Coarse Grain Automatic Differentiation: a Practical Approach to Fast and Exact Computation of First and Second Order Derivatives in Software

Henri-Olivier Duché\textsuperscript{1} \hspace{1em} François Galilée\textsuperscript{2}

November 23, 2017

Abstract

The evaluations done by a program at runtime can be modeled by computational Directed Acyclic Graphs (DAGs) at various abstraction levels. Applying the multivariate chain rule on those computational DAGs enables the automation of total derivatives computation, which is exploited at a fine-grain level by Automatic Differentiation (AD). Coarse Grain Automatic Differentiation (CGAD) is a framework that exploits this principle at a higher level, leveraging on software domain model. All nodes in the computational DAG are responsible for computing local partial derivatives with respect to their direct dependencies while the CGAD framework is responsible for composing them into first and second order total derivatives. This separation of concerns between local and global computations offers several key software engineering advantages: it eases integration, makes the system decoupled and inherently extensible and allows hybrid differentiation (i.e. connecting derivatives from different sources using different calculation methods and different languages in the same evaluation). Additionally, the nodes can take advantage of local symbolic differentiation to significantly speed up total derivatives computation, compared to traditional differentiation approaches. As a concrete application of this methodology to a financial software context, we present a Java implementation that computes the premiums and 82 first and second order total derivatives of 2000 call options in 262 milliseconds, with a time ratio of 1 : 2.2 compared to premiums alone.

Introduction

Derivatives are fundamental in many numerical activities, from physics to economics, to predict an evolution, speed up a computation or help to make a decision. In finance the total derivatives of instrument prices with respect to their underlying parameters are called Greeks. They are instrumental for risk management and highly desirable for the quick Taylor approximation of prices when underlying parameters change (either in response to real time feed events or simulation scenarios).

Unfortunately every classical methods for calculating derivatives in software comes with its own drawbacks. And to make matters worse, those drawbacks get magnified when computing second order derivatives.

\textsuperscript{1}hoduche@yahoo.fr
\textsuperscript{2}francois.galilee@acm.org
Choosing one differentiation method is usually a trade-off between responding to user experience concerns (precision, speed) and responding to software engineering concerns (decoupling, integrability ...).

1. Symbolic differentiation (i.e. coding the derivative formulas) addresses user experience concerns as it is both exact and fast. It works very well on small blocks of code, however it becomes impractical on large codes as derivative formulas tend to grow exponentially making the code too complex to write and maintain [DG17]. In addition, total derivative formulas are global quantities that contain local partial derivatives from all calculation modules, linked together by the chain rule. So coding total derivative formulas unfortunately induces coupling between all calculation modules.

2. Finite difference approach consists in slightly bumping one factor, repeating the calculation and approximating a derivative by a difference quotient. It addresses software engineering concerns as it is simple to code and induces no coupling. However it does not meet user experience requirements as it is often highly inefficient and potentially inaccurate [Nau12].

3. AD is an increasingly popular alternative that is exact up to machine precision and efficient at first order in specific cases. It comes in two modes: forward mode and adjoint mode. The forward mode is efficient when the function has one input and a large number of outputs while the adjoint mode (commonly called AAD) is efficient when the function has a large number of inputs and one output as it is typically the case in finance [GG06]. However, at second order, the speed advantage of AD compared to finite difference is lost as it becomes linear in the number of inputs [Hen17]. Additionally, AD integration is a time-consuming process as it requires a very bottom-up approach [Hen17].

CGAD is a new method to compute first and second order total derivatives that aims at responding to user experience concerns as well as software engineering concerns.

1 Computational DAGs in Software

1.1 Computational DAG at Assembler Level

It is possible to draw the data flow diagram (DFD) of a program at runtime. This is accomplished by monitoring numbers written and read by a processor [GW08] as in Figure 1 where data are displayed in boxes and assembler instructions in circles.

By letting arrows implicitly represent assembler instructions and labeling data with the assembler instruction that output them, the DFD is semantically equivalent to the tree in Figure 2.

Since numbers in memory are typically used as the inputs of several instructions, the tree can be further compacted and represented as the DAG in Figure 3.
The AD community has explored a way to differentiate at this instruction level and built a prototype called ADAC (Automatic Differentiation of Assembler Code) [GNC07].

1.2 Computational DAG at Programming Language Level

The assembler instructions that are executed at runtime are typically generated by a compiler from a code written in a higher-level programming language. Such a programming language offers a way for the developer to control the instructions and memory access generated for the processor via arithmetic operators, intrinsic mathematical functions (sqrt, exp, ln . . .), variable scopes and caching mechanisms.

Therefore, when coding an evaluation program, the developer is actually giving hints (not totally controlling as the compiler and the processor have the final say) for the topological ordering in which a data flow DAG should be constructed, traversed and discarded at runtime to produce the desired output.

Considering those arithmetic operators and intrinsic mathematical functions, we can draw a higher abstraction level DAG representation for the same program runtime evaluation. The nodes no longer correspond to assembler instruction outputs but to the assignment of input variables and programming language function results [Nau12].

As a concrete example from the financial world, consider a program to evaluate European call options. A European call option $C$ is a contract giving the right to buy a stock $S$ at an agreed price $K$ (called the strike), on a specified future maturity date $T$. Such option is typically valued using the Black formula that requires as additional inputs the stock volatility $V$ and a risk-free interest rate $Z$ [Bla76]:

$$ C = DF \cdot (F\Phi(d_1) - K\Phi(d_2)) $$

where:

\[ DF = e^{-ZT} \quad ; \quad F = \frac{S}{DF} \quad ; \quad d_1 = \frac{\ln(\frac{F}{K})}{V\sqrt{T}} + \frac{V\sqrt{T}}{2} \quad ; \quad d_2 = \frac{\ln(\frac{F}{K})}{V\sqrt{T}} - \frac{V\sqrt{T}}{2} \]

and \( \Phi \) stands for the cumulative distribution function of the standard normal law.

The runtime computational DAG of an implementation of the Black formula in a classical programming language like C++ or Java is given in Figure 4.
AD typically works on this abstraction level computational DAG by providing and composing
the derivatives of the arithmetic operators and intrinsic mathematical functions of the program-
ming language [Nau12, BH96].

1.3 Computational DAG at Domain Model Level

When designing complex software, developers typically build a domain model to represent mean-
ingful concepts, relations and behaviours in the application domain [Fow02]. The arithmetic
operators and intrinsic mathematical functions of the programming language serve as basic
materials to implement the high abstraction level functions of the domain model.

It is therefore possible to draw a even higher abstraction level computational DAG representa-
tion of the same program runtime evaluation by considering only domain model functions. The
nodes no longer correspond to single assignments of programming language function results but
to single assignments of domain model function results.

For example, for the same runtime evaluation of Figure 4, it is possible to arbitrarily only
consider the high-level domain model functions framed in Figure 5, to get the domain model
computational DAG of Figure 6.

Domain model computational DAGs are of paramount importance in software as they illustrate
at runtime the domain model built at design-time, as shown in Figure 6 and Figure 7.

CGAD works on this higher abstraction level computational DAG by composing the local partial
derivatives provided by the domain model functions.
1.4 Computational DAGs Equivalence

As illustrated by Figure 8, several computational DAGs correspond to the same evaluation at runtime, seen from different perspectives i.e. depending on the set of functions considered: assembler instructions, programming language functions or domain model functions. And each of those different abstraction level computational DAGs has its associated way of computing total derivatives.
2 Total Derivatives on a Computational DAG

2.1 Multivariate Chain Rule

In Leibniz's notation, if a variable \( z \) directly depends on a variable \( y \), which itself depends on a variable \( x \): \( z(y(x)) \), the first order univariate chain rule states that:

\[
\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}
\]

In the multivariate case, if a variable \( z \) directly depends on \( n \) variables \( y_1, \ldots, y_n \), which themselves all depend on a variable \( x \): \( z(y_1(x), \ldots, y_n(x)) \), it is necessary to distinguish partial from total derivatives:

- Partial derivatives with respect to one variable consider that the other variables are constant. They are noted:
  \[
  \left( \frac{\partial z}{\partial y_i} \right)_{\{y_j \mid j \in [1,n] \land j \neq i\}}
  \]
  or simply \( \frac{\partial z}{\partial y_i} \)

- Total derivatives with respect to one variable take into account indirect dependencies between the variables. They are noted:
  \( \frac{dz}{dx} \)

The first order multivariate chain rule is given by Formula 1:

\[
\frac{dz}{dx} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \cdot \frac{dy_i}{dx} \tag{1}
\]

To state the second order multivariate chain rule, we need to consider another variable \( u \) that can either be identical to or distinct from \( x \) (both cases leading to the same generic formula).

If a variable \( z \) directly depends on \( n \) variables \( y_1, \ldots, y_n \), which themselves all depend on two variables \( x \) and \( u \): \( z(y_1(x,u), \ldots, y_n(x,u)) \), by differentiating Formula 1 with respect to \( u \) and by the symmetry of second derivatives, we get Formula 2 which is the second order multivariate chain rule:

\[
\frac{d^2 z}{dx \, du} = \frac{d^2 z}{du \, dx} = \frac{d}{du} \left( \frac{dz}{dx} \right) = \frac{d}{du} \left( \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \cdot \frac{dy_i}{dx} \right)
\]

applying the sum and then the product differentiation rules:

\[
\frac{d^2 z}{dx \, du} = \sum_{i=1}^{n} \frac{d}{du} \left( \frac{\partial z}{\partial y_i} \right) \cdot \frac{dy_i}{dx} + \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \cdot \frac{d}{du} \left( \frac{dy_i}{dx} \right)
\]

yet, as \( \frac{\partial z}{\partial y_i} \) depends on the same variables as \( z \), by applying the first order multivariate chain rule with respect to \( u \), we get:

\[
\frac{d}{du} \left( \frac{\partial z}{\partial y_i} \right) = \sum_{j=1}^{n} \frac{\partial}{\partial y_j} \left( \frac{\partial z}{\partial y_i} \right) \cdot \frac{dy_j}{du} = \sum_{j=1}^{n} \frac{\partial^2 z}{\partial y_i \partial y_j} \cdot \frac{dy_j}{du}
\]

thus:

\[
\frac{d^2 z}{dx \, du} = \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{\partial^2 z}{\partial y_i \partial y_j} \cdot \frac{dy_i}{dx} \cdot \frac{dy_j}{du} + \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \cdot \frac{d^2 y_i}{dx \, du} \tag{2}
\]
2.2 Multivariate Chain Rule on a Computational DAG

Consider \( G = (V, E) \), a computational DAG of any abstraction level with \( V \) and \( E \) its vertices and edges.

Note that:
- A vertex \( v \in V \) directly depends on another vertex \( w \in V \) or, equivalently, \( w \) is a direct dependency of \( v \), if there is a directed edge in \( E \) pointing from \( w \) to \( v \)
- A vertex \( v \in V \) indirectly depends on another vertex \( w \in V \) or, equivalently, \( w \) is an indirect dependency of \( v \), if \( v \) is reachable from \( w \) with no directed edge in \( E \) pointing from \( w \) to \( v \)
- The indegree \( n_v \) of a vertex \( v \in V \) is the number of vertices on which \( v \) directly depends (if \( n_v = 0 \), then \( v \) is a source vertex)
- The inverse adjacency list of a vertex \( v \in V \) corresponds to the set of vertices on which \( v \) directly depends or, equivalently, to the list of direct dependencies of \( v \). Its cardinality is \( n_v \)

**Definition 1.** Let \( z \) be a vertex in a computational DAG \( G = (V, E) \), \( n_z \) be its indegree and \( (y_1, \ldots, y_{n_z}) \) be its inverse adjacency list. We call local partial derivatives of \( z \) the quantities:

\[
\left( \frac{\partial z}{\partial y_i} \right)_{\{y_j | j \in [1, n_z] \land j \neq i\}} \quad \text{or simply} \quad \frac{\partial z}{\partial y_i}, \quad i \in [1, n_z]
\]

If a vertex \( z \in V \) directly or indirectly depends on a vertex \( x \in V \) (i.e. if it exists a directed path from \( x \) to \( z \)), then a deviation \( \Delta x \) in \( x \) causes a deviation \( \Delta z \) in \( z \) [Bau74].

**Definition 2.** Let \( x \) and \( z \) be two vertices in a computational DAG \( G = (V, E) \). We call total derivative of \( z \) with respect to \( x \) the quantity:

\[
\frac{dz}{dx} = \lim_{\Delta x \to 0} \frac{\Delta z}{\Delta x}
\]

A fundamental property of total derivatives on computational DAGs is given by Bauer in [Bau74]:

**Property 1.** If a vertex \( z \in V \) does not directly nor indirectly depend on a vertex \( x \in V \) (i.e. if there is no directed path from \( x \) to \( z \)), then \( \frac{dz}{dx} = 0 \)

Even if this fundamental property sounds like an obvious result, it is important to note that it is specific to computational DAGs and not transposable to standard mathematics where equalities go both ways and functions can be implicitly defined.

A direct corollary of Property 1 is:

**Corollary 1.** If a vertex \( z \in V \) directly or indirectly depends on a distinct vertex \( x \in V \) then, as \( G \) is acyclic, \( x \) cannot in turn depend on \( z \) and therefore \( \frac{dx}{dz} = 0 \)

From Property 1 and assuming differentiability on all vertices, the first and second order multivariate chain rule can be extended to computational DAGs yielding Formula 3 and Formula 4.
At first order:

\[
\forall (z, x) \in V^2, \frac{dz}{dx} = \begin{cases} 
1 & \text{if } z = x \\
\sum_{i=1}^{n_z} \frac{\partial z}{\partial y_i} \cdot \frac{dy_i}{dx} & \text{if } z \neq x
\end{cases}
\]

or equivalently, applying Corollary 1 on the direct dependencies of \(x\) when \(z = x\):

\[
\forall (z, x) \in V^2, \frac{dz}{dx} = 1_{[z=x]} + \sum_{i=1}^{n_x} \frac{\partial z}{\partial y_i} \cdot \frac{dy_i}{dx}
\]  

(3)

At second order:

\[
\forall (z, x, u) \in V^3, \frac{d^2z}{dx \, du} = \sum_{i=1}^{n_z} \sum_{j=1}^{n_z} \frac{\partial^2 z}{\partial y_i \partial y_j} \cdot \frac{dy_i}{dx} \cdot \frac{dy_j}{du} + \sum_{i=1}^{n_x} \frac{\partial z}{\partial y_i} \cdot \frac{d^2 y_i}{dx \, du}
\]  

(4)

2.3 Recursive Definition of Total Derivatives on a Computational DAG

As highlighted in Formula 5, Formulas 3 and 4 exhibit a joint-multivariate recursive definition of all first and second order total derivatives in a computational DAG \(G = (V, E)\) equipped with some topological ordering:

If we know the first and second order total derivatives of all the direct dependencies \((y_1, \ldots, y_{n_z})\) of a vertex \(z\) with respect to all vertices in \(V\) and pairs of vertices in \(V^2\):

\[
\left\{ \frac{dy_i}{dx} \mid x \in V \land i \in [1, n_z] \right\} \cup \left\{ \frac{d^2 y_i}{dx \, du} \mid (x, u) \in V^2 \land i \in [1, n_z] \right\}
\]

then it is possible to combine them with the first and second order local partial derivatives of \(z\):

\[
\left\{ \frac{\partial z}{\partial y_i} \mid i \in [1, n_z] \right\} \cup \left\{ \frac{\partial^2 z}{\partial y_i \partial y_j} \mid (i, j) \in [1, n_z]^2 \right\}
\]

to get the first and second order total derivatives of \(z\) with respect to all vertices in \(V\) and pairs of vertices in \(V^2\):

\[
\left\{ \frac{dz}{dx} \mid x \in V \right\} \cup \left\{ \frac{d^2 z}{dx \, du} \mid (x, u) \in V^2 \right\}
\]

The recursive definition is seeded on all source vertices \(s\) in \(V\) (where \(n_s = 0\)), by Formulas 3 and 4 that reduce to:

\[
\forall x \in V, \frac{ds}{dx} = 1_{[s=x]} \quad \text{and} \quad \forall (x, u) \in V^2, \frac{d^2 s}{dx \, du} = 0.
\]
3 CGAD

CGAD is a framework that exploits Formula 5 on the high-level domain model computational DAGs of the software industry.

All nodes are solely responsible for computing local partial derivatives with respect to their direct dependencies while total derivatives computation is delegated to a Multivariate Chain Rule Engine (MCRE) as illustrated in Formula 6.

To compute first and second order total derivatives, the MCRE does not require more than the natural topological ordering traversal of the computational DAG that computes values.
3.1 CGAD Software Engineering Advantages

The sharp division of roles between local and global computations in CGAD is a structuring separation of concerns [Dij82] that offers several key software engineering advantages:

3.1.1 Hybrid Differentiation

The encapsulation of local partial derivative computation methods inside each node allows hybrid differentiation i.e. connecting derivatives from different sources using different calculation methods and different languages in the same evaluation. Each node’s implementer can independently choose the most appropriate method to generate its local partial derivatives: symbolic differentiation, finite difference, AD, or even CGAD at a finer local grain.

3.1.2 Extensibility

Integrating CGAD in software makes the system inherently extensible. The code is made of self-contained blocks: the nodes, that encapsulate their local partial derivatives computation. The nodes are coded once and can be composed afterwards in any way. Any new node composition gives a new computational DAG at runtime, whose total derivatives are automatically calculated by the MCRE.

3.1.3 Flexibility

The MCRE can work on any abstraction level computational DAG. However, as stated in [BH96], fine-grained computational DAGs (at the programming language level) do not scale and become impractical on large programs. Conversely, excessively increasing the grain sizes leads to code complexity that goes beyond the mathematical and software engineering skills of the developers (i.e. the code complexity they are able to write, test, debug and maintain).

CGAD offers the flexibility to chose the most appropriate granularity to the context. This choice will typically be a compromise between the scalability and integrability of coarse grains on the one hand and the decoupling and code simplicity of fine grains on the other hand.

3.1.4 Integrability

CGAD offers two integration advantages:

1. At a global scale, the hybrid characteristic of CGAD enables transparent integration of different computation modules.
2. At a local scale, the flexible characteristic of CGAD enables cost-effective top-down integration inside a computation module. This is a competitive advantage compared to fine-grained AD that requires time-consuming bottom-up integration [Hen17].
3.1.5 Decoupling

CGAD brings to light domain model computational DAGs. This strengthens software modularity and clears up module dependencies. By delegating global computations to a separated MCRE, it allows full decoupling between the implementation modules, that only perform local computations.

3.1.6 Common Language

Domain Driven Design (DDD) is a software development approach that is well adapted to the complex domains of the software industry. It leads developers and domain experts to build (and constantly use) a common language describing the domain model called the “Ubiquitous Language” [Eva04].

CGAD naturally spurs developers and domain experts to work together in DDD mode: It supports the domain model and the “Ubiquitous Language” by placing domain model computational DAGs at the heart of the development process: in specifications, code and results.

3.2 CGAD Compelling Combinations

As mentioned in Section 3.1.1, the hybrid characteristic of CGAD allows any combination of local partial derivatives computation methods in the same evaluation. However combinations focused on symbolic differentiation and AD have very appealing arguments for user experience: precision and speed.

3.2.1 Precision

An interesting combination enabled by CGAD is to get all nodes appearing in a computational DAG to use only symbolic differentiation or AD to compute their local partial derivatives. In such a situation, each node provides the MCRE with local partial derivatives that are exact up to machine precision. The associativity and commutativity of the summations in Formula 6 ensures that the total derivatives computed by the MCRE are also exact up to machine precision.

3.2.2 Speed

As shown in the implementation described in Section 4, CGAD can outperform AD and finite difference by doing local symbolic differentiation inside all nodes of a computational DAG. The substantial performance gain typically comes from:

- Mathematical Simplification

Mathematical expressions of local partial derivatives can comprise significant simplifications that are well exploited by computer algebra systems. Symbolic differentiation, unlike finite
difference and fine-grained AD [GW08], can exploit those simplifications to significantly reduce the algorithmic complexity of local partial derivative calculations [Hen11].

- **Implicit Function Differentiation**

A node in a computational DAG can depend on other nodes through a set of implicit equations. Those implicit equations are usually solved by an iterative method that can be time-consuming. In such a case, the implicit function theorem gives the explicit expressions of the node’s first order local partial derivatives which can, in turn, be differentiated explicitly [Kap52] to get its second order local partial derivatives. This method is considerably faster than fine-grained AD, that differentiates inside the solver, and finite difference, that runs the time-consuming solver several times [Hen11].

- **Local Computation Reuse**

Symbolic differentiation makes it possible to fruitfully reuse local variables of the value computation. This can reduce the added complexity of the local partial derivatives computation up to a fraction of the node’s value complexity. As a basic illustration of reuse, if the node’s value is $z = x^n$, a $O(\log(n))$ operation, then $\frac{\partial z}{\partial x} = nx^{n-1}$ and $\frac{\partial^2 z}{\partial x^2} = n(n-1)x^{n-2}$ may still appear as having $O(\log(n))$ runtime complexity. Since $z$ is known, however, the derivatives can be much more efficiently computed as $\frac{\partial z}{\partial x} = \frac{n}{x} \cdot z$ and $\frac{\partial^2 z}{\partial x^2} = \frac{n-1}{x} \cdot \frac{\partial z}{\partial x}$, reducing complexity to $O(1)$.

### 3.3 CGAD Optimization Options

It is possible to increase CGAD algorithmic efficiency by exploiting parallelism in Formula 6, total derivatives sparsity and lazy initialization.

Those three optimizations are used in the financial Java implementation described in Section 4.

#### 3.3.1 Parallelism

The local partial derivatives computations done in the nodes do not depend on the total derivatives computations done by the MCRE. Thus, the MCRE can be executed in a parallel computation pipeline acting as a transparent consumer on the local partial derivatives and following a similar topological ordering traversal of the computational DAG.

#### 3.3.2 Sparsity

Given the sparse characteristic of total derivatives in a computational DAG resulting from Property 1 and Corollary 1, CGAD can gain in efficiency by maintaining only non-null total derivatives.

In such case, the MCRE performs two separate steps on each node $z$ during the topological ordering traversal of the computational DAG, as highlighted in Formula 7:

1. A creation step in which new first and second order total derivatives are created on $z$
2. A composition step in which the total derivatives of \( z \)'s direct dependencies \( (y_1, \ldots, y_n) \) are multiplied by \( z \)'s corresponding first order local partial derivatives.

The set of non-null total derivatives is then gradually enriched by the MCRE on each node along the topological ordering traversal of the computational DAG.

### 3.3.3 Lazy Initialization

In practice, users are often interested in a subset of all possible non-null total derivatives. For instance, in software with a graphical user interface, the subset of total derivatives to display might depend on the view chosen by the user.

CGAD can take advantage of such a situation by implementing the lazy initialization idiom [Mar08] on total derivatives:

1. The user specifies, at runtime, a set of first and second order total derivatives creation criteria
2. Those criteria are checked on each node during the computational DAG traversal
3. The total derivatives of the node are only created if they match the criteria

Lazy initialization of total derivatives is possible as the recursive definition exhibited in Formula 5 can be narrowed to first order total derivatives with respect to all nodes in any subset \( D_1 \subset V \) and second order total derivatives with respect to all pairs of nodes in any subset \( D_2 \subset D_1^2 \), as shown in Formula 7.

\[
\forall z \in V
\forall x \in D_1 \subset V
\forall (x, u) \in D_2 \subset D_1^2
\]

\[
\begin{align*}
\frac{dz}{dx} &= 1_{\{z=x\}} + \sum_{i=1}^{n_z} \sum_{j=1}^{n_z} \frac{\partial^2 z}{\partial y_i \partial y_j} \frac{dy_i}{dx} \frac{dy_j}{du} \\
\frac{d^2 z}{dx \, du} &= \sum_{i=1}^{n_z} \frac{\partial z}{\partial y_i} \frac{dy_i}{dx} + \sum_{i=1}^{n_z} \frac{\partial z}{\partial y_i} \frac{d^2 y_i}{dx \, du}
\end{align*}
\]

(7)

### 4 Implementation in Financial Software

#### 4.1 Context

In this section we see how various requirements of the finance industry induce strong constraints on computational DAGs and total derivatives computation for software builders willing to propose fast evaluations and good time to market.
4.1.1 Pricing

In a pricing screen, the financial product and the evaluation parameters are defined at runtime in reaction to the user input. The corresponding computational DAG cannot remain static and should permit dynamic modifications.

![Dynamic computational DAG](image)

*Figure 9: Dynamic computational DAG*

4.1.2 Portfolio management

A financial product is not evaluated in isolation. Financial product transactions are gathered in a portfolio and evaluated together in a portfolio management screen. The different products in the portfolio typically share common features and market data that might take time to compute and calibrate. Significant performance gains can be achieved by allowing the corresponding computational DAGs to share common nodes resulting in one multi-sinked DAG.

![Multi-sinked computational DAG](image)

*Figure 10: Multi-sinked computational DAG*

4.1.3 Continuous Evaluations

A financial product is not evaluated only once. A portfolio management screen is continuously updated by real time market data feeds and user scenarios. Significant performance gains can be achieved by keeping the computational DAG in memory and traversing it again when the source node values change; either in response to real time feed events or simulation scenarios.
4.1.4 Proxy Evaluations

Continuous evaluation in a portfolio management screen can be impossible to put in place if the computation time is bigger than the expected refresh cycle. A pragmatic trade-off used by finance practitioners is to set a longer full evaluation refresh cycle in addition to a faster but less accurate approximation when underlying parameters change. These faster computations typically use a Taylor series expansion that requires fast and exact first and second order total derivatives. Such fast proxy evaluations are also used by finance practitioners on simulation scenarios.

\[
A_2 \approx A_1 + \frac{dA_1}{dC_1} \cdot (C_2 - C_1) + \frac{1}{2} \cdot \frac{d^2A_1}{dC_1^2} \cdot (C_2 - C_1)^2
\]

Figure 12: Proxy evaluation

4.1.5 Intermediary Risks

Finance practitioners sometimes hedge with derivative products that embed several risk factors. For instance, it is possible to hedge an option with a forward to simultaneously get rid of spot risk and dividend risk. The number of forward contracts to buy or sell is given by the total derivatives of the option node with respect to the forward node (which is an internal node in the computational DAG). It should therefore be possible to compute total derivatives with respect to internal nodes in the computational DAG.
4.1.6 New Financial Products

The finance industry constantly creates new products based on existing ones. For instance, options on futures, options on swaps or swaps on option payoffs. They are called derivatives as their values derives from the value of the underlying products on which they are built. The computational DAGs of those new products are built upon the computational DAGs of their underlyings. Consequently, the sinks of today’s computational DAGs should not be frozen as they can become internal nodes of tomorrow’s computational DAGs.

4.1.7 Model Fine-tuning

When an old and faithful model has shown its limits in representing the reality of market behaviors, the finance industry has a pragmatic tendency to keep it and fine tune it by giving it additional degrees of freedom to stick to the market. For instance make the volatility of a stock depend on the option moneyness \( M = \frac{K}{S} \). The computational DAGs of those fine-tuned models contain additional nodes at their bottom, corresponding to the new degrees of freedom. Consequently the sources of today’s computational DAGs should not be frozen as they can become internal nodes of tomorrow’s computational DAGs.
4.1.8 Openness

Financial practitioners want to be able to plug their own local evaluation model inside a global evaluation framework. This implies that part of the computational DAG can be provided by an external library.

4.2 Specification

Our objective is to compute the premium and certain Greeks of a 18 months maturity, 34 EUR strike, European call option $C$ on a 35 EUR stock $S$ using the Black Formula [Bla76].

As mentioned in Section 1.2, the Black formula requires the stock volatility $V$ and a risk-free interest rate $Z$ as additional inputs.

The risk-free interest rate $Z$ is not directly observable. It is interpolated on a zero-coupon yield curve $CZ_i$ that is an implicit function of coupon-bearing instrument quoted rates $QR_i$:

| Instrument                              | Maturity | Rate  |
|-----------------------------------------|----------|-------|
| Loan                                    | O/N      | $QR_1$|
| Loan                                    | T/N      | $QR_2$|
| Loan                                    | 2M       | $QR_3$|
| Loan                                    | 3M       | $QR_4$|
| Short-Term Interest Rate Future         | DEC17    | $QR_5$|
| Interest Rate Swap                       | 3Y       | $QR_6$|

The volatility $V$ also is not directly observable. It is interpolated on a two-dimensional surface
of quoted volatilities $QV_i$ for different option moneyness and maturities:

|       | 1Y | 2Y |
|-------|----|----|
| $QV_1$ | 0.96 |    |
| $QV_2$ | 0.98 |    |
| $QV_3$ |    |    |
| $QV_4$ |    |    |

The user is interested in the 82 first and second order Greeks of Table 1:

| 14 first order Greeks | 68 second order Greeks |
|-----------------------|------------------------|
| 1 Delta $\frac{dC}{dS}$ | 1 Gamma $\frac{d^2C}{dS^2}$ |
| 1 Vega $\frac{dC}{d\sqrt{V}}$ | 2 Vannas $\frac{d^2C}{dS \sqrt{V}}$, $\frac{d^2C}{d\sqrt{V} dS}$ |
| 4 Implied Volatility Vegas $\frac{dC}{dQV_i}$ | 1 Volga $\frac{d^2C}{d\sqrt{V}^2}$ |
| 2 Calibrated Zero-Coupons Sensitivities $\frac{dC}{dCZ_i}$ | 8 Implied Volatility Vannas $\frac{d^2C}{dS dQV_i}$, $\frac{d^2C}{dQV_i dS}$ |
| 6 Market Rate Sensitivities $\frac{dC}{dQR_i}$ | 16 Implied Volatility Volgas $\frac{d^2C}{dQV_i dQV_j}$ |
|                        | 4 Calibrated Zero-Coupons Convexities $\frac{d^2C}{dCZ_i dCZ_j}$ |
|                        | 36 Market Rate Convexities $\frac{d^2C}{dQR_i dQR_j}$ |

Table 1: Requested Greeks

4.3 Implementation

We describe a CGAD implementation written in Java, that satisfies the constraints on computational DAGs and total derivatives computation of Section 4.1. Our description uses standard programming constructs found in mainstream object oriented languages for easy integration in a wide variety of existing codebases.

4.3.1 Framework Design

As illustrated in the UML class diagram of Figure 17, the CGAD framework consists of two classes: an abstract NodeCalculator that all nodes in the computational DAG have to implement and a DerivativesController.
• The computational DAG is implemented in the NodeCalculator class. Each NodeCalculator object stores its own inverse adjacency list, which represents its direct dependencies in the global computation.

• The MCRE is implemented in the abstract NodeCalculator class using the NVI (Non-Virtual Interface) design pattern [Sut05] which is an extension of the template method design pattern [GHJV95]. This pattern decouples the MCRE from the different node implementations, enabling both to evolve independently. The NodeCalculator class exhibits a public client interface that triggers the global computation and an independent private contract that defines the local computations to be implemented in each node.

• Local partial derivatives and total derivatives are maintained separately by the NodeCalculator class. The symmetry of second order derivatives is exploited both on local partial derivatives and on total derivatives. To deal with sparsity and dynamic creation criteria, they are stored as maps (associative arrays). The keys in the maps are based on the nodes in the computational DAG: $x$ representing the $\frac{\partial}{\partial x}$ and $\frac{d}{dx}$ first order operators and $(x, u)$ representing the $\frac{\partial^2}{\partial x \partial u}$ and $\frac{d^2}{dx du}$ second order operators. The values in the maps are double-precision numbers representing the derivative values.

• The DerivativesController class maintains total derivatives creation criteria set by the client. Those criteria are checked by the MCRE during the computational DAG traversal to determine which first and second order total derivatives should be created.

4.3.2 Algorithm

When a client triggers the global computation, the computational DAG is traversed in topological ordering. On each node, the MCRE template method splits the computation algorithm in two steps: a local polymorphic step done by the node implementation and a global invariant step done by the MCRE.
• **Node Local Polymorphic Step**

The node is asked by the framework to compute its value and local partial derivatives. The traversal in topological ordering of the computational DAG ensures that all its direct dependencies are already computed.

• **MCRE Global Invariant Step**

After each node local polymorphic step, the MCRE computes the node’s total derivatives according to Formula 7. It proceeds in three stages:

1. It creates the node’s canonical first order total derivative, if the DerivativesController instructs to
2. It does the Cartesian product of the node’s direct dependencies first order total derivatives and creates the second order total derivatives that the DerivativesController instructs to
3. It composes the first and second order total derivatives of the node’s direct dependencies by multiplying them by the node’s corresponding first order local partial derivatives

Throughout this step, the MCRE gathers the different contributions to each total derivative, by summing values on the same key in the node’s total derivatives map.

A first implementation of the algorithm executes the MCRE step sequentially after each node valuation during the computation DAG traversal. A second implementation performs the MCRE step in parallel of the other nodes valuations, as seen in Section 3.3.1.

4.3.3 **Node Implementations**

To cope with the various constraints of Section 4.1, our software architecture is composed of a Quotes module, a Rate Curve module, a Volatility Surface module, a Forward Curve module and an Option module, as shown in the UML package diagram of Figure 18.

![Figure 18: Architecture UML package diagram](image-url)
Those modules contain the implementation of self-contained nodes from the domain model that encapsulate their local partial derivatives computation, as explained in Section 3.1.2. All nodes provide their local partial derivatives very efficiently, taking advantage of local symbolic differentiation through implicit function theorem, mathematical simplifications and local computation reuses, as described in Section 3.2.2.

4.4 Results

A DerivativesController is instantiated and filled with the first and second order Greeks creation criteria of Table 1. Then the option premium and Greeks computation is triggered.

The corresponding runtime high-level domain model computational DAG is represented in Figure 19 where the 25 nodes are colored according to the module to which they belong:

![Figure 19: Runtime global computational DAG](image)

The topological ordering traversal of the computational DAG is seeded on the source nodes from the Quotes module that are initialized with market quotes. It crosses all module boundaries and terminates on the $C$ node from the Option module giving a 7.270 premium. It is displayed with the local results of each node (values and local partial derivatives) in Table 2:
The MCRE composes the 94 local partial derivatives computed by the nodes into the 82 Greeks in Table 3 (with values not requested to the DerivativesController labeled as “n.r.”):
### 4.5 Performance

The performance measurements were run on a Linux (4.4.0-53-generic x86_64) PC equipped with an Intel Core i7-6700 processor (4 cores, 3.8 GHz), 32 GB RAM, running Java 1.8.0_111.

The standard run was an evaluation of 2000 options.

To separately measure the premiums, local partial derivatives and total derivatives computation times, the runs were launched in four contexts:

1. Only computing the option premiums
2. Computing the option premiums and the 94 local partial derivatives of all the nodes in the computational DAGs
3. Computing the option premiums, the 94 local partial derivatives and the 82 requested first and second order total derivatives
4. Computing the option premiums, the 94 local partial derivatives and the 82 total derivatives in parallel mode as described in Section 3.3.1

The retained measures for each context are the averages on 20 standard runs:

| Context | Time   | Ratio |
|---------|--------|-------|
| Premiums| 118 ms | (1.0×) |
| Premiums + Local Partial Derivatives| 206 ms | (1.7×) |
| Premiums + Local Partial Derivatives + Total Derivatives| 308 ms | (2.7×) |
| Premiums + Local Partial Derivatives + Total Derivatives (in parallel mode)| 262 ms | (2.2×) |

*Table 4: Time to price 2000 options*
We can draw several interesting conclusions from the timings in Table 4:

First, the 94 local partial derivatives of all the nodes in the computational DAGs were computed with a ratio of $1 : 1.7$ to the premiums-only computation time, i.e. only increasing the computation time by 70%. This result illustrates the unrivalled speed of local symbolic differentiation described in Section 3.2.2

Second, the 82 requested first and second order total derivatives were computed with a ratio of $1 : 2.7$ to the premiums-only computation time. This represents an overhead of 50% with respect to the computation time of the premiums and local partial derivatives. It illustrates CGAD efficiency, that comes from three factors:

1. working on coarse-grained computational DAGs
2. exploiting total derivatives sparsity, as described in Section 3.3.2
3. implementing lazy initialization on total derivatives, as described in Section 3.3.3

Finally, if the MCRE runs in a dedicated separated thread, as introduced in Section 3.3.1, 50% of its overhead can be masked. This yields to a computation of the 82 requested first and second order total derivatives with a time ratio of $1 : 2.2$ to the premiums-only computation time.

As a comparison, a finite difference engine would roughly exhibit a quadratic ratio of $1 : 56$ on the same example. Indeed, it would require 1 evaluation for the premiums, 14 for the 14 first order Greeks and 41 for the 68 second order Greeks, exploiting symmetry.

In AD, the adjoint mode would compute the 14 first order Greeks of this example with a constant ratio around $1 : 3$. However, the second order Greeks would require another AD pass on those 14 outputs, typically run in forward mode with a linear ratio around $1 : 1.5 \times 14$. The AD global ratio for the first and second order Greeks would therefore be linear and around $1 : 63$.

So, on this example, our CGAD implementation largely outperforms alternatives such as AD of finite difference.

5 Higher Order

Faà di Bruno formula [FdB85] ensures that the joint-multivariate recursive definition of all total derivatives in a computational DAG exhibited in Formula 6 is not particular to first and second order and can be extended to higher orders:

If a variable $z$ depends on $n_z$ variables $y_1, \ldots, y_{n_z}$ and if all the up to $m^{th}$ order total derivatives of all the $y_i$ are known, it is possible to combine them with the up to $m^{th}$ order local partial derivatives of $z$ to get all the up to $m^{th}$ order total derivatives of $z$.

The nodes in the computational DAGs just have to provide their local partial derivatives up to $m^{th}$ order and the MCRE has to be updated accordingly.

For instance at order 3, this yields Formula 8:
\[ \forall (z, x, u, v) \in V^4, \quad \frac{d^3 z}{dx \, du \, dv} = \sum_{i=1}^{n_z} \sum_{j=1}^{n_z} \sum_{k=1}^{n_z} \frac{\partial^3 z}{\partial y_i \partial y_j \partial y_k} \cdot \frac{dy_i}{dx} \cdot \frac{dy_j}{du} \cdot \frac{dy_k}{dv} \]
\[ + \sum_{i=1}^{n_z} \sum_{j=1}^{n_z} \frac{\partial^2 z}{\partial y_i \partial y_j} \left( \frac{d^2 y_i}{dx \, du} \cdot \frac{dy_j}{dv} + \frac{d^2 y_i}{dx \, dv} \cdot \frac{dy_j}{du} + \frac{d^2 y_i}{du \, dv} \cdot \frac{dy_j}{dx} \right) \]
\[ + \sum_{i=1}^{n_z} \frac{\partial z}{\partial y_i} \cdot \frac{d^3 y_i}{dx \, du \, dv} \]  
(8)

**Conclusion**

CGAD offers a new way to compute first and second order total derivatives that fully addresses software engineering concerns. It induces no coupling in software, letting all modules independently chose their local partial derivatives computation methods and connecting them using the multivariate chain rule. Additionally, its hybrid and flexible characteristics enables a cost-effective top-down integration.

Moreover, when combined with local symbolic differentiation, CGAD also addresses user experience concerns as it becomes exact up to machine precision and runs significantly faster than AD and finite difference in certain contexts.

**References**

[Bau74] F. L. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

[BH96] Christian H Bischof and Mohammad R Haghighat. Hierarchical approaches to automatic differentiation. *Computational Differentiation: Techniques, Applications, and Tools*, pages 83–94, 1996.

[Bla76] Fischer Black. The pricing of commodity contracts. *Journal of Financial Economics*, 3(1-2):167–179, 1976.

[DG17] H. O. Duché and F. Galilée. Second order differentiation formula in a computational graph. 2017. Available at SSRN: https://ssrn.com/abstract=2905103.

[Dij82] Edsger W Dijkstra. On the role of scientific thought. In *Selected writings on computing: a personal perspective*, pages 60–66. Springer, 1982.

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[FdB85] Francesco Faà di Bruno. Sullo sviluppo delle funzioni. *Annali di scienze matematiche e fisiche*, 6:479, 1885.

[Fow02] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[GG06] Mike Giles and Paul Glasserman. Smoking adjoints: Fast monte carlo greeks. *Risk*, 19(1):88–92, 2006.
[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GNC07] D Gendler, U Naumann, and B Christianson. Automatic differentiation of assembler code. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 431–436, 2007.

[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.

[Hen11] Marc Henrard. Adjoint algorithmic differentiation: Calibration and implicit function theorem. *OpenGamma Quantitative Research*, (1), 2011.

[Hen17] Marc Henrard. *Algorithmic Differentiation in Finance Explained*. Springer, 2017.

[Kap52] W. Kaplan. *Advanced Calculus*. Addison-Wesley mathematical series. Addison-Wesley Press, 1952.

[Mar08] Robert C. Martin. Clean code: A handbook of agile software craftsmanship, 2008.

[Nau12] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Number 24 in Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

[Sut05] Herb Sutter. *Exceptional C++ style: 40 new engineering puzzles, programming problems, and solutions*. Addison-Wesley, 2005.