant from `U[0]` saving storage and runtime. One can easily see that the complexity of the above linear operations is $O(2^n)$.

For any subset $\mathbf{i} \subset \{0, \ldots, n-1\}$, we will denote by $v_\mathbf{i}$ the cross–derivatives obtained by differentiating with respect to variables x_k, $k \in \mathbf{i}$. The generalized Leibniz formula for the *multiplication* of two functions $v = u \cdot w$ then states that:

$$v_\mathbf{i}(x) = \sum_{\mathbf{j} \subseteq \mathbf{i}} u_\mathbf{j}(x) w_{\mathbf{i}-\mathbf{j}}(x) \,.$$

Assume now that $n \notin \mathbf{i}$. Then the above convolution sum can be split as

$$v_{\mathbf{i} \cup \{n\}}(x) = \sum_{\mathbf{j} \subset \mathbf{i}} u_{\mathbf{i}-\mathbf{j}}(x) w_{\mathbf{j} \cup \{n\}}(x) + \sum_{\mathbf{j} \subset \mathbf{i}} u_{\mathbf{j} \cup \{n\}}(x) w_{\mathbf{i}-\mathbf{j}}(x)$$

Note that, fixing the same subset $\mathbf{i}$, all three sums have the same structure. They all operate inside separate halves of cubes. Varying $\mathbf{i} \subset \{1, 2, \ldots, n-1\}$ over the full half–cube results in the reducion of the multiplication of cubes of n variables to 3 multiplications of cubes of $(n-1)$ variables. This leads to the following recursive multiplication procedure:

```
void crossmult (int h, double*u, double*w, double* v) {
   if (h==1) { v[0] += (u[0]*w[0]); return; }
   h /= 2;
   crossmult(h,u,w+h,v+h); crossmult(h,u+h,w,v+h);
   crossmult(h,u,w,v);
}
```

Due to the recursive nature of this procedure, there will be 3^n final function calls with $h = 1$ resulting in 3^n multiplications and the same number of additions.

Another, equivalent version of this function with the same operations count which reduces the number of recursive calls from $1.5 \cdot 3^n$ to 3^n is:

```
void crossmult2 (int h, double* u, double* w, double* v) {
   int i; for (i=h/2; i>0; i/=2) {
      crossmult2 (i,u,w+i,v+i); crossmult2 (i,u+i,w,v+i); }
   v[0]+=u[0]*w[0];
}
```

In Section 4 we will discuss optimizations of the implementation of the multiplication of cubes by using a nonrecursive multiplication algorithm for small cubes.

The remaining arithmetic operations have operations counts that are expressable by the cost of a multiplication. Division is dominated by one multiplication, whereas the cost of computing squares and square roots is one half of the cost of one multiplication.

To perform a *division* of a cube u by a cube w with result $v = u/w$, we consider the Leibniz formula for the equivalent equation $u = w \cdot v$:

$$u_{\mathbf{i}}(x) = \sum_{\mathbf{j}\subseteq\mathbf{i}} w_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j}}(x) = w_{\emptyset} v_{\mathbf{i}}(x) + \sum_{\mathbf{j}\subsetneq\mathbf{i}} w_{\mathbf{i}-\mathbf{j}}(x)\, v_{\mathbf{j}}(x)$$

To extract the highest derivative $v_{\mathbf{i}}$ we divide by $w_{\emptyset}$ and introduce scaled values $\tilde{w}_{\mathbf{j}} = w_{\mathbf{j}}/w_{\emptyset}$ and $\tilde{u}_{\mathbf{j}} = u_{\mathbf{j}}/w_{\emptyset}$ for all subsets $\mathbf{j} \subset \mathbf{i}$. Additionally, we set $\tilde{w}_{\emptyset} = 0$. After addends rearrangement we get:

$$v_{\mathbf{i}}(x) = \tilde{u}_{\mathbf{i}}(x) - \sum_{\mathbf{j}\subset\mathbf{i}} \tilde{w}_{\mathbf{i}-\mathbf{j}}(x)\, v_{\mathbf{j}}(x)$$

This is a recursive formula for the computation of the coefficients $v_{\mathbf{i}}$. To ensure its correctness, the computation of coefficients $v_{\mathbf{j}}$ needs to be finalized for all $\mathbf{j} \subset \mathbf{i}$. In devising an algorithm, this means that of the entire cube, the computation of the first half of v comes first, then it needs to be applied to the second half, and only then the second half acts on itself. The same applies to the recursive computation of the convolution product, which necessitates a separate convolution function named *decremental convolution*.

```
void crossdeconv (int h, double* u, double* w, double* v) {
   v[0]-=u[0]*w[0];
   int i; for (i=1; i<h; i*=2) {
      crossdeconv (i,u+i,w,v+i); crossdeconv (i,u,w+i,v+i); }
}

void crossdivide(int h, double* u, double* w, double* v) {
   int i;   double w0=w[0]; w[0] = 0;
   for(i=0;i<h;i++) { v[i]=u[i]/w0;  w[i] /= w0; }
   for(i=1;i<h;i*=2) {
      crossdeconv(i,w+i,v,v+i); crossdeconv(i,w,v+i,v+i); }
   for(i=1;i<h;i++) { w[i]*=w0; } w[0] = w0;
}
```

The operation count is the same as in a full multiplication, plus $3 \cdot 2^n$ multiplications for the scaling and reconstruction of the arrays. The careful consideration of the order of operations allows to use them also for in–place versions of the arithmetic operations, `u*=v` can be implemented as `crossmult(h,v,u,u)` and `u/=v` as `crossdivide(h,u,v,u)`.

For the *square* function, $v = u^2$, the derivative by the last variable is $v_n = 2uu_n$. Picking some index set $\mathbf{i} \subset \{1, 2, \ldots, n-1\}$, this generalizes to:

$$v_{\mathbf{i}\cup\{n\}} = 2\sum_{\mathbf{j}\subseteq\mathbf{i}} u_{\mathbf{i}-\mathbf{j}}(x) u_{\mathbf{j}\cup\{n\}}(x)$$

So the second half of the resulting cube v is the procuct of the first and second half of the input cube u. The same argument, now in $n-1$ variables, applies recursively to the first half of v An implementation code may be as follows:

```
void square(int h, double* u, double* v) {
   int i; for(i=1;i<h;i++){ v[i] = 0; }
   v[0]=u[0]*u[0];
   for(i=1;i<h;i*=2) { crossmult (i,u,u+i,v+i); }
   for(i=1;i<h;i++) { v[i] *= 2; }
}
```

The $n = \log_2 h$ calls to the multiplication function add up to one half the cost of a full multiplication, since $1+3+\cdots+3^{n-1} = \frac{1}{2}(3^n - 1)$. One would obtain the same result using a call to the multiplication procedure `crossmult(h,u,u,v)` instead, but this would double the effort.

2.3 Nonlinear intrinsic functions

We now provide methods for propagating cross–derivatives of some intrinsic nonlinear functions. The main idea is to utilize the defining differential equation and to derive from it relations between the cross–derivatives of input and result similar to the Leibniz rule in the multiplication case. These relations will be transformed into executable code mainly using the `crossmult()` function and thus inheriting its complexity.

The *exponential* function $v = \exp(u)$ has a very simple identity for the first partial derivatives, $v_k = vu_k$. This generalizes for $k \notin \mathbf{i}$ to:

$$v_{\mathbf{i}\cup\{k\}} = \sum_{\mathbf{j}\subseteq\mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x) u_{\mathbf{j}\cup\{k\}}(x)$$

The second half cube of v is thus obtained by multiplying the previously computed first half cube of v and the second half cube of u.

```
void exponent(int h, double* u, double* v) {
   int i; for(i=0;i<h;i++) { v[i] = 0.0; }
   v[0]=exp(u[0]);
   for(i=1;i<h;i*=2) { crossmult (i,v,u+i,v+i); }
}
```

Again, the $n = \log_2 h$ calls to the multiplication function add up to one half the cost of a full multiplication.

Both *sine* and *cosine* have mutual ODE's, so we would prefer to use this property and to propagate their cubes simultaneously. Denote $v = \sin(u)$, $w = \cos(u)$, then the first partial derivatives obey $v_k = wu_k$ and $w_k = -vu_k$. The generalized mutual differential equations for $k \notin \mathbf{i}$ are:

$$v_{\mathbf{i}\cup\{k\}} = \sum_{\mathbf{j}\subseteq\mathbf{i}} w_{\mathbf{i}-\mathbf{j}}(x)\, u_{\mathbf{j}\cup\{k\}}(x), \qquad w_{\mathbf{i}\cup\{k\}} = -\sum_{\mathbf{j}\subseteq\mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x)\, u_{\mathbf{j}\cup\{k\}}(x)\,.$$

Again, transformation to a code is done by means of cross–multiplication:

```
void trigon(int h, double* u, double* sine, double* cose) {
   int i; for(i=1;i<h;i++) { sine[i] = cose[i] = 0.0; }
   sine[0] = sin(u[0]); cose[0] = cos(u[0]);
```

```
    for(i=1;i<h;i*=2) {
        crossmult (i,cose,u+i,sine+i); crossdeconv (i,sine,u+i,cose+i); }
}
```

Note that the hyperbolic sine and cosine can be calculated the same way (only with a different sign in the 2nd ODE). Also note that the input to the multiplication procedures is well defined (already computed). The number and type of calls corresponds to the semi-iterative `crossmult2()` procedure. Thus the computation of the trigonometric functions has the same complexity as one full multiplication.

2.4 Overall Complexity

While the initialization and linear operations are of $O(2^n)$ complexity, most operations for evaluating cross–derivatives have a complexity similar to the multiplication function `crossmult()`, which is of $O(3^n)$ complexity, or more specifically requires 3^n multiplications, a similar number of additions and $1.5 \cdot 3^{n-b}$ or, in the `crossmult2()` version, 3^n recursive function calls with an recursion depth of n. Tests were carried out on a Pentium 3.0 Ghz machine. The observations are that for 18 variables the computation of one multiplication takes $3.6s$, for 19 variables $10.8s$ and for 20 variables $32.75s$ (note that the time indeed triples for each added variable). With $3 * 10^9$ processor cycles per second, this amounts to about 28 processor cycles per multiplication.

However, the advantage of the direct method can be felt if one can construct an architecture which carries out the cross-multiplication function as one of its basic operations. Anyway, we present in the following chapter another method for propagating cross–derivatives which is of $O(n^2 2^n)$ complexity and can also be applied to any programmed mathematical function.

3 Computation of cross–derivatives via univariate expansions

Consider the task of computing the highest order cross–derivative $f_{\{1,2\}}(x) = \partial_1\partial_2 f(x)$ of a bivariate function $f : \mathbb{R}^2 \to \mathbb{R}$ using only univariate Taylor expansions. From the polarization of quadratic forms we know that this is possible using the pair of directions $\{(1,1), (1,-1)\}$ or the triple $\{(0,1), (1,0), (1,1)\}$. This mixed derivative is obtained as the quadratic term in the Taylor expansion of the linear combinations

$$\tfrac{1}{2} f\big(x+t(1,1)\big) - \tfrac{1}{2} f\big(x+t(1,-1)\big) = t\,\partial_2 f(x) + t^2\,\partial_1\partial_2 f(x) + \ldots$$

or

$$f\big(x+t(1,1)\big) - f\big(x+t(0,1)\big) - f\big(x+t(1,0)\big) = -f(x) + t^2\,\partial_1\partial_2 f(x) + \ldots$$

As we will proceed to show, the second variant is easy to generalize to the computation of higher order cross–derivatives (see [GUW00])

$$f_{\mathbf{i}}(x) = \frac{\partial^k f}{\partial x_{i_1} \ldots \partial x_{i_k}}(x), \qquad i_1 < i_2 < \cdots < i_k\,.$$

In conclusion, to compute the cross–derivatives in this manner we need an automatic differentiation tool like ADOL-C or FADBAD/TADIFF that allows to compute Taylor expansions of a given degree in any direction.

3.1 Propagation of Taylor polynomials

Given a function $f : \mathbb{R}^n \to \mathbb{R}$, a point $x \in \mathbb{R}^n$ and a direction $v \in \mathbb{R}^n$, the task is to compute the Taylor polynomial up to some degree d. We assume again, as is typical for the theory of automatic differentiation, that f is defined as a concatenation of elementary operations, so that the function is at least piecewise analytical and the Taylor polynomials are well defined almost everywhere. Then it is sufficient to explore the propagation of Taylor polynomials through such elementary operations (see [GW08]).

Multiplication is done as truncated polynomial multiplication. Suppose $u = \sum_{k=0}^{d} u_k t^k$ and $w = \sum_{k=0}^{d} w_k t^k$ are given Taylor polynomials, then $v = u * w = \sum_{k=0}^{d} v_k t^k$ has the coefficients

$$v_k = \sum_{\ell=0}^{k} u_\ell w_{k-\ell}, \quad k = 0, 1, \ldots, d\,.$$

The direct implementation of this formula leads to a operations count of $d(d+1)/2$ multiplications and $d(d-1)/2$ additions of coefficients. Since the operations count is dominated by the multiplications, we only trace their number. For sufficiently high degree d one might also use speed–up tricks such as Karatsuba or Toom-Cook multiplication.

Division $v = u/w$ may be considered as the solution to $u = v * w$, setting $\bar{u} = u/w_0$ and $\bar{w} = w/w_0 - 1$, this can be achieved by the formula $v = \bar{u} - v * \bar{w}$, that is,

$$v_k = \frac{1}{w_0}\left(u_k - \sum_{\ell=0}^{k-1} v_\ell w_{k-\ell}\right) = \bar{u}_k + \sum_{\ell=0}^{k-1} v_\ell \bar{w}_{k-\ell}, \quad k = 0, 1, \ldots, d\,.$$

Thus, division is mainly the multiplication of v and $\bar{w}$ and has thus the same complexity.

The computation of the *square root* $v = \sqrt{u}$ is again performed via the solution of the defining equation $v^2 = u$. In a first step, $v_0 = \sqrt{u_0}$ is computed. Set $\bar{u} = u/u_0 - 1$ and $\bar{v} = v/v_0 - 1$, then $\bar{u} = 2\bar{v} + \bar{v}^2$,

$$\begin{aligned}
\bar{v}_{2k-1} &= \tfrac{1}{2}\bar{u}_{2k-1} - \sum_{\ell=1}^{k-1} \bar{v}_\ell \bar{v}_{2k-1-\ell}, && k = 1, 2, \ldots, \lceil d/2 \rceil \\
\bar{v}_{2k} &= \tfrac{1}{2}\left(\bar{u}_{2k} - \bar{v}_k{}^2\right) - \sum_{\ell=1}^{k-1} \bar{v}_\ell \bar{v}_{2k-\ell}, && k = 1, 2, \ldots, \lfloor d/2 \rfloor \\
v_k &= v_0\,\bar{v}_k, && k = 1, 2, \ldots, d\,.
\end{aligned}$$

Exploiting the symmetry, the computation of the square root as well as of the square requires half the complexity of a Taylor multiplication.

The *exponential function* $y(x) = e^x$ is the solution of the differential equation $y'(x) = y(x)$, so that $v = y(u) = e^u$ also satisfies $\dot{v} = y'(u)\dot{u} = v * \dot{u}$ where $\dot{u} = \frac{d}{dt}u = \sum_{k=1}^{d} k u_k t^{k-1}$ etc. are the derivatives by

t. This equation yields a recursive formula for the coefficients starting with $v_0 = e^{u_0}$ and iterating

$$kv_k = \sum_{\ell=0}^{k-1} (\ell u_\ell) v_{k-1-\ell},$$

$$\text{so that} \quad v_k = \frac{1}{k} \sum_{\ell=0}^{k-1} \bar{u}_\ell v_{k-1-\ell}, \quad k = 1, \ldots, d$$

with precomputed $\bar{u}_k = k u_k$. Again the complexity is essentially the one of a Taylor multiplication.

In a similar way one may compute the pair of the *cosine and sine functions*, since they are solutions of a system of differential equations of order one. Their simultaneous evaluation takes the form of two Taylor multiplications, so that their operations count is about d^2 multiplications of coefficients. The Taylor polynomial of an inverse functions essentialy requires the same complexity as the Taylor polynomial of the function itself. All in all, if the evaluation of some given function requires the evaluation of L elementary functions and operations, then the computation of the Taylor polynomial of degree d requires at most Ld^2 floating point multiplications or at most $2Ld^2$ floating point operations.

3.2 Interpolation of all cross–derivatives

Let $f : \mathbb{R}^n \to \mathbb{R}$ be at least n times continuously differentiable. Define for every subset $\mathbf{i} \subset \{1, 2, \ldots, n\}$ the vector

$$e_{\mathbf{i}} = \sum_{m \in \mathbf{i}} e_m \,.$$

Those vectors are the vertices of the n–dimensional hypercube $\{0, 1\}^n$. To each vertex $e_{\mathbf{i}}$ we may now associate

- the Taylor polynomial $T_{e_{\mathbf{i}}}(t)$ of order n in that direction,

$$T_{e_{\mathbf{i}}}(t) = \sum_{k=0}^{n} t^k \sum_{\alpha \in \mathbb{N}^n : |\alpha| = k,\, \mathrm{supp}(\alpha) \subset \mathbf{i}} \frac{\partial^k f}{\partial x^\alpha}(x) \,,$$

- a partial sum of that Taylor polynomial containing all the terms where the multiindex α has exactly $\mathbf{i}$ as support, $\mathrm{supp}\,\alpha = \{m : \alpha_m > 0\} = \mathbf{i}$:

$$p_{\mathbf{i}}(t) = \sum_{\alpha \in \mathbb{N}^n :\, \mathrm{supp}(\alpha) = \mathbf{i}} \frac{t^{|\alpha|}}{\alpha!} \frac{\partial^{|\alpha|} f}{\partial x^\alpha}(x) = \sum_{k=0}^{n} t^k \sum_{\substack{\alpha \in \mathbb{N}^n : |\alpha| = k \\ \mathrm{supp}(\alpha) = \mathbf{i}}} \frac{1}{\alpha!} \frac{\partial^k f}{\partial x^\alpha}(x) + O(t^{n+1})$$

- and finally the cross–derivative

$$f_{\mathbf{i}}(x) = \frac{\partial^{|\mathbf{i}|} f}{\prod_{j \in \mathbf{i}} \partial x_j}(x) \,.$$

These three objects are connected in the following way. The cross–derivative $f_{\mathbf{i}}(x)$ is a coefficient in $p_{\mathbf{i}}(t)$. In fact it is the lowest degree coefficient in $p_{\mathbf{i}}(t)$. The lowest degree in t is $|\mathbf{i}|$, since $\mathrm{supp}(\alpha) = \mathbf{i}$ requires $\alpha_m \geq 1$ for $m \in \mathbf{i}$. This lowest degree is realized by exactly one multiindex α with $\alpha_m = 1$ for

$m \in \mathbf{i}$ and $\alpha_m = 0$ otherwise. This multiindex in turn corresponds to the cross–derivative. As already said, $p_{\mathbf{i}}(t)$ is a partial sum of the Taylor polynomial $T_{\mathbf{i}}(t)$.

For the task at hand we would like to go the opposite direction, start with the Taylor polynomials at all the vertices of the hypercube, extract the partial sums and from them the cross derivatives. The last step is just the extraction of a coefficient of known degree. The first step amounts to the solution of a system of linear equations.

Lemma 3.1 *Let the notations and assumptions be as above. Then $T_{e_{\mathbf{i}}}(t)$ really is the Taylor polynomial in direction $e_{\mathbf{i}}$, that is, $f(x+te_{\mathbf{i}}) = T_{e_{\mathbf{i}}}(t) + O(t^{n+1})$ and*

$$T_{e_{\mathbf{i}}}(t) = \sum_{\mathbf{j}\subset\mathbf{i}} p_{\mathbf{j}}(t) \tag{3.1}$$

and conversely

$$p_{\mathbf{i}}(t) = \sum_{\mathbf{j}\subset\mathbf{i}} (-1)^{|\mathbf{i}\setminus\mathbf{j}|} T_{e_{\mathbf{j}}}(t)\,. \tag{3.2}$$

Proof. The Taylor polynomial of f of degree n in an arbitrary direction v is

$$T_v(t) = \sum_{k=0}^{n} t^k \sum_{\alpha\in\mathbb{N}^n:|\alpha|=k} \frac{\partial^k f}{\partial x^\alpha}(x)\, v^\alpha = f(x+tv) + O(t^{n+1})\,.$$

Specializing to $v = e_{\mathbf{i}}$, the product $v^\alpha = v_1^{\alpha_1} \dots v_n^{\alpha_n}$ is only nonzero if $\operatorname{supp}\alpha = \{m : \alpha_m > 0\} \subset \mathbf{i}$. In this case, $v^\alpha = (e_{\mathbf{i}})^\alpha = 1$. If we now sort the multiindices by their support we arrive at the first expression (3.1), since $p_{\mathbf{j}}(t)$ contains exactly those terms where the multiindex has support $\mathbf{j} \subset \mathbf{i}$.

For the second formula, we find for the right hand side that the alternating sum is equal to

$$\begin{aligned}\sum_{\mathbf{j}\subset\mathbf{i}} (-1)^{|\mathbf{i}\setminus\mathbf{j}|} T_{e_{\mathbf{j}}}(t) &= \sum_{\mathbf{k}\subset\mathbf{j}\subset\mathbf{i}} (-1)^{|\mathbf{i}\setminus\mathbf{j}|} p_{\mathbf{k}}(t) \\ &= \sum_{\mathbf{k}\subset\mathbf{i}} p_{\mathbf{k}}(t) \sum_{\mathbf{j}\subset(\mathbf{i}\setminus\mathbf{k})} (-1)^{|\mathbf{j}|} = p_{\mathbf{i}}(t)\,.\end{aligned}$$

The transformation of the sum in the second factor results from replacing $\mathbf{j}$ by $(\mathbf{i}\setminus\mathbf{j}) \subset \mathbf{i}$. Then the remaining condition $\mathbf{k} \subset (\mathbf{i}\setminus\mathbf{j})$ is equivalent to $\mathbf{j} \subset (\mathbf{i}\setminus\mathbf{k})$, which also implies the first condition.

Now if $\mathbf{k} = \mathbf{i}$ then the only subset is the empty set with an even number of elements, resulting in a factor of 1. If the difference set $(\mathbf{i}\setminus\mathbf{k})$ contains at least one element, then there is a one–to–one correspondence of subsets that contain this element and those that do not. Each corresponding pair results in a pair of -1 and 1 in the sum, which gives a grand total of zero. □

3.3 Efficient transformation of Taylor polynomials to cross–derivatives

The naive way to compute the cross–derivative terms would just evaluate formula (3.2). This would require an iteration over all subsets of subsets $\mathbf{j} \subset \mathbf{i} \subset \{1,\dots,n\}$. Since each index may occur in $\mathbf{i}$ alone, in $\mathbf{i}$ and $\mathbf{j}$ or not at all, this gives 3^n combinations resulting in $3^n - 2^n$ operations.

However, in the course of those computation, certain subexpressions are evaluated repeatedly. Consider again the computation of the full polynomial $p_{\mathbf{i}}(t)$ for some subset $\mathbf{i} \subset \{1,\dots,n\}$. Let $i_1 \in \mathbf{i}$ be the smallest element in $\mathbf{i}$. Then we can separate subsets of $\mathbf{i}$ containing i_1 from those not containing it.

$$\begin{aligned} p_{\mathbf{i}}(t) &= \sum_{i_1 \in \mathbf{j} \subset \mathbf{i}} (-1)^{|\mathbf{i}\setminus\mathbf{j}|} T_{e_{\mathbf{j}}}(t) + \sum_{\mathbf{j} \subset \mathbf{i}\setminus\{i_1\}} (-1)^{|\mathbf{i}\setminus\mathbf{j}|} T_{e_{\mathbf{j}}}(t) \\ &= \sum_{\mathbf{j} \subset \mathbf{k}} (-1)^{|\mathbf{k}\setminus\mathbf{j}|} T_{e_{\mathbf{j}\cup\{i_1\}}}(t) - \sum_{\mathbf{j} \subset \mathbf{k}} (-1)^{|\mathbf{k}\setminus\mathbf{j}|} T_{e_{\mathbf{j}}}(t) \end{aligned}$$

with $\mathbf{k} = \mathbf{i} \setminus \{i_1\}$. The last term appears in the corresponding decomposition of all $p_{\mathbf{k}\cup\{j_1\}}(t)$ where j_1 is smaller than the minimal element of $\mathbf{k}$.

This decomposition can be repeated for the next to smallest index and so on. A systematical computation of these intermediary expressions can be achieved using the identity

$$p_{\mathbf{i}}^{(k)}(t) = \begin{cases} p_{\mathbf{i}}^{(k-1)}(t) - p_{\mathbf{i}\setminus\{k\}}^{(k-1)}(t) & \text{for } k \in \mathbf{i} \\ p_{\mathbf{i}}^{(k-1)}(t) & \text{for } k \notin \mathbf{i} \end{cases}, \quad k = 1,\dots,n,$$

where $p_{\mathbf{i}}^{(0)}(t) = T_{e_{\mathbf{i}}}(t)$ and the cross–derivative $f_{\mathbf{i}}(x)$ occurs as the degree $|\mathbf{i}|$ coefficient of $p_{\mathbf{i}}^{(n)}(t)$ for all $\mathbf{i} \subset \{1,\dots,n\}$.

For a practical implementation we need to be able to enumerate the 2^n subsets of $\{1,2,\dots,n\}$. This is most readily done by identifying the subsets with the numbers $0,1,2,\dots,2^n-1$ via their binary representation, for instance as $b(\mathbf{i}) = \sum_{k\in\mathbf{i}} 2^{k-1}$. Then if some index k is not contained in $\mathbf{i}$ we know that $b(\mathbf{i}\cup\{k\}) = b(\mathbf{i}) + 2^{k-1}$. One possible implementation of the alternating sums is thus

```
int dist=1<<n; while ( dist>1 ){
  int k=1; dist/=2; while ( k<(1<<n) ){
    if ( k mod dist==0 ) k+=dist;
    p[k+dist]-=p[k]; /* overloaded for polynomial subtraction */
    k++; }  }
```

Note that the polynomial operations on $p_{\mathbf{i}}^{(k)}(t)$ may leave out all coefficients of degree smaller than $|\mathbf{i}|$. In total this results in n iterations over 2^{n-1} subsets with an average of $n/2$ modified coefficients, thus a total of $n^2\, 2^{n-2}$ subtractions. This cost occurs only once for one evaluation of the cross–derivatives and is negligible against the complexity of evaluating the Taylor polynomials, provided the evaluation of the function involves a significant number of elementary functions.

3.4 Cross–derivatives of limited order

Computing the full set of cross-derivatives is only practical for up to 32 or possibly 64 variables, corresponding to the addressable memory in a computer. However, applications in quasi-Monte Carlo integration nowadays use several hundreds of variables, but to explore the effective dimension of a function, are only interested in low order cross–derivatives, with orders in the range from 2 to 5 (see [SW98, Kuo03]). For these it is sufficient to also compute the Taylor polynomials only to that degree.

The challenge now is to enumerate the set $\{\mathbf{i} \subset \{1,2,\ldots,n\} : |\mathbf{i}| \leq d\}$ of size

$$K = 1 + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{d} \approx \frac{n^d}{d!}$$

in such a way that allows fast determination if some number k is contained in a subset $\mathbf{i}$, and what the index of $(\mathbf{i} \cup k)$ resp. $(\mathbf{i} \setminus k)$ is. Provided that such functions exist, the algorithms similar to the full evaluation can be employed. The complexity of evaluating the cross–derivatives up to order d can then be given as $d^2 K \approx d^2 n^d / d!$ times the operation count of the evaluated function.

4 Comparison of direct and interpolation approach

4.1 Optimizing the direct multiplication

For the multiplication of small cubes one could generate the multiplication code by hand or use a direct iteration approach based on the correspondence between the bit pattern of an index in a cube and the subset of variables involved in the corresponding cross–derivative. We can use bit manipulation operators to perform set operations such as union and intersection.

```
void shortmult (int h, double* u, double* w, double* v){
   int i,j; v[0]+=w[0]*u[0];
   for ( i=0; i<h; i++ ) for( j=0; j<i; j++ )
          if (i&j==0) /* disjoint subsets contribute to union */
             v[i|j]+=(u[i]*w[j] + u[j]*w[i]);
}
```

This iteration generates a highly nonordered access pattern to elements of the three cubes. From this there results a tradeoff between a reduced number of recursive function calls and an increasing number of cache misses.

Runtime expirements were performed to find the optimal size of a small cube, at which the recursive multiplication procedure would switch to the non–recursive variant. In the experiments we used different sizes of the big input cubes (corresponding to different number of input variables) and for each of them different sizes for the small cube condition. These experiments are summarized in Figure 1. The x-axis denotes the number of variables in a small cube, corresponding to an array size of $h = 2^x$. The y–axis gives a measure for the runtime. To avoid widely differing scales, the runtime was converted into processor cycles per multiplication, thus the actual runtime is $y \cdot 3^n / 3GHz$, with n the number of variables in a big cube.

As one can see, the graphs for different n match very well. In evaluating the graphs, at first, as the variables in a small cube grow to 4, we gain considerably in runtime. Then until 15 variables in a small cube the runtime remains approximately the same. For more than 15 variables the runtime again increases. As a conclusion, in the implementation for all the other experiments we used a hand–coded version of the multiplication of small cubes for 4 variables.

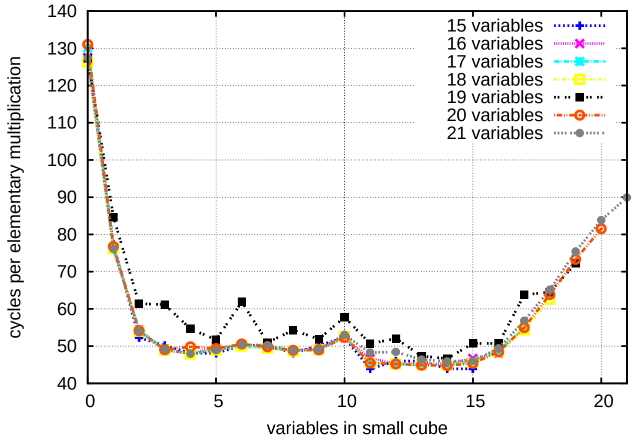

Figure 1: Runtimes for different sizes of small cubes in the direct multiplication

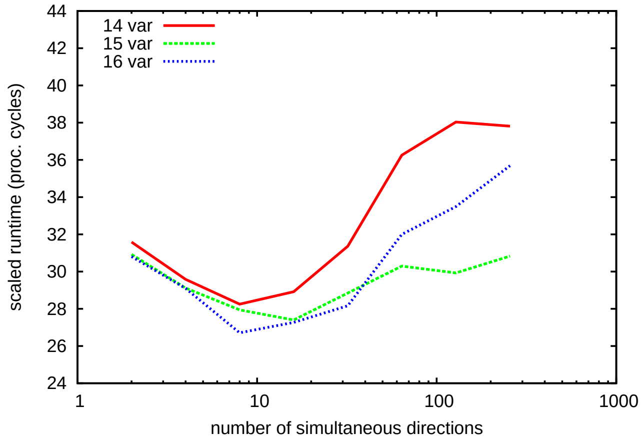

Figure 2: Stripmining of the calls to `hov_forward()`

4.2 Optimizing the Taylor polynomial method

For the implementation of the Taylor arithmetics the ADOL-C automatic differentiation library was used. This library allows with a call to the `hov_forward()` procedure to simultaneously evaluate sevaral univariate Taylor polynomials in different directions. This simultaneous evaluation allows the reuse of the control flow and intermediate function values. Again, there is a trade–off in that the memory blocks for the Taylor polynomials of intermediate values increase in size proportional to their number, but the management of those blocks becomes inefficient with increasing velocity.

This effect can be seen in Figure 2, where a test function with evaluation complexity $O(n)$ was evaluated for its cross–derivatives. Thus the expected runtime is proportional to $n^3 2^n$. This factor was used to normalize the graphs for the different numbers of variables, the actual runtime was accordingly $y \cdot n^3 2^n / 3GHz$. The logarithmic x axis contains the number of simultaneous directions, there is a clear trough from 8 to 16 directions.

4.3 Cross–over

The operation count for the computation of cross–derivatives of a function in n variables increases roughly proportional to 3^n for the direct method and proportional to $n^2 \cdot 2^n$ for the Taylor polynomial method. For large dimensions n, the Taylor method will have better runtimes. However, the direct method uses direct function calls for the elementary operations, whereas the Taylor method via ADOL-C interprets an internal representation of the operation sequence. Therefore, one has to expect some factor larger than one in the Taylor method. As can be seen in Figure 3, assuming a moderate factor between 1 and 2 one read off a cross–over point between $n = 10$ and $n = 16$, which is consistent with the experimental deteminations of cross–over points in Figures 4 and 5. Figure 4 compares runtimes for the computation of the permanent of randomly generated matrices of different sizes using the algorithm of the introduction Section 1.

The set of experiments combined in the diagrams of Figure 5 compare the runtimes of both methods

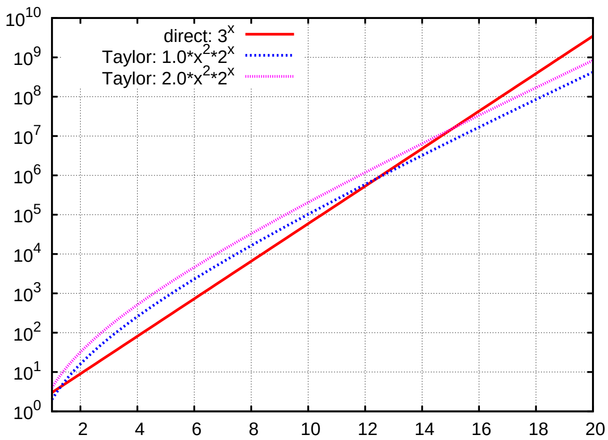

Figure 3: Theoretical runtime multipliers for the direct and the Taylor approch to evaluating cross derivatives

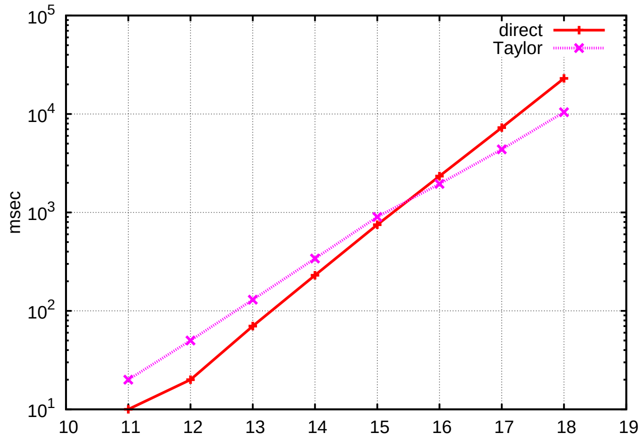

Figure 4: Runtimes for the direct and Taylor methods for the permanent computation

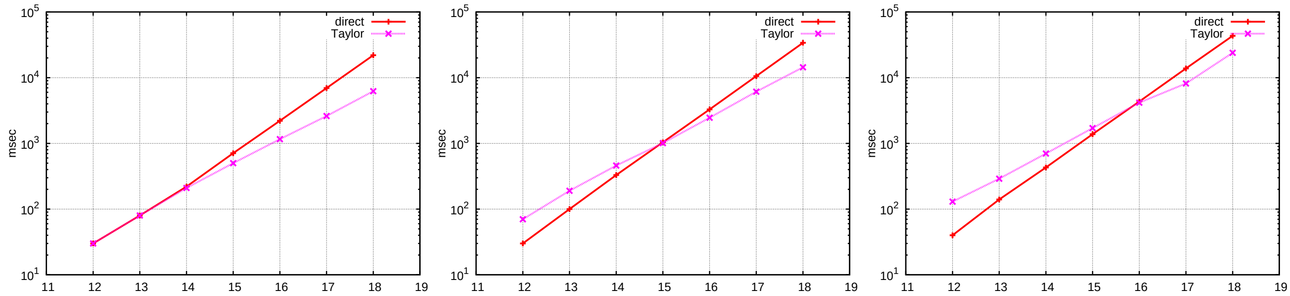

Figure 5: Runtimes for the direct and Taylor methods for three variants of the integration test function

for a family of test functions from the theory of high–dimensional integration,

$$f(x) = \prod_{k=1}^{n} \frac{h_k(x) + a_k}{1 + a_k}, \quad \text{where} \quad h_k(x) = \begin{cases} b_k\,|x_k - c_k| & \text{in test 1} \\ \exp(b_k\,|x_k - c_k|) & \text{in test 2} \\ \exp(b_k^2 (x_k - c_k)^2) & \text{in test 3} \end{cases}$$

where the constants a_k were taken to be $a_k = 1$ for all k; the b_k and c_k sequences as well as the coordinates of the evaluation point x were generated randomly in the $[-1, 1]$ interval.

In both instances, the direct method performed better for up to 14 variables, and the Taylor polynomial method from 15 variables on. However, as we will discuss directly below, the extra speed of the Taylor method resulted in a significant loss in accuracy.

4.4 Numerical Accuracy

In the second method the interpolation of the cross–derivatives from the Taylor polynomials by alternating sums involves the cancellation of unwanted higher order derivatives containing differentiations w.r.t. some variable in multiplicities higher than one. It may happen that those unwanted derivatives

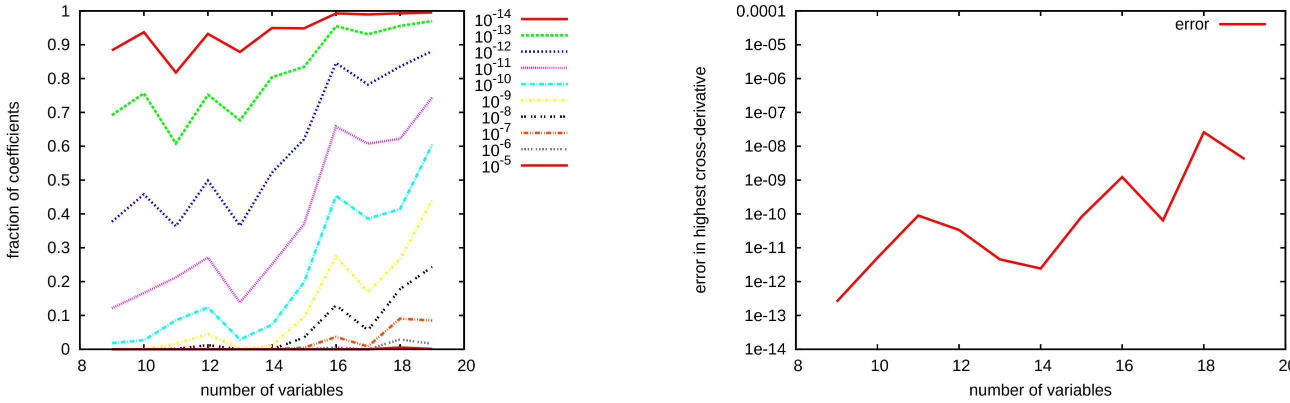

Figure 6: Errors while evaluating the third test function using the Taylor method

have values that are orders of magnitudes bigger than the cross–derivatives, leading to rounding errors that influence the values of the cross–derivatives. That such effects have to be taken into account is shown in Figure 6, where the left diagram shows the fraction of all 2^n cross–derivatives below different error thresholds ranging from 10^{-5} to 10^{-14}, and the right diagram shows the error of the highest order cross–derivative alone. Since the product structure of the function leads to a corresponding product structure in the cross–derivatives, and this structure is exactly captured by the evaluation via the direct method, we may assume for this test case that the results of the direct method are exact within the machine precision. The errors are computed against these values.

5 Summary and conclusion

We have shown how to implement the computation of the full set of cross–derivatives in an automatic differentiation setting in two different ways, a direct method that propagates a suitable data type containing all cross–derivatives along the chain of elementary operations representing the function, and a second method propagating much smaller Taylor polynomials along this chain and recombining the cross–derivatives from Taylor polynomials in as many directions.

We concluded from experiments that the more we go beyond 15 input variables the more the Taylor polynomial method is faster albait less accurate. This is in line with a theoretical estimate of the runtime development that puts the cross–over at 10 variables but is sensitive to changes in the multiplicative constants.

Since the memory requirements of the data types, and also the runtimes grow exponentially, both approaches are limited in their applicability. Future research has to provide methods, based on the presented approaches, to evaluate all cross–derivarives up to some order that is small relative to the number of variables, where the number of variables is in the hundreds or even thousands.

6 Appendix

6.1 Implementation of other math.h functions in the direct method

For the computation of the cube of the *natural logarithm* $v = ln(u)$ of a given cube u, we note the relation of first order derivatives $uv_k = u_k$ and generalize it to the higher order case:

$$\sum_{\mathbf{j} \subseteq \mathbf{i}} u_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x) = u_{\mathbf{i} \cup k}(x)$$

Afterwards we apply the same steps as in the division case (except that here we normalize by u_ϕ):

$$u_\phi v_{\mathbf{i} \cup k} + \sum_{\mathbf{i} \neq \mathbf{j} \subset \mathbf{i}} u_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x) = u_{\mathbf{i} \cup k}(x)$$

$$v_{\mathbf{i} \cup k} = \tilde{u}_{\mathbf{i} \cup k} - \sum_{\mathbf{j} \subseteq \mathbf{i}} \tilde{u}_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x)$$

Note that as in the division case we use already computed part of V's cube to propagate $v_{\mathbf{i} \cup k}$. An implementation of this is:

```
void naturalog(int h, double* u, double* v) {
   int i;  double u0=u[0]; u[0] = 0;
   v[0]=log(u0); for(i=1;i<h;i++) { v[i]=u[i]/=u0; }
   for(i=1;i<h;i*=2) { crossdeconv(i,u,v+i,v+i); }
   u[0] = u0; for(i=1;i<h;i++) { u[i] *= u0; }
}
```

Consider the *power* function $v = u^c$, where $c \in \mathbb{R}$. Differentiating w.r.t one variable yields: $v_k = cu^{c-1}u_k$, after multiplying both sides of the equation by u we get: $uv_k = cvu_k$. Generalizing to the higher order case and using the same steps as in the division and natural logarithm cases, we obtain following equations:

$$\sum_{\mathbf{j} \subseteq \mathbf{i}} u_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x) = c \sum_{\mathbf{j} \subseteq \mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x) u_{\mathbf{j} \cup k}(x)$$

$$u_\phi v_{\mathbf{i} \cup k} + \sum_{\mathbf{i} \neq \mathbf{j} \subset \mathbf{i}} u_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x) = c \sum_{\mathbf{j} \subseteq \mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x) u_{\mathbf{j} \cup k}(x)$$

$$v_{\mathbf{i} \cup k} = c \sum_{\mathbf{j} \subseteq \mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x) \tilde{u}_{\mathbf{j} \cup k}(x) - \sum_{\mathbf{j} \subseteq \mathbf{i}} \tilde{u}_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x)$$

A code implementing the above equations is:

```
void power(int h, double r, double* u, double* v) {
   int i,j; double u0=u[0]; u[0]=0;
   v[0] = pow(u0,r); for(j=1;j<h;j++) { u[j]/=u0; v[j]=0; }
   for(i=1;i<h;i*=2) {
         crossmult(i,v,u+i,v+i);
         for(j=i;j<2*i;j++) { v[j] *= r; }
         crossdeconv(i,u,v+i,v+i); }
   u[0] = u0; for(i=1;i<h;i++) { u[i]*=u0; }
}
```

The *square root* function $v = \sqrt{u}$ is a special case of the power function with $c = 0.5$. To obtain the ODE, We can substitute $u^{1/2}$ into the ODE of the power function, or derive it from scratch by differentiating the square root: $vv_k = 0.5u_k$. Generalization gives:

$$\sum_{\mathbf{j} \subseteq \mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x) = 0.5 u_{\mathbf{i} \cup k}(x)$$

$$v_\phi v_{\mathbf{i} \cup k} + \sum_{\mathbf{i} \neq \mathbf{j} \subset \mathbf{i}} v_{\mathbf{i}-\mathbf{j}}(x) v_{\mathbf{j} \cup k}(x) = 0.5 u_{\mathbf{i} \cup k}(x)$$

We normalize the last equation by $v_\phi^2 (= u_\phi)$ and denote: $\tilde{v}_{\mathbf{j}} = v_{\mathbf{j}}/v_\phi$ and $\tilde{u}_{\mathbf{j}} = u_{\mathbf{j}}/u_\phi$. For the convenience of the computation we also set $\tilde{v}_\phi = 0$, and finally we get:

$$\tilde{v}_{\mathbf{i} \cup k} = 0.5 \tilde{u}_{\mathbf{i} \cup k(x)} - \sum_{\mathbf{j} \subseteq \mathbf{i}} \tilde{v}_{\mathbf{i}-\mathbf{j}}(x) \tilde{v}_{\mathbf{j} \cup k}(x)$$

The implementation code is:

```
void squaroot(int h, double* u, double* v) {
   int i; v[0]=0; for(i=1;i<h;i++) { v[i] = 0.5*u[i]/u[0]; }
   for(i=1;i<h;i*=2) { crossdeconv(i,v,v+i,v+i); }
   v[0]=sqrt(u[0]); for(i=1;i<h;i++) {v[i]*=v[0];}
}
```

The *general power function* is a power function which is of the form $v = u^w$ where u and w are both arbitrary functions. In this case we can try to operate with its ODE, however the ODE is not such simple to derive the generalized direct version and to implement it by means of `crossmult()`. Therefore, we would prefer to regard the general power as a composition of already known functions for which we have already derived propagation rules.

First, we apply the natural logarithm to both sides of the equation and obtain $\ln(v) = w \cdot \ln(u)$, now we can propagate cross–derivatives of $\ln(u)$, then propagate cross–derivatives of the multiplication $w \cdot ln(u)$ and finally of the exponential $v = \exp(w \cdot \ln(u))$.

This is demonstrated in the following code:

```
void raisepow (int h, double* u, double* w, double* v)
{
   int i; double *arr1, *arr2;
   arr1 = (double*) calloc(h,sizeof(double)); // initialized with 0's
   arr2 = (double*) calloc(h,sizeof(double)); // initialized with 0's
   naturalog (h,u,arr1); crossmult(h,w,arr1,arr2);
   exponent (h,arr2,v);
   free (arr1); free (arr2);
}
```

References

[GJ79] M. R. Garey and D. S. Johnson. *Computer and Intractability. A Guide to the Theory of NP–Completeness*. Freeman & Co., New York, 1979.

[GT81] Andreas Griewank and Philippe L. Toint. On the unconstrained optimization of partially separable functions. In M.J.D. Powell, editor, *Nonlinear Optimization*, pages 301–312. Academic Press, London, 1981.

[GUW00] Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Math. Comput.*, 69(231):1117–1130, 2000.

[GW08] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[Hla61] E. Hlawka. Funktionen von beschränkter variation in der theorie der gleichverteilung. *Ann. Mat. Pura Appl.*, 54(4):325–333, 1961.

[KS05] Frances Y. Kuo and Ian H. Sloan. Lifting the curse of dimensionality. *Notices of the AMS*, 52:1320–1329, 2005.

[Kub08] Koichi Kubota. Combinatorial computation with automatic differentiation. In Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and Jean Utke, editors, *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*, pages 315–325. Springer, Berlin, 2008.

[Kuo03] F.Y. Kuo. Component–by–component constructions achieve the optimal rate of convergence for multivariate integration in weighted Korobov and Sobolev spaces. *J. Complexity*, 19(3):301–320, 2003.

[Rys63] H. J. Ryser. *Combinatorial Mathematics*. Number 14 in Carus Mathematical Monographs. Mathematical Association of America, 1963.

[Sob01] Ilya M. Sobol'. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Math. Comput. Simul.*, 55(1–3):271–280, 2001.

[SW98] Ian H. Sloan and Henryk Woźniakowski. When are quasi–Monte Carlo algorithms efficient for high dimensional integrals? *J. Complex.*, 14(1):1–33, 1998.

[WF03] Xiaoqun Wang and Kai-Tai Fang. The effective dimension and quasi–Monte Carlo integration. *J. Complex.*, 19(2):101–124, 2003.

[Zar68] S. C. Zaremba. Some applications of multidimensional integration by parts. *Ann. Polon. Math.*, 21:85–96, 1968.