AUTOMATIC EVALUATIONS OF CROSS-DERIVATIVES

ANDREAS GRIEWANK, LUTZ LEHMANN, HERNAN LEOVEY, AND MARAT ZILBERMAN

Abstract. Cross-derivatives are mixed partial derivatives involving at most one differentiation in each one of n coordinate directions. They are a computational tool in combinatorics and of potential use in high-dimensional integration. Here we present two methods that evaluate all $2^n$ cross-derivatives at a given point. The computational complexity is, respectively, $3^n$ and $n^22^n$ times that of the underlying function. The asymptotically faster method involves a final interpolation step, which can easily be carried out using extra-accurate subtractions to reduce the effect of numerical round-off. Further complexity reductions for large $n$ can be obtained through faster polynomial multiplications, e.g., Karatsuba’s method or FFT.

1. Introduction and motivation

Throughout this paper the term cross-derivatives will refer to the mixed partial derivatives obtained by differentiating w.r.t. each variable at most once. If $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is an $n$ times continuously differentiable function, we can associate with each subset $\mathbf{i} \subseteq D = \{1, 2, \ldots, n\}$ of size $k = |\mathbf{i}|$ the cross-derivative $f_{\mathbf{i}}$ where the partial derivatives are w.r.t. the variables $x_i$ with index $i \in \mathbf{i}$, that is,

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

The cross-derivative corresponding to the empty set is defined to be the original function, $f_{\emptyset}(x) = f(x)$. Just as there are $2^n$ subsets of $D$, there also are $2^n$ different cross-derivatives.

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 [GW08b]. If $u$ and $w$ are functions in $n$ variables that appear as an 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 their partial derivatives have a structure that allows the use of fast algorithms.

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 $O(n^{2n})$ per elementary operation. However, an extra-accurate addition should be used in order to achieve accuracy that is comparable to that of the direct method. In the remainder of the introduction we discuss selected applications.

1.1. Cross-derivatives applied to combinatorics. We present here examples which demonstrate 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 whose associated decision problems are NP-complete. The existence of polynomial time evaluation algorithms would imply that P=NP. Thus we can expect a runtime growing exponentially with the problem size for any algorithm computing this solution. The best available algorithm to compute the permanent has indeed a complexity of $O(n^{2n})$ arithmetic operations for $(n \times n)$-matrices [Rys63].

Kubota presents in [Kub08] transformations of both problems to the determination of certain coefficients of suitably constructed multivariate polynomials. These coefficients belong to the maximally mixed terms of degree $n$ polynomials in $n$ variables. Thus, they are obtainable as highest order cross-derivatives. Kubota evaluates them using a transformation to high-degree univariate polynomials and using FFT based multiplication for their evaluation to arrive at operation counts of order $O(n^{32n})$ for the permanent and $O(n^{42n})$ for the Hamiltonian cycles. Using our second method to evaluate the full set of cross-derivatives we arrive at similar complexities that differ only in the power of $n$. This method turns out to implicitly execute the fast Möbius transform also known as the subset convolution method (see [BHKK07]).

1.1.1. Hamiltonian cycles. Given a simple directed graph $G = (V, E)$, $E \subseteq V \times V$, a Hamiltonian cycle is a cycle of edges in $E$ that visits each vertex in $V$ exactly once (except the initial and final vertex which are visited twice). To count the total number of Hamiltonian cycles of graph $V$ by evaluation of cross-derivatives, we first define the adjacency matrix $A$ representing 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 the matrix

$$H(x_1, x_2, \ldots, x_n) = (X \cdot A)^n \quad \text{with} \quad X \equiv \text{diag}(x_1, x_2, \ldots, x_n).$$

Note that $(X \cdot A)^k$ is nonzero at position $(i, j)$ if there exists 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 $v_i = v_{i_1} \rightarrow v_{i_2} \rightarrow \cdots \rightarrow v_{i_k} = v_j$, then the monomial $x_{i_1} x_{i_2} \ldots x_{i_{k-1}}$ will be present. In consequence, each Hamiltonian cycle will contribute a monomial $x_1 x_2 \ldots x_n$ 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 $h_D$ of the first diagonal entry $h \equiv H_{11}$ of $H$.

1.1.2. Computation of the permanent of a matrix. Given a matrix $A$, its permanent is defined as:

$$\text{per}(A) = \sum_{\sigma \in S^n} a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)}$$
where \( \sigma \in S^n \) runs over all the permutations of \( \{1, 2, \ldots, n\} \). One way to calculate the permanent is to define an \( n \)-variate polynomial \( f \):

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

and to compute its \( n \)-th order cross-derivative \( f_D(x) = \text{per}(A) \).

1.2. High-dimensional Gaussian integration. In practical applications such as mathematical finance or statistical physics one often encounters high-dimensional integrals of the form

\[
I(f) = \frac{1}{(2\pi)^{n/2} \sqrt{\det(C)}} \int_{\mathbb{R}^n} f(x) e^{-\frac{1}{2} x^\top C^{-1} x} dx, \quad x = (x_1, \ldots, x_n),
\]

where \( C \) denotes the covariance matrix of the Gaussian density function. The integrand \( f \) can be smooth (as is usually the case in statistical physics) or present a kink due possibly to a \( \max(\cdot) \) function (as is usually the case in option valuation problems). These integration problems result from discretizations of path integrals and can be effectively approximated by the use of quasi-Monte Carlo methods (QMC) or Sparse Grids (SG) (see [Gri06]). Examples of QMC methods are digital nets (see [Nie92] and [DP10]) and modern lattice rules (see [KS05]).

Normally, a transformation to the unit cube is applied such that the problem takes the form

\[
I(f) = \int_{[0,1]^n} f(A\Phi^{-1}(z)) dz, \quad z = (z_1, \ldots, z_n).
\]

Here \( AA^\top = C \) is some factorization of the covariance matrix, and \( \Phi^{-1}(z) := (\Phi^{-1}(z_1), \ldots, \Phi^{-1}(z_n))^\top \), where \( \Phi^{-1}(\cdot) \) is the inverse of the normal cumulative distribution function \( \Phi(\cdot) \).

QMC methods are equal-weight quadrature rules of the form

\[
Q_{N,n}(f) = \frac{1}{N} \sum_{i=0}^{N-1} f(z_i),
\]

where \( z_0, \ldots, z_{N-1} \) are carefully selected points in \([0,1]^n\). The modern setting of QMC error analysis considers different weighted function spaces. A general development of the theory can be found in [Hic98], [NW08], [NW10]. These weighted spaces are reproducing kernel Hilbert spaces. They are often taken to be tensor product spaces of univariate functions, although spaces with generalized kernels are also of interest. Examples of these are the weighted anchored and unanchored Sobolev spaces with product or generalized weights, and the weighted Korobov spaces.

In practice one is interested in finding optimal weights in product form that correspond to a given fixed function at hand, to embed the function into the appropriate space for integration. Usually the functions considered in practical applications do not lie in these weighted spaces. This is due to strong divergence of their derivatives close to the borders of the unit cube, or due to possible kinks generated by \( \max(\cdot) \) functions (like in the case of option valuation), which reduces the smoothness of the function to the Lipschitz case; see [GKS10]. It is widely believed that the success of QMC for these kinds of integrands from mathematical finance lies in the fact that functions in these applications have low effective dimension in the truncation
or superposition sense (see [WF03]), and that QMC yields better low dimensional projections than Monte Carlo (MC). It is shown in [GKS10] that a truncation of the ANOVA (analysis of variance) decomposition of such functions up to their effective part, that is, the terms accumulating most of the variance up to some predefined accuracy level (usually 99%), may result in a much smoother function than the original one. This effect is especially observable in the case involving max(·) functions.

In addition, the problem of strong divergence of derivatives close to the borders of the unit cube can be overcome by a truncation of the domain. The result is a function belonging to one of the considered weighted reproducing kernel Hilbert spaces. For those it has been shown that the integration problem is tractable under a mild decay condition on the weights (see [NW08]).

It is therefore reasonable to expect that for fixing weights one needs to consider only the effective part $f_{\text{EFF}}$ of the function $f$ composed of the dominant ANOVA terms, possibly after a truncation in the domain. Here we are interested in functions exhibiting low effective dimension $d$ in the truncation sense, so we define

\[ f_{\text{EFF}}(x) \equiv \sum_{I \subseteq \{1, \ldots, d\}} f^I(x) \approx f(x) \]

with $f(x) = \sum_{I \subseteq \{1, \ldots, n\}} f^I(x)$ the full ANOVA decomposition of the function $f$ (see [WF03]). The quadrature error of $f$ by a QMC method takes the form

\[ |(I - Q_{N,n})(f)| \leq |(I - Q_{N,n})(f_{\text{EFF}})| + |(I - Q_{N,n})\left(\sum_{I \subseteq \{1, \ldots, n\} \setminus I \subseteq \{1, \ldots, d\}} f^I(x)\right)|. \]

Now we assume that for the purpose of fixing weights on practical problems, the error contributions not belonging to $f_{\text{EFF}}$ are small enough to be neglected.

A way of choosing generalized weights $\gamma_{n,1}$ for a fixed function $g$ belonging to the weighted unanchored Sobolev space was proposed by Sloan and Wang in [WS07]. Their approach is to choose weights that minimize the bound obtained by the average worst case error on the unit ball of the weighted space times the semi-norm of the function at hand,

\begin{equation}
|I - Q_{N,n})(g)| \leq \frac{1}{N^{\frac{1}{2}}} \left( \sum_{\emptyset \neq I \subseteq \{1, \ldots, n\}} (\gamma_{n,1})^{-\frac{1}{2}} \right) \left\| g \right\|_{n,\gamma,3}^* \end{equation}

with the unanchored norm $\left\| \cdot \right\|_{n,\gamma,3}$ defined as

\begin{equation}
\left\| g \right\|_{n,\gamma,3} := \left( \sum_{I \subseteq D} \gamma_{n,1}^{-1} \int_{[0,1]^{|I|}} \left( \int_{[0,1]^{n-|I|}} \frac{\partial |I|}{\partial x_1} g(x_1, x_{D-1}) dx_{D-1} \right)^2 dx_1 \right)^{\frac{1}{2}},
\end{equation}

with $D = \{1, \ldots, n\}$, and $\left\| \cdot \right\|_{n,\gamma,3}^*$ defined as the semi-norm obtained by erasing the term corresponding to $I = \emptyset$ of the norm. This bound is valid because one can always apply the quadrature error bound to $g - I(g)$ instead. An immediate consequence of the Hölder inequality is that

\begin{equation}
\left\| g \right\|_{n,\gamma,3}^* \leq \left( \sum_{\emptyset \neq I \subseteq D} \gamma_{n,1}^{-1} \int_{[0,1]^n} \left( \frac{\partial |I|}{\partial x_1} g(x) \right)^2 dx \right)^{\frac{1}{2}}.
\end{equation}
For a given $g$ these integrals are a little cheaper to approximate than the original ones. As stated above, for practical applications we would like to find weights that minimize the bound (1.4) for $g = f_{\text{EFF}}$.

Weights in product form $\gamma_{n,i} := \prod_{j \in i} \gamma_{n,j}$ are of special interest in applications because they are required for fast component by component constructions of rank-1 lattice rules (see [NC06] and [Nuy07]). In practical cases, one could proceed by calculating generalized weights and then trying to construct product weights from them, because the optimal generalized weights can be explicitly given (see [WS07]). Another option is to deal directly with product weights and solve (maybe only numerically) the related nonlinear optimization problem. This issue is a subject of current research by the authors. In any case, the dominant cost seems to be the computations of the integrals involved in the norm of the function.

A direct application of the new method for calculating cross-derivatives appears in the case when the original function $f$ is smooth. This is the case in the so-called Asian options valuation problem if we assume a strike price $K = 0$, even more with a domain truncation as in [GKS10]. In the case of arithmetic average Asian options with the Brownian bridge construction, it is known from [WF03] that the effective dimension in truncation sense does not exceed the value $d = 8$, independently of the real dimension $n$. In general, if we have a Gaussian integral with a smooth $f$ that exhibits low effective dimension in truncation sense, then the estimate (1.4) for $g = f_{\text{EFF}}$ can be further bounded to obtain

\begin{equation}
(1.7) \quad |(I - Q_{N,n})(f_{\text{EFF}})| \leq \frac{1}{N^{\frac{1}{2}}} \left( \sum_{\emptyset \neq i \subseteq D} (\gamma_{n,i}) 6^{-|i|} \right)^{\frac{1}{2}} \cdot \left( \sum_{\emptyset \neq i \subseteq \{1,\ldots,d\}} \gamma_{n,i}^{-1} \int_{[0,1]^n} \left( \frac{\partial |i|}{\partial x_i} f(x) \right)^2 dx \right)^{\frac{1}{2}}.
\end{equation}

The latter bound follows by considering a similar bounding of the semi-norm $\|f_{\text{EFF}}\|_{n,\gamma,3}^*$ as in (1.6), using properties of the ANOVA decomposition and Hölder’s inequality. Note that in the right-hand side of this inequality we wrote $f$ instead of $f_{\text{EFF}}$. This is valid if the original function $f$ is smooth enough, say $f \in W_{2,\gamma}^{1,\ldots,1}(\{0,1\}^n)$. A derivation of the latter bound can be found for example in [LG12]. We expect that the optimization of this bound w.r.t. generalized or product weights will also be efficient for fixing weights in practical problems.

Here we can apply directly the new technique for computing cross-derivatives. We demonstrate it for the efficient simultaneous calculation of the mixed derivatives involved in the integrals of the proposed bound.

The most expensive part of the function evaluation are the $n$ evaluations of the inverse of the normal cumulative distribution function $\Phi^{-1}$. These intermediates $\Phi^{-1}$ always enter as univariate functions in the evaluation of $f$ in this kind of transformation of Gaussian integrands to $[0,1]^n$. In Section 4.5 we consider the example of simplified arithmetic average Asian options with Brownian bridge construction (without a kink). It will be shown that evaluating simultaneously all first mixed derivatives corresponding to $i \subseteq \{1,\ldots,d\}$ with $d = 8$ incurs a cost of at most 37 function evaluations of the original function, for the different real dimensions $n = 8, 16, 32, 64, 128, 256, 360, 512$. 

License or copyright restrictions may apply to redistribution; see https://www.ams.org/journal-terms-of-use
If the given transformed integrand has a particular evaluation procedure structure, as in the case of the standard construction of arithmetic average Asian options, we can expect that this relative cost for fixed $d$ decreases as $n$ increases, as will be seen in 4.5.

2. Direct propagation of cross-derivatives

This section provides a description of the direct approach for evaluating cross-derivatives. For this, we assume as given an evaluation procedure for a function that is a composition of elementary operations and intrinsic functions. The initializations of the independent variables are considered as linear functions and cross-derivatives are assigned accordingly. Thereafter, the cross-derivatives are propagated step by step to all intermediate values, and finally to the dependent variables representing function values.

We now only need to be concerned with the propagation 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. Arrays in C and allocation in a cube. Arrays of double precision numbers are handled by pointer variables of type (double *). If $u$ is such a pointer for an array of allocated length $h$, then $u[0]$, $u[1]$, ..., $u[h-1]$ are the elements of the array. Another feature of pointers in C is the so-called pointer arithmetics. If $k$ is an integer, then $(u+k)$ is a pointer to the subarray starting at $u[k]$, that is, $(u+k)[i]$ is the same as $u[k+i]$. This gives an easy way to split an array $u$ of even length $2h$ in the middle. One only has to note that $h$ is the length of both subarrays and use the pointers $u$ for the first half and $(u+h)$ for the second half.

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 $u(x)$ in a flat array $u[]$ with $2^n$ entries. We call such data structure an $n$-dimensional cube with reference to the set $\{0,1\}^n$ of the $n$-dimensional unit cube $[0,1]^n$. Each corner $c$ corresponds uniquely to some subset $i \subseteq \{1,2,\ldots,n\}$ where $i \in i$ if and only if $c_i = 1$. Each corner and its associated subset can be numbered within $0,1,\ldots,2^n-1$ by interpreting the coordinates of the corner (in reverse order) as a binary number, i.e., by the allocation function $\text{ind}(i) \equiv \sum_{i \in i} 2^{i-1}$.

This arrangement renders the split in the middle of the cube array into a meaningful operation in the context of the “divide and conquer” principle. The second
half of the cube array has the same internal structure as the first half, except that the corresponding directions of differentiation also include the last one. This property allows us to create simple recursive functions for evaluating cross-derivatives.

2.2. Polynomial operations. For a function \( u(x) \) 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 set \( u[0] = c \) and all the remaining cube entries are initialized to zero. For an input variable \( u(x) = x_j, j = 1, 2, \ldots, n \), interpreted as coordinate function, we initialize its cube by setting \( u[0] = x_j \) and \( u[1 << (j-1)] = 1 \) (where \( 1 << k \) produces the value \( 2^k \)), the remaining entries are set to zero.

For linear operations of the form \( v(x) = u(x) + c w(x) \) with \( c \in \mathbb{R} \) the corresponding propagation rule is: \( v[i] = u[i] + c \cdot w[i] \) for all \( i = 0, 1, \ldots, 2^n - 1 \) with obvious simplifications for addition of subtraction, i.e., \( c \in \{ -1, 1 \} \) and incremental operations where \( u \) and \( v \) point to the same array. In all these linear cases the complexity of terms of scalar arithmetic operations and (very regular) memory accesses is \( O(2^n) \).

The generalized Leibniz formula for the multiplication of two functions \( v(x) = u(x) \cdot w(x) \) states that

\[
v_i(x) = \sum_{j \subseteq i} u_j(x)w_{i-j}(x).
\]

Assume now that \( n \notin i \). Then the complementary cross-derivative containing derivation by \( x_n \) can be split as

\[
v_{i \cup \{n\}}(x) = \sum_{j \subseteq i} u_{i-j}(x)w_{j \cup \{n\}}(x) + \sum_{j \subseteq i} u_{j \cup \{n\}}(x)w_{i-j}(x).
\]

Note that, fixing the same subset \( i \), all three sums have the same structure. They all operate inside separate halves of cubes. Varying \( i \subseteq \{1, 2, \ldots, n-1\} \) over the full half-cube results in the reduction 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:

```c
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+v,v+h); crossmult(h,u+v,w,v+h);
    crossmult(h,u,v,v);
}
```

Due to the recursive nature of this procedure, there will be \( 3^n \) function calls with \( h = 1 \) resulting in \( 3^n \) multiplications and the same number of additions. A version of this routine performing the same arithmetic operations but reducing the total number of recursive function calls from \( 1.5 \cdot 3^n \) to \( 3^n \) is given by

```c
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 non-recursive multiplication algorithm for small cubes.

The remaining arithmetic operations have operation counts that are proportional to the cost of one 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 \( v(x) = u(x)/w(x) \), we consider the Leibniz formula for the equivalent equation \( u(x) = w(x) \cdot v(x) \):

\[
u_1(x) = \sum_{J\subseteq I} w_{I-J}(x)v_J(x) = w_\emptyset v_1(x) + \sum_{J\subseteq I} w_{I-J}(x)v_J(x).
\]

To extract the highest derivative \( v_1 \) we divide by \( w_\emptyset \) and introduce scaled values \( \tilde{w}_J = w_J/w_\emptyset \) and \( \tilde{u}_J = u_J/w_\emptyset \) for all subsets \( J \subseteq I \). Setting \( \tilde{w}_\emptyset = 0 \) we then get

\[
u_1(x) = \tilde{u}_I(x) - \sum_{J\subseteq I} \tilde{w}_{I-J}(x)v_J(x).
\]

This is a recursive formula for the computation of the coefficients \( v_1 \). To ensure its correctness, the computation of coefficients \( v_1 \) must have been completed for all \( J \subseteq I \). In devising an algorithm, this means that the computation of the first half of \( v_I \) from the first halves of \( u_I \) and \( w_I \) comes first, then it needs to be applied to the second half, and only then the second half acts on itself. The computation involving the first halves could be performed by a recursive call to the division function, or the calls from these recursive calls can be collected in a for-loop, as seen before in the multiplication. The same applies to the recursive computation of the convolution product, which necessitates a separate convolution function named **decremental convolution**.

```c
void crossdeconv (int h, double* u, double* w, double* v) {
    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); 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 also allows to use them 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, \( v(x) = u(x)^2 \), the derivative by the last variable is \( v_n = 2uu_n \). Picking some index set \( I \subseteq \{1,2,\ldots,n-1\} \), this generalizes to

\[
v_{I\cup\{n\}} = 2\sum_{J\subseteq I} u_{I-J}(x)u_{I\cup\{n\}}(x).
\]

So the second half of the resulting cube \( v_I \) is the product of the first and second half of the input cube \( u_I \). The same argument, now in \( n-1 \) variables, applies recursively to the first half of \( v_I \), so that
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 \texttt{crossmult}(h,u,u,v)\) instead, but this would double the effort.

2.3. Transcendental functions. We now provide methods for propagating cross-derivatives of some intrinsic nonlinear functions. As customary for univariate Taylor series we may exploit the defining differential equation 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 \texttt{crossmult()} function and thus inheriting its complexity.

The \textit{exponential} function \( v(x) = \exp(u(x)) \) has a very simple identity for the first partial derivatives, namely \( v_k = vu_k \). It generalizes for \( k \not\in i \) to

\[
 v_{i \cup \{k\}} = \sum_{j \subseteq i} v_{i - j}(x)u_{j \cup \{k\}}(x).
\]

The second half cube \( v[] \) is thus obtained by multiplying the previously computed first half cube of \( v[] \) and the second half cube of \( u[] \).

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

Both \textit{sine} and \textit{cosine} satisfy mutual ODE’s, so we can use this property and to propagate their cubes simultaneously. Denoting \( v(x) = \sin(u(x)) \) and \( w(x) = \cos(u(x)) \) we find that the first partial derivatives obey \( v_k = wu_k \) and \( w_k = -vu_k \). The generalized mutual differential equations for \( k \not\in i \) are

\[
 v_{i \cup \{k\}} = \sum_{j \subseteq i} w_{i - j}(x)u_{j \cup \{k\}}(x), \quad w_{i \cup \{k\}} = -\sum_{j \subseteq i} v_{i - j}(x)u_{j \cup \{k\}}(x).
\]

Again, transformation to a code is done by means of cross-multiplication.

Note that the hyperbolic sine and cosine can be calculated the same way (only with a different sign in the second 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 \texttt{crossmult2()} procedure. Thus the computation of the trigonometric functions has the same complexity as one full multiplication.
2.4. Overall complexity of the direct method. In general, the evaluation of cross-derivatives has a complexity of $O(3^n)$:

**Proposition 2.1.** The direct computation of all cross-derivatives $f_*$ of a function $f$ given as an evaluation procedure is itself an evaluation procedure with complexity

$$\text{OPS}(f_*) = O(3^n) \cdot \text{OPS}(f)$$

for the runtime and with a factor of $2^n$ in the memory size.

Here OPS refers to a platform-independent measure of evaluation complexity. The unit is the cost of one multiplication, which is also assumed to be the cost of addition or subtraction.

**Proof.** The initialization and linear operations require $2^n$ operations, most evaluation procedures of cross-derivatives for elementary operations have a complexity similar to the multiplication procedure $\text{crossmult}()$, which is $3^n$ scalar multiplications. Other functions like the power function $v(x) = u(x)^w(x)$ already have in their scalar version a complexity of several scalar multiplications, so that the factor of $3^n$ for the operations count remains intact. □

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 $3 \cdot 10^9$ processor cycles per second, this amounts to about 28 processor cycles per multiplication.

A speed-up of the direct method might be obtained if one were to construct an architecture which carries out the cross-multiplication function as one of its basic operations. In the following chapter an alternative method for propagating cross-derivatives is presented which is of $O(n^2 2^n)$ complexity and also applies to any programmable 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 $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ using only univariate Taylor expansions. From the polarization of quadratic forms we know that this is possible using the pair $\{(1,1), (1,-1)\}$ or the triplet $\{(0,1), (1,0), (1,1)\}$ of directions. This mixed derivative is obtained as the quadratic term in the Taylor expansion of the linear combinations

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

or

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

The second variant has been generalized in [GUW00] to the computation of all mixed partial derivatives up to some given order $d$ of some function $f$ in $n$ variables on the basis of univariate Taylor expansions in all directions $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ with $\sum_{j=1}^n \alpha_j = d$. There are $(n+d-1)$ such directions and each univariate expansion is at most $(d+1)(d+2)/2$ times as expensive as the evaluation of $f$ itself. (see Section 13.2 in [GW08b]). A recent and fast implementation of this method is available in the Rapsodia software package [CU09].
While the cross-derivatives are a subset of all mixed partial derivatives up to order \( d = n \), their computation using the approach sketched above would be inefficient. As we will see, we get by using \( 2^n - 1 \) rather than the \( \binom{2n}{n} \approx 4^n \) directions in the scheme of [GUW00], where the asymptotic estimate follows from Stirling’s formula. Moreover, the reconstruction of the cross-derivatives from the \((n+1)(2^n - 1)\) univariate Taylor coefficients can be done in \( n^2 2^n \) operations.

3.1. **Interpolation of all cross-derivatives from Taylor coefficients.** Define for every subset \( i \subseteq D = \{1, 2, \ldots, n\} \) the vector \( e_i = \sum_{i \in I} e_i \). Those vectors are the vertices of the \(n\)-dimensional hypercube \( \{0, 1\}^n \). As observed in [GUW00], the highest order cross-derivative \( T_D \) of any polynomial \( T : \mathbb{R}^n \rightarrow \mathbb{R} \) of degree less than or equal to \( n \) satisfies

\[
T_D = \sum_{j \subseteq \{1, 2, \ldots, n\}} (-1)^{n-|j|} T(e_j),
\]

which can be easily checked by partial integration.

Let \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) be at least \( n \) times continuously differentiable. Consider the Taylor polynomial \( T_f, n(v) \) for \( f(x + v) \) at some point \( x \in \mathbb{R}^n \) in direction \( v \in \mathbb{R}^n \) of order \( n \). The components of equal degree \( k \) in \( v \) will be arranged in the homogeneous parts \( T_k(v) \),

\[
T_f, n(v) = \sum_{k=0}^{n} T_k(v), \quad \text{with} \quad T_k(v) = D^k f(x)[v, v, \ldots, v]/k!.
\]

Obviously, any cross-derivative with index set \( i \subseteq \{1, 2, \ldots, n\} \) can now be obtained by formula (3.1) from the homogeneous component \( T_k \) of order \( k = |i| \) by considering all other coordinates \( x_j \) with \( j \notin i \) as constants, that is, by keeping their \( v_j = 0 \):

\[
f_i(x) = (T_{|i|})_1 = \sum_{j \subseteq \{1, \ldots, n\}} (-1)^{|i|-|j|} T_{|i|}(e_j).
\]

Note that \( |i - j| = |i| - |j| \) since \( j \subseteq i \).

The naive way to compute the cross-derivative terms would just evaluate this formula, requiring an iteration over all subsets of subsets \( j \subseteq i \subseteq \{1, \ldots, n\} \). Since each index may occur in \( i \) alone, in \( i \) and \( j \) or not at all, this gives \( 3^n \) combinations resulting in \( 3^n - 2^n \) operations to evaluate all cross-derivatives. However, in the course of those computations, certain subexpressions are evaluated repeatedly, which can be avoided.

3.2. **Efficient transformation of Taylor polynomials to cross-derivatives.**

To obtain a more efficient algorithm we combine all homogeneous components simultaneously in the \((n+1)\)-dimensional vector \( T_*(v) = (T_0(v), T_2(v), \ldots, T_n(v)) \). From those vectors we build alternating sums for pairs of index sets \( i, \ell \subseteq \{1, 2, \ldots, n\} \) with \( i \cap \ell = \emptyset \),

\[
Q_*(i, \ell) = \sum_{j \subseteq \{1, \ldots, n\}} (-1)^{|i|-|j|} T_*(e_{j\setminus \ell}),
\]

that is, \( i \) is the variable and \( \ell \) the constant part of the combined index set \( i \cup \ell \). As is easy to check, these alternating sums satisfy the recurrence relations:

- \( Q_*(\emptyset, \ell) = T_*(e_{\ell}) \) and
- \( Q_*(i \cup \{k\}, \ell) = Q_*(i, \{k\} \cup \ell) - Q_*(i, \ell) \).
Now, if we ensure that $\max i < k < \min \ell$ always holds in the recurrence relation, then the cross-derivatives can be simultaneously computed by transferring each index $k = 1, 2, \ldots, n$ from the constant to the variable index set. At the end of this process, the cross-derivative can be read off as $f_i(x) = Q_{|i|}(i, \emptyset)$.

Given the ordering of $1 < k < \ell$, the separation of $i \cup \ell$ and $i \cup \{k\} \cup \ell$ into the variable and constant subsets is uniquely specified by $k$, the recursion transforming the array of Taylor coefficients $T_\epsilon(e_i)$ into the array of alternating sums $Q_\epsilon(i, \emptyset)$ can be executed in place, that is, only using the input array. Further, note that

$$\text{ind}(i \cup \{k\} \cup \ell) = \text{ind}(i \cup \ell) + 2^{k-1}$$

and that there are, in the order prescribed by the ind-function, alternating groups of $2^{k-1}$ subsets without and $2^{k-1}$ subsets with the index $k$ as element. This gives the following as one possible implementation:

```c
int k; for(k=1;k<=n;k++){
    int i, mi=1<<(k-1), ml=1<<(n-k); /* mi=2^(k-1), ml=2^(n-k) */
    for(l=0;l<ml;l++)for(i=0;i<mi;i++){
        int ind=2*l*mi+i;
        Q[ind+mi]-=Q[ind]; /* overloaded vector subtraction */
    }
}
```

Hence we observe that there are $n2^{n-1}$ cycles of the three loops and the vector subtraction concerns only the last $n$ elements, since the function value in the first component is known. This results in $n^2 2^{n-1}$ operations. One could avoid some more operations since $Q_m(i, \ell) = 0$ for $m < |i|$.

Since the computation of each cross-derivative $f_i(x) = Q_{|i|}(i, \emptyset)$ involves $2^{|i|} - 1$ subtractions, serious cancellation errors may accumulate, especially for higher order derivatives. To mitigate this effect, a doubly accurate subtraction method (derived from the accurate addition by Knuth [Knu69], see also [ORO05]) was used that captures the floating point errors and accumulates them for a first order correction of the result. Hence the command $Q[ind+mi]-=Q[ind]$ is replaced with

```c
a=Q[ind+mi]; b=Q[ind]; d=a-b; /* d=fl(a-b), d+eps=a-b */
c=d-a; eps=(a-(d-c))-(b+c)); /* err_d=eps+err_a-err_b */
Q[ind+mi]=d; err[ind+mi]+=eps-err[ind];
```

The correction is applied by using $Q[ind]+err[ind]$ in the final result.

In our experience, the interpolation using accurate differences takes about three times as long as in double precision arithmetics. This extra effort has no noticeable effect on the overall evaluation times, since it only needs to be performed once at the end of a long chain of operations. Each nonlinear operation already causes costs of $O(n^2 2^n)$.

Proposition 3.1. The application of exact interpolation of Taylor polynomials with the proposed algorithm to the computation of all cross-derivatives $f_\epsilon$ of a function $f$ given as an evaluation procedure is itself an evaluation procedure with complexity

$$OPS(f_\epsilon) = O(n^2 2^n) \cdot (OPS(f) + c), \quad c \leq 4,$$

for the runtime and with a factor of $(n+1) 2^n$ in the memory size.

Proof. This method first evaluates Taylor polynomials of degree $n$ in $(2^n - 1)$ directions. Each evaluation has a cost of about $n^2 OPS(f)$. The extraction of the cross-derivatives from the Taylor coefficients requires $n^2 2^{n-1}$ subtractions in native double precision and eight times that for the accurate subtraction. \qed
4. Comparison of the direct and interpolation approaches

We compare the two approaches in numerical experiments. To get a fair picture we first optimize the basic multiplication procedure of the direct method by replacing the \(d\) lowest levels of the recursion by a single function call.

4.1. Optimizing the direct multiplication. In the implementation of the direct method, the procedures mainly consist of some preparation and then calls to the multiplication or convolution procedures. Making this faster thus increases the speed of the whole evaluation method. The lowest \(d\) levels of the recursion can be replaced by one call to a function that contains all of the operations required for the multiplication of two small \(d\)-dimensional cubes. The required operations on subsets can be emulated by logical operators on the bit patterns of the index variables such as “or” \(i|j\) for the union and “and” \(i&j\) for the intersection of the corresponding subsets.

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

This iteration generates a highly non-ordered access pattern to elements of the three cubes and results in a tradeoff between a reduced number of recursive function calls and an increasing number of cache misses.

Runtime experiments were performed to find the optimal dimension of a small cube at which the recursive multiplication procedure \texttt{crossmult()} would switch to the non-recursive variant \texttt{directmult()}. These experiments are summarized in Figure 1 (left). We used different dimensions \(n\) of the big input cubes (corresponding to the total number of input variables) and for each of them generated a curve representing the runtime \(t\) of the multiplication on the vertical axis in dependence on the dimension \(d\) of the small cube, corresponding to an array size of \(h = 2^d\). To avoid widely differing scales for different values of \(n\), the runtime was converted into processor cycles per scalar multiplication, thus the actual runtime is \(t \cdot \frac{3^n}{3\text{GHz}}\).

As one can see, the graphs for various dimensions \(n\) of the big cube are very similar. By inspection we see that initially there is a considerable reduction in the
runtime until the small cube dimension reaches 4 – 5. Then until dimension 15 of the small cube the runtime remains approximately constant. Thus there is no gain for the increased complexity. For bigger dimensions the runtime again increases. As a conclusion, in the implementation for all the other experiments we stopped the recursion at small cubes of dimension 4, with additional provisions for initial cube dimensions smaller than 4.

4.2. Optimizing the Taylor polynomial method. For the implementation of the Taylor arithmetics the ADOL-C automatic differentiation library was used. With a call to the `hov_forward()` procedure, this library allows to simultaneously evaluate several 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, and the management of those blocks becomes increasingly inefficient.

This effect can be seen in Figure 1 (right), where a test function with evaluation complexity $O(n)$ was evaluated for its cross-derivatives. For each of $n \in \{14, 15, 16\}$, the runtime $t$ is represented against the block size $b$. The expected runtime is proportional to $n^3 2^n$. To achieve a common range the curves for the different $n$ are scaled by this factor, i.e., the actual runtime was $t \cdot n^3 2^n/3GHz$. The logarithmic horizontal axis contains the number $b$ of simultaneous directions in each of the about $2^n/b$ calls to `hov_forward()`. There is a clearly visible 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. Asymptotically, the interpolation method can be improved by propagating the univariate Taylor expansion using faster polynomial multiplication, reducing the overall complexity to $O(2^n n^1.58)$ for Karatsuba and $O(2^n n \log_2(n))$ for FFT. However, especially for FFT the benefits
are only expected to accrue for values of \( n \) where the leading exponential \( 2^n \) by itself is already unacceptably large.

Currently, the direct method uses direct function calls for the elementary operations, whereas the Taylor method via `hov_forward()` in ADOL-C interprets an internal representation of the operation sequence. Therefore, one has to expect some factor larger than 1 in the Taylor method until the so-called traceless mode of ADOL-C has been extended to also propagate univariate Taylor series. As can be seen in Figure 2 (left), assuming a moderate factor between 1 and 2, a cross-over point occurs between \( n = 12 \) and \( n = 15 \), which is consistent with the experimental determinations of cross-over points in Figures 2 (right) and 3. There, Figure 2 (right) 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 3 compare the runtimes of both methods 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

Figure 3. Runtimes for the direct and Taylor methods for three variants of the integration test function
we will discuss directly below, the Taylor interpolation method should be implemented with an extra-accurate subtraction in order to maintain a reasonable level of 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 auxiliary derivatives have values that are orders of magnitudes bigger than the cross-derivatives, leading to rounding errors that influence the values of the cross-derivatives.

![Figure 4. Errors while evaluating the third test function using the Taylor method](image)

That such effects have to be taken into account is shown in Figure 4 for the third test function. In the left diagram each curve shows, ranging over the number $n$ of variables, the fraction of all $2^n$ cross-derivatives that have a numerical error above the error thresholds associated to that curve, ranging from $10^{-8}$ to $10^{-16}$. For instance, in the considered range only 20% or less of the coefficients have an error larger than $10^{-12}$. The right diagram shows the numerical error of the highest order cross-derivative alone. Since the product structure of the test functions leads to a corresponding product structure in the cross-derivatives, and this structure is exactly captured by the evaluation via the direct method, we assumed for this test case that the results of the direct method are exact within the machine precision. The errors are computed against these values. Without the extra-accurate subtraction the errors were orders of magnitude larger.

4.5. Cross-derivatives for a class of Asian options. We consider as examples simplified versions of functions used in mathematical finance. More specifically, we consider the standard and Brownian bridge construction for arithmetic average Asian options, simplified to the case of strike price $K = 0$. We assume that the asset $S_t$ follows the geometric Brownian motion model based on the stochastic differential equation

\[
\frac{dS_t}{S_t} = rS_t dt + \sigma S_t dW_t,
\]

where $r$ is the risk-free interest rate, $\sigma$ is the volatility and $W_t$ is the standard Brownian motion (see [WS07]). This stochastic equation can be solved to obtain

\[
S_t = S_0 \exp \left( \left( r - \frac{\sigma^2}{2} \right) t + \sigma W_t \right).
\]
Thus simulating asset prices reduces to simulating discretized Brownian motion paths $W_{t_1}, \ldots, W_{t_n}$. We assume that the asset prices are sampled at equally spaced times $t_j = j \Delta t$, $j = 1, \ldots, n$ and $\Delta t = T/n$. Because the Brownian motion is a Gauss process the vector $(W_{t_1}, \ldots, W_{t_n})$ is normally distributed, and in this particular case exhibits mean zero and covariance matrix $C$ given by

$$C = (\min(t_i, t_j))_{i,j=1}^n.$$

The option valuation problem for the arithmetic average Asian options takes the form

$$V = \frac{e^{-rT}}{(2\pi)^{n/2} \sqrt{\det(C)}} \int_{\mathbb{R}^n} \max\left(\frac{1}{n} \sum_{j=1}^n S_j(w) - K, 0\right) e^{-\frac{1}{2}w^\top C^{-1}w} dw,$$

with $w = (W_{t_1}, \ldots, W_{t_n})^\top$. After a factorization $C = AA^\top$ of the covariance matrix, it is equal to

$$V = \frac{e^{-rT}}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} \max\left(\frac{1}{n} \sum_{j=1}^n S_j(Az) - K, 0\right) e^{-\frac{1}{2}z^\top z} dz,$$

with $Az = w$. The new variable vector $z = (z_1, \ldots, z_n)$ can be assumed to consist of independent standard-normally distributed random variables. Then the identity $w = Az$ defines a construction method for Brownian paths.

We can transform the integral in (4.3) using the inverse normal cumulative distribution function $\Phi^{-1}(\cdot)$. The valuation problem now consists of the evaluation of

$$V = e^{-rT} \int_{[0,1]^n} \max\left(\frac{1}{n} \sum_{j=1}^n S_j(A\Phi^{-1}(x)) - K, 0\right) dx,$$

with $A\Phi^{-1}(x) = (\Phi^{-1}(x_1), \ldots, \Phi^{-1}(x_n))^\top$. Under the simplification $K = 0$, the valuation problem reduces to

$$V = e^{-rT} \int_{[0,1]^n} \frac{1}{n} \sum_{j=1}^n S_j(A\Phi^{-1}(x)) dx.$$

We consider two particular constructions of Brownian paths associated to factorizations of $C$ (see [GKS10]):

- The standard construction (SC):

$$W_0 = 0, W_{t_j} = W_{t_{j-1}} + \sqrt{\Delta t} z_j, \ j = 1, \ldots, n.$$

This results in a factorization matrix $A$ with entries

$$A_{i,j} = \sqrt{\Delta t} 1_{\{i \geq j\}},$$

corresponding to the Cholesky factorization of $C$. 

License or copyright restrictions may apply to redistribution; see https://www.ams.org/journal-terms-of-use
The Brownian bridge construction (BB):
assume for simplicity that \( n = 2^p \). Then the Brownian motions are generated by determining its values at \( T, T/2, T/4, 3T/4, \ldots \) as follows:

\[
\begin{align*}
W_0 & = 0, \\
W_T & = \sqrt{T} z_1, \\
W_{T/2} & = (W_0 + W_T)/2 + \sqrt{T/4} z_2, \\
W_{T/4} & = (W_0 + W_{T/2})/2 + \sqrt{T/8} z_3, \\
W_{3T/4} & = (W_{T/2} + W_T)/2 + \sqrt{T/8} z_4, \\
& \vdots \\
W_{(n-1)T/n} & = (W_{(n-2)T/n} + W_T)/2 + \sqrt{T/(2n)} z_n.
\end{align*}
\]

This case corresponds to a different factorization of the matrix \( C \).

The construction can be generalized to include subdivision intervals of unequal length, thus allowing any dimension number \( n \geq 2 \). Given the values \( W_{t_j} \) and \( W_{t_k} \), we simulate \( W_{t_i} \) for any \( t_j < t_i < t_k \) following the BB formula

\[
W_{t_i} = (1 - \nu)W_{t_j} + \nu W_{t_k} + \sqrt{\nu(1-\nu)(k-j)/2} z,
\]

with \( z \) following the standard normal distribution and \( \nu = \frac{i-j}{k-j} \). In this article we simulate intermediate values \( W_{t_i} \) by taking \( t_i = t_j + \lfloor \frac{k-j}{2} \rfloor \Delta t \), corresponding to the choice \( i = j + \lfloor \frac{k-j}{2} \rfloor \); see 3.1 in [Gla04] for more details.

We summarize the first part of the numerical results in Table 1 and Table 2. In Table 1 we show the relative cost of evaluating the full cross-derivatives of the SC and BB function examples for \( d = n \) with respect to the cost of evaluating the functions itself. In Table 2 we show the relative cost of evaluating the first cross-derivatives \( f_i, i \subseteq \{1, \ldots, d\} \), of the SC and BB function examples with \( d = 8 \) and real dimensions \( n = 8, 16, 32, 64, 128, 256, 360, 512 \). We tested this by taking 20 runs of 1000 (pseudo)random point evaluations of the functions and their cross-derivatives in \([0,1]^n\) for the tests in Table 1, and 100 runs of 10000 (pseudo)random point evaluations for the tests in Table 2.

We show the average relative time cost over the runs rounded up to the next integer. The maximum relative time cost for any one of the runs was never more than 30% higher. For the (pseudo)random numbers we have used the Mersenne-Twister generator from Matsumoto and Nishimura [MN98]. For the function \( \Phi^{-1}(\cdot) \), we have used Moro’s algorithm [Gla04]. We extended it to double precision by adding a Newton step using \( \Phi(\cdot) = 1.0 - \text{cPhi}(\cdot) \), with \( \text{cPhi} \) from [Mar04], which seems to be more accurate than the available C-library functions \( \text{erf}(\cdot) \) or \( \text{erfc}(\cdot) \).

We have fixed the parameters to \( K = 0, S_0 = 100, r = 0.1, \sigma = 0.1, T = 1.0 \). No domain truncation strategy was considered in these tests. We do not expect very different results for truncation strategies like those considered in [GKS10], because the truncation constants affect the single entries of the univariate functions \( \Phi^{-1}(\cdot) \).

In Table 1 we observe that the runtime ratios for total cross-derivation (\( d = n \)) behave like \( \mathcal{O}(3^n) \), but with small implied constants. Further calculations showed
Table 1. \((d = n)\)

<table>
<thead>
<tr>
<th>Example (n = )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime(crossSC)</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>27</td>
<td>74</td>
</tr>
<tr>
<td>Runtime(SC)</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>15</td>
<td>36</td>
</tr>
</tbody>
</table>

that the cost ratio grows like \(10 \cdot 3^{n-6}\) for large \(d = n \geq 6\) on the SC function example, and like \(10 \cdot 3^{n-7}\) for large \(d = n \geq 7\) on the BB function example.

Table 2 is more interesting for potential applications in high dimensional integration. In a slightly different implementation, it shows that the computation of cross-derivatives for weights calculation (which in practice may require much less accuracy than the intended integral to calculate) can be accessible if the effective dimension \(d\) is small itself and compared to the real dimension \(n\). This is the case for the simplified arithmetic Asian option BB example.

Table 2. \((d = 8)\)

<table>
<thead>
<tr>
<th>Example (n = )</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>360</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime(crossSC)</td>
<td>74</td>
<td>42</td>
<td>24</td>
<td>15</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Runtime(SC)</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
<td>37</td>
</tr>
</tbody>
</table>

We proceed to investigate the calculation of optimal weights for construction of shifted rank-1 lattice rules. The weights are used by the CBC algorithm (see \cite{NC06}) for construction of the generator vector of the integration lattice. More specifically, we consider the calculation of product weights \(\gamma_{n,i} := \prod_{j \in i} \gamma_{n,j}\). For this purpose, we consider the integration error bound for the unanchored Sobolev space \(W_2^{(1,...,1)}([0,1]^n)\) stated in \eqref{eq:1.7}. The latter bound can be rewritten as

\[
| (I - Q_{N,n})(f_{\text{EFF}}) | \leq \frac{1}{N^{\frac{1}{2}}} \left( \prod_{j=1}^{n} \left( 1 + \frac{\gamma_{n,j}}{6} \right) - 1 \right)^{\frac{1}{2}} B_{\gamma,f},
\]

where we used the short notation

\[
B_{\gamma,f} := \left( \sum_{\emptyset \neq I \subseteq \{1,...,d\}} \gamma_{n,1}^{-1} b_{i,f} \right)^{\frac{1}{2}} \quad \text{and} \quad b_{i,f} := \int_{[0,1]^n} \left( \frac{\partial^{|i|}}{\partial x_i} f(x) \right)^2 dx.
\]

We use the new differentiation technique to evaluate the terms \(b_{i,f}\) inside \(B_{\gamma,f}\). Given a fixed small \(\epsilon_0\), the weights \(\gamma_{n,j}, \ d < j \leq n\) corresponding to less important variables (in ANOVA sense) can be chosen small enough to ensure an integration error bound for the effective part of the function of the form

\[
| (I - Q_{N,n})(f_{\text{EFF}}) | \leq (1 + \epsilon_0) \frac{1}{N^{\frac{1}{2}}} \left( \prod_{j=1}^{d} \left( 1 + \frac{\gamma_{n,j}}{6} \right) - 1 \right)^{\frac{1}{2}} B_{\gamma,f}.
\]

The latter bound holds because the right-hand side of \eqref{eq:4.9} is an increasing function of the weights \(\gamma_{n,j}, \ d < j \leq n\) (see also \cite{LG12}). We calculate the weights for the simplified arithmetic Asian option BB example by minimizing \eqref{eq:4.10}. Note
that the integrand does not belong to the weighted unanchored space due to the
divergence of the terms $b_{i,f}$. To obtain an integrand in the weighted unanchored
Sobolev space, a domain truncation strategy as described in [GKS10] is applied.
We estimate the terms $b_{i,f}$ by taking a small number of sample points compared
to the final size of the sample used for the integral of $f$. In this particular example
we take about 1% of the number of points used for the actual integration. We
believe that this approach will deliver a practical estimation of the terms $b_{i,f}$ relevant
for the final sample size. We approximate these integrals by using Sobol’ points
with direction numbers taken from [JK08], and apply the domain truncation strat-
egeny proposed in [GKS10]. Note that the domain truncation error $\epsilon$ from [GKS10]
applied to this example implies that the resulting integral differs from the original
one by at most $\epsilon S_0$.

For the constant parameters $K = 0, S_0 = 100, r = 0.1, \sigma = 0.1, T = 1$, we
consider the different dimensions $n = 8, n = 16$, and $n = 128$ with fixed $d = 8$. We
set the domain truncation error to $\epsilon = 0.1$ and estimate the integrals of the squared
derivatives $b_{i,f}$ by:

- $2^{11}$ Sobol’ points for $n = 8$, $n = 16$, and $n = 128$; denoted by “n8_S11”,
  “n16_S11” and “n128_S11”, respectively.
- $2^{14}$ Sobol’ points for $n = 8$, $n = 16$, and $n = 128$; denoted by “n8_S14”,
  “n16_S14” and “n128_S14”, respectively.
- $10 \times 2^{18}$ Sobol’ points for $n = 8$ and $10 \times 2^{22}$ Sobol’ points for $n = 16$
  denoted by “n8_acc” and “n16_acc”, respectively.

The estimations with $2^{11}$ and $2^{14}$ Sobol’ points are meant to be used for 10-times
randomly shifted lattice rules with prime number of points rounding $2^{14}$ and $2^{17}$,
respectively.

We randomly shifted 10 times $2^{18}$ and $2^{22}$ Sobol’ points for the accurate estima-
tions “n8_acc” and “n16_acc”, respectively, and repeated this procedure 10 times to
estimate the standard errors. For the accurate estimations, the computations were
carried out using long-double precision. We observed that at least one significant
digit accuracy in every term $b_{i,f}$ was reached for “n8_acc” and “n16_acc”. Moreover,
the integrals of squared first and second order mixed derivatives exhibited 3 and
often 5 significant digits accuracy in this case. Under these truncations and QMC-
sampling strategies, a smaller domain truncation error $\epsilon$ can require a significant
greater amount of samples for the accurate calculation of the terms $b_{i,f}$.

Due to the special form of the BB construction, an important part of the cal-
culated mixed derivatives equals zero. We implemented the cross-derivation in a
straightforward way. A specialized implementation for this example could reduce
the relative runtime of cross-derivatives.

The first order necessary conditions for the minimization of bound (4.10) seems
only explicitly solvable for $d = 2$, and even then it can be seen that the problem
is not necessarily convex. However, all level sets are compact and lie within the
open positive orthant $\gamma_{n,j} > 0$, with $1 \leq j \leq d$. To compute optimal weights
for $d = 8$, we used the NLP solver LRAMBO that was developed for small to
medium scaled dense problems. The algorithm of LRAMBO is based on a total
quasi-Newton method (see [GW02] and [BGLS10]), where the required derivative
matrices are approximated by low rank updates; see [GWK07]. The gradient is
efficiently computed by Algorithmic Differentiation; see [GW08a]. In this example,
stepping outside of the orthant $\gamma_{n,j} > 0, 1 \leq j \leq d$ can be prevented by a simple line-search modification.

A robust test using 1000 different random starting values was carried out for LRAMBO for all scenarios listed below. The average runtime never exceeded 0.08 seconds. All runs lead to the same local minimum for each scenario. Usually, setting all variables to the value 1.0 for the starting point resulted in very fast convergence.

In the following table we list the values of the calculated optimal weights for each scenario for the terms $b_{i,f}$. The optimal weights achieved in most cases a reduction of the bound value (4.10) of about 30% compared to the bounds evaluated with harmonically or quadratically decaying weights.

<table>
<thead>
<tr>
<th>weight</th>
<th>n8_S11</th>
<th>n8_S14</th>
<th>n8_acc</th>
<th>n16_S11</th>
<th>n16_acc</th>
<th>n128_S11</th>
<th>n128_S14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma_{n,1}$</td>
<td>1.1024</td>
<td>1.1328</td>
<td>1.1244</td>
<td>1.3683</td>
<td>1.4548</td>
<td>1.4422</td>
<td>1.9008</td>
</tr>
<tr>
<td>$\gamma_{n,2}$</td>
<td>0.4493</td>
<td>0.4567</td>
<td>0.4574</td>
<td>0.5881</td>
<td>0.6036</td>
<td>0.6141</td>
<td>0.8837</td>
</tr>
<tr>
<td>$\gamma_{n,3}$</td>
<td>0.1431</td>
<td>0.1488</td>
<td>0.1476</td>
<td>0.1854</td>
<td>0.1987</td>
<td>0.1943</td>
<td>0.3120</td>
</tr>
<tr>
<td>$\gamma_{n,4}$</td>
<td>0.1642</td>
<td>0.1785</td>
<td>0.1762</td>
<td>0.2062</td>
<td>0.2488</td>
<td>0.2400</td>
<td>0.2764</td>
</tr>
<tr>
<td>$\gamma_{n,5}$</td>
<td>0.0477</td>
<td>0.0493</td>
<td>0.0492</td>
<td>0.0597</td>
<td>0.0637</td>
<td>0.0640</td>
<td>0.0797</td>
</tr>
<tr>
<td>$\gamma_{n,6}$</td>
<td>0.0535</td>
<td>0.0587</td>
<td>0.0565</td>
<td>0.0670</td>
<td>0.0826</td>
<td>0.0755</td>
<td>0.0876</td>
</tr>
<tr>
<td>$\gamma_{n,7}$</td>
<td>0.0601</td>
<td>0.0609</td>
<td>0.0613</td>
<td>0.0856</td>
<td>0.0808</td>
<td>0.0834</td>
<td>0.2278</td>
</tr>
<tr>
<td>$\gamma_{n,8}$</td>
<td>0.0621</td>
<td>0.0654</td>
<td>0.0639</td>
<td>0.0805</td>
<td>0.0889</td>
<td>0.0871</td>
<td>0.1108</td>
</tr>
</tbody>
</table>

The calculation of the coefficients $b_{i,f}$ based on at most 1% of the sample size used for the integration of $f$ incurs an extra overhead of about 50% of the total runtime for the integration of $f$, in the case of the BB function example. Further reductions in the overhead and better accuracy can be achieved by improved sampling strategies.

The optimization tool works very fast for this particular form of (low dimensional) objective functions. Therefore the rough estimation of optimal weights is feasible in the “pre-computing” part before the real integration takes place. An improved integration strategy for the coefficients $b_{i,f}$ should especially sample more points on the border regions of the sub-cube corresponding to the first $d$ variables, where the integrands of the terms $b_{i,f}$ become more sensitive.

The considered bound (1.7) exhibits only convergence $O(N^{-\frac{1}{2}})$, like MC. An improved bound for the worst case error in (1.7) exhibiting rates of convergence $O(N^{-\tau})$, for $\tau \in [\frac{1}{2}, 1)$, can be founded in Theorem 16.21 in [NW10]. Randomly shifted lattice rules usually achieve convergence like $O(N^{-1})$ in the BB option valuation problem. Choosing a different convergence rate constant $\tau$ closer to 1 affects the minimization of the integration error bound and therefore also the resulting weights, but the estimation of the semi-norm remains unaffected. Comparison of performances of shifted lattice rules constructed by choosing optimal weights for different constants $\tau$, and comparisons with other weights usually considered by practitioners, will be reported in [LG12].

The proposed method to calculate weights assumes that the effective dimension in truncation sense of the integrand is known in advance. In several applications the integrands are given as part of a parametric family of functions, where the effective dimension is not very sensitive to the parameters. This seems to be the case for the simplified example of Asian options with BB construction, were one can expect $d \leq 8$ for a wide range of realistic parameters $r, \sigma$ and $T$. 

License or copyright restrictions may apply to redistribution; see https://www.ams.org/journal-terms-of-use
5. Summary and conclusion

We demonstrated how to implement the computation of the full set of cross-derivatives 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 Taylor polynomials along this chain and recombining the cross-derivatives from the Taylor coefficients. In agreement with the theoretical complexity estimates we found that the cross-over in terms of runtimes lies at about 12 to 15 variables, but is sensitive to changes in the multiplicative constants. In order to maintain a reasonable level of accuracy one should implement it with an accurate difference evaluation, which has no significant effect on the overall runtime.

Using the second method the average cost of each cross-derivative is less than \( n^2 \) that of evaluating the function itself. This factor can be asymptotically reduced to \( O(n \log(n)) \) by use of FFT.

For the pricing of Asian options it was found that using the direct differentiation approach to calculate all cross-derivatives with respect to \( d \) variables caused a much smaller increase in the runtime than the expected factor \( 3^d \). Moreover, for the Brownian bridge construction it is well known that the effective dimension \( d \) is much smaller than the total number \( n \) of variables, which represent trading periods. We demonstrated the possibility to calculate all relevant cross-derivatives and to construct improved quasi-Monte Carlo methods for the Brownian bridge construction for Asian options.

Since the memory size of the output, and naturally also the runtimes, grow exponentially with the number of variables, the computation of all cross-derivatives is limited in its applicability. Future research has to provide methods to evaluate all cross-derivatives up to some order that is much smaller than the number of variables, which may be in the hundreds or thousands. Of course, then the convenient cube allocation is no longer possible.

Rather than to evaluate cross-derivatives pointwise, one can propagate bounds on their size in an algorithmic differentiation fashion, with an effort that is just quadratic in the total number of variables \( n \). These bounds could then be used to construct special QMC methods. We are currently investigating whether the resulting bounds are still useful, i.e., not too conservative, in relevant applications.

We would like to thank the reviewers for their comments and suggestions which were of great help and lead us to improve the article.

References


Department of Mathematics, Humboldt-Universität zu Berlin, Unter den Linden, 610099 Berlin

E-mail address: griewank@math.hu-berlin.de

Department of Mathematics, Humboldt-Universität zu Berlin, Unter den Linden, 610099 Berlin

E-mail address: llehmann@math.hu-berlin.de

Department of Mathematics, Humboldt-Universität zu Berlin, Unter den Linden, 610099 Berlin

E-mail address: leovey@math.hu-berlin.de

Haifa

E-mail address: marat.zilberman@gmail.com