CALCULATING KEY RATIOS FOR FINANCIAL PRODUCTS USING AUTOMATIC DIFFERENTIATION AND MONTE CARLO SIMULATION

ESBEN BISTRUP HALVORSEN

Abstract. A library for higher order, multivariate automatic differentiation is combined with a library for Monte Carlo simulation, and the theoretical background, implementation, usage and performance are discussed.

Contents

1. Introduction

There is an increasing need for computation in the world of finance. Not only is the number of financial products continuously growing, but so is the need to forecast their values into the future. Key ratios and simulations of future developments can provide the insight needed to make the right decisions, and enormous computational resources are devoted to calculating key ratios and simulating scenarios as quickly and precisely as possible.

This project implements and discusses a framework for automatic differentiation of mathematical functions, including Monte Carlo simulated functions. The framework allows key ratios to be computed accurately rather than approximated by numerical differentiation (shocking), which is customary in the financial sector. The integration with Monte Carlo simulated functions makes the framework useful for the computation of key ratios, since many pricing functions do not have a closed formula but must be simulated.

The implementation is made in Haskell by overloading functions. This means that it is not necessary to modify any existing mathematical functions in order to obtain their derivatives. Said differently, the framework can be considered a plug-in that provides extra functionality for computing derivatives of existing functions. In addition, the implementations of automatic differentiation and Monte Carlo simulation have been kept apart so that, if desired, one can use one without the other.

Date: December 12, 2012.
This project was completed under the auspices of the HIPERFIT group at DIKU with Martin Elsman as supervisor.

Performance and completeness have not been essential in this project. In contrast, the project should only be seen as a first step towards a complete framework for calculating key ratios for financial products. The discussion part of this paper presents ideas for additional steps in this direction.

The main part of the project is about automatic differentiation, since the integration with Monte Carlo simulation is almost a formality. We have used Conal Elliot's *beautiful differentiation* [3] as the starting point for the implementation. Unfortunately, Elliot's description is incomplete, and his own implementations in [5] are either flawed or quite far from the original paper, so we have made a number of adjustments to make things work. As it turned out, the resulting code actually became simpler from the corrections.

1.1. **Organization.**

The paper is organized as follows. Section 2 gives a short presentation of Monte Carlo simulated functions. Section 3 introduces higher order derivatives for multivariate functions as linear maps and discusses some of the differentiation rules. Section 4 discusses various techniques for implementing differentiation in practice, including two versions of automatic differentiation. Section 5 describes the Haskell code which constitutes the implementation part of this project and gives a few examples of how to use the implementation. Finally, Section 6 discusses correctness, performance and various other aspects of the implementation and gives suggestions for future development.

1.2. **Contributions.**

The main contributions of this project are:

- An implementation of automatic differentiation combined with Monte Carlo simulation.
- Corrections and simplifications of Conal Elliot's implementations [5] of *Beautiful differentiation* [3], making the implementation both correct and true to the original article; see the discussion in Section 6.1.
- Generalizations (6.1) and (6.2) of the chain rule in a formulation with derivatives as linear maps together with a complexity analysis and suggestions for performance improvements.

1.3. **Related work.**

The subject of automatic differentiation has been discussed for decades and is treated in an abundance of literature. Wengert [12] introduced the forward and backward accumulation techniques (see Section 4.3) all the way back in 1964, and Iverson [6] implemented a derivative operator for the APL language back in 1979. Karczmarczuk [8] came up with the idea of a lazy derivative towers for higher order derivatives in 2001, but restricted to the one-dimensional case. Pearlmutter [11] showed in 2007 how to handle higher order derivatives for multivariate functions using dual numbers (see Section 4.4), and Elliot [3] extended Karczmaczuk's derivative towers in 2009 to the multivariate case using linear maps over general vectors spaces. Among other interesting literature on the subject, we find the collection [1] of articles from the proceedings of the SIAM Workshop on the Automatic Differentiation of Algorithms in 1996, and some recent books by Griewank and Walther [4] and Naumann [10]. The web site `http://www.autodiff.org` is devoted to automatice differentiation and contains a long list of additional literature on the subject. No literature that we are aware of combines multivariate, higher order automatic differentiation with Monte Carlo simulation.

1.4. **Prerequisites.**

The reader is assumed to be knowledgeable about basic linear algebra, calculus and probability theory. This includes norms and linear maps of finite-dimensional $\mathbb{R}$-vector spaces; higher order differentiation and integration of multivariate functions; and probability distributions and expected values. The reader is also assumed to be familiar with the standard features in Haskell and have a good understanding of Haskell's type system.

1.5. **Notation.** Throughout this text, $\mathbb{R}$ denotes the set of real numbers, and $\mathbb{R}^k$ denotes the set of k-tuples from $\mathbb{R}$ considered as a k-dimensional, normed vector space over $\mathbb{R}$. We will generally use the letters U, V and W to denote $\mathbb{R}$-vector spaces of finite dimension. A map f from a set A to a set B is denoted $f\colon A \to B$. When dealing with vector spaces, we write $U \multimap V$ in place of $U \to V$ to emphasize that the map is linear. Such a map can be represented by a matrix, and we will freely interchange a linear map with its matrix representation. In particular, we will sometimes write fx and gf in place of $f(x)$ and $g \circ f$, respectively, when applying a linear map $f\colon U \multimap V$ to a vector $x \in U$ or composing it with another linear map $g\colon V \multimap W$. This is to emphasize that applications and compositions of linear maps are obtained by multiplications of matrices by vectors or other matrices.

2. Monte Carlo simulation

There is no general consensus in the literature on what exactly defines the term "Monte Carlo". Some sources use the term for any simulation procedure that involves taking (independent) samples from a probability distribution and using them to obtain an estimate for something that cannot otherwise be found with a closed formula. Other sources denote such procedures by "stochastic simulation" and reserve the word "Monto Carlo" to the special case in which the aim of the stochastic simulation is to find an integral of a mathematical function or an expected value of a probability distribution. See [18] for further details.

In this project we will consider *Monte Carlo simulated* functions: that is, mathematical functions whose values cannot be found by a closed formula but must be calculated as the average of a collection of samples. Many examples of functions with no closed formula arise when taking integrals: the integral of a function does not always have a closed formula even though the function itself does. Using Monte Carlo simulation to find integrals is a typical scenario in the world of finance.

Suppose that $X_1, \ldots, X_N$ are independent and identically distributed random variables drawn from some distribution with expected value μ and variance σ^2. The *law of large numbers* says that the sample average

$$S_N = \frac{1}{N} \sum_{i=1}^{N} X_i$$

converges in probability (and almost surely) to μ as N tends to infinity. The *central limit theorem* states that the distribution of S_N as N grows actually approximates a normal distribution with mean value μ and variance σ^2/N. This implies that S_N displays a $1/\sqrt{N}$ convergence, meaning that a quadrupling of N halves the error of the approximation to μ. This is not a very good convergence ratio: it takes a lot of samples to get precise results! The *curse of dimensionality* makes this problem even more apparent in higher dimensions. In some situations, it may therefore be better to use random variables that are dependent or even not random at all; see the discussion in Section 6.3 on page 28.

Now, suppose that we want to simulate a function $f(x)$. We do not have a closed formula for $f(x)$, but we can simulate a distribution whose mean value is exactly equal to $f(x)$. Computing $f(x)$ can therefore be done by drawing samples from this distribution and taking the average. The aforementioned results ensure that we can use the average to approximate the function value and that we can compute the number of samples needed to obtain a sufficiently small error.

The typical way to simulate a probability distribution is by drawing samples from a simple and well-known probability distribution, for example the uniform distribution on $[0;1]$, and transforming the samples using a known function. Thus, we can approximate a function value $f(x)$ by a formula in the form

$$f(x) \approx \frac{1}{N}\sum_{i=1}^{N} \hat{f}(x, w_i), \tag{2.1}$$

where the w_i are drawn independently from a simple and well-known probability distribution, which is transformed by the function $\hat{f}(x,-)$ to a distribution with mean value $f(x)$.

As an example, consider the function $f\colon [a;b] \to \mathbb{R}$ defined as an integral of another function:

$$f(x) = \int_a^x g(t)\mathrm{d}t$$

We can approximate $f(x)$ by drawing samples w_i from the uniform distribution on $[0;1]$ and using (2.1) with

$$\hat{f}(x, w_i) = (x-a)g(a + (x-a)w_i)$$

The intuition is that the $a + (x-a)w_i$ are uniformly distributed on the interval $[a;x]$, and that $\frac{1}{N}\sum_{i=1}^{N} g(a+(x-a)w_i)$ therefore is the average value of g on this interval, wherefore its product with the length $x-a$ of the interval is the area under the graph of g. This idea is known as the *sample mean method*

As another almost canonical example, consider the constant π. We can approximate π by drawing samples $w_i = (w_{i1}, w_{i2})$ from the uniform distribution on $[0,1] \times [0,1]$ and using (2.1) with

$$\hat{f}(w_i) = \begin{cases} 4, & \text{if } w_{i1}^2 + w_{i2}^2 < 1 \\ 0, & \text{else} \end{cases}$$

(Note that $\hat{f}$ does not depend on x since we are simulating a constant.) The intuition is as follows: the ratio of the area of a circle with a square that circumscribes it is $\pi/4$. Thus, if we throw darts at random against the square and compute the ratio of darts that fall inside the circle, this ratio will approximate $\pi/4$. Multiply by 4 to obtain π. (For simplicity, the example here only considers the top right corner of the unit circle .) This idea is know as the *hit or miss method* and can be used to compute the area of any irregular volume.

All in all, implementing a machinery to handle Monte Carlo simulated functions is quite easy: we just need to handle functions that are defined as in (2.1). Choosing the right Monte Carlo method and deciding how many samples to use to obtain the desired precision is a much harder task. We will not concern ourselves with this problem here but simply leave it to the user of the system.

3. Derivatives

The derivative of a function f at a point x is a measure for the local changes of function values $f(x)$ happening around x. More precisely, the derivative of f at x gives a *linear approximation* to f in a neighborhood of x. In order to have concepts such as “linearity“, “approximation” and “neighborhood” available, we restrict the functions under investigation to those of type $f\colon \mathbb{R}^k \to \mathbb{R}^\ell$. From a financial point of view, this is not a restriction at all, since any collection of key ratios we can think of can be described as a finite-dimensional vector over $\mathbb{R}$.

This section introduces derivatives for functions $f\colon \mathbb{R}^k \to \mathbb{R}^\ell$. The purpose is not to give a precise mathematical definition of derivatives but to strengthen the reader’s intuition for what derivatives really are, and how and why the derivation rules work. Having a clear understanding of the dual role of derivatives as numbers as well as of linear functions is important for a large part of this text.

3.1. **One-dimensional derivatives.**

The traditional way of describing the derivative of a simple, one-dimensional function $f\colon \mathbb{R} \to \mathbb{R}$ at a point $x \in \mathbb{R}$ is as a number $f'(x) \in \mathbb{R}$ which is the slope of the tangent line to the graph of f at x. This slope can be found as the limit of the slope of nearby secant lines:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.1}$$

When this limit exists, f is *differentiable* at x.

When f is differentiable at x, we can approximate the value of f at points in a neighborhood of x as

$$f(x+h) \approx f(x) + f'(x)h \tag{3.2}$$

where the approximation becomes better the smaller h is. For fixed x, we see that the approximation is linear in h. The term $f'(x)h$ indicates how much the function value changes, approximately, when the argument changes by h.

The idea that the derivative should be a number in $\mathbb{R}$ does not generalize to higher dimensions. A more general way to define the derivative of f at a point x is as a *linear map* $f'(x)\colon \mathbb{R} \multimap \mathbb{R}$ which takes $h \in \mathbb{R}$ as input and returns the number $f'(x)h \in \mathbb{R}$. With this interpretation, we should really write $f'(x)(h)$ rather than $f'(x)h$ to emphasize that $f'(x)$ is a function and not a number. But since linear maps can be represented by matrices, we can think of the term $f'(x)h$ as the multiplication of the 1×1-matrix $f'(x)$ by the one-dimensional vector h.

3.2. **Multivariate derivatives.**

The description of derivatives as linear maps can easily be extended to the multivariate case. Indeed, the approximation in (3.2) makes sense for a general $f\colon \mathbb{R}^k \to \mathbb{R}^\ell$ when we think of $f'(x)$ as a linear map $\mathbb{R}^k \multimap \mathbb{R}^\ell$ which we can apply to a vector $h \in \mathbb{R}^k$. Making this idea more precise, we define the derivative of f at x as a linear map $f'(x)\colon \mathbb{R}^k \multimap \mathbb{R}^\ell$ such that

$$\lim_{\|h\| \to 0} \frac{\|f(x+h) - f(x) - f'(x)h\|}{\|h\|} = 0 \tag{3.3}$$

where $\|\cdot\|$ denotes the usual Euclidean norm. If such a linear map $f'(x)$ exists, then it is unique, and we say that f is *differentiable* at x. Since linear maps from $\mathbb{R}^k$ to $\mathbb{R}^\ell$ can be represented by $\ell \times k$-matrices, we can think of $f'(x)$ both as a map and as a matrix. Represented as a matrix, $f'(x)$ is called the *Jacobian* of f at x. The entries of this matrix are the *partial derivatives* of f at x, and the (i,j)-entry is denoted $\partial f_i / \partial x_j$, where $f = (f_1, \ldots, f_\ell)$ are the coordinate functions of f and $x = (x_1, \ldots, x_k)$ are the coordinates of x.

The interpretation of (3.2) is clear, even in the multivariate case: if we are currently at the point x and know the function value $f(x)$, then moving in direction h (from x to $x+h$) leads to a change in function value which can be (linearly) approximated by $f'(x)h$.

As an example, consider the function $f\colon \mathbb{R}^2 \to \mathbb{R}^3$ given by

$$f(x_1, x_2) = (x_1 \sin x_2, x_1 x_2^2, 1 + x_2).$$

The derivative of f at a point $x = (x_1, x_2)$ is given by

$$f'(x_1, x_2) = \begin{pmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \\ \dfrac{\partial f_3}{\partial x_1} & \dfrac{\partial f_3}{\partial x_2} \end{pmatrix} = \begin{pmatrix} \sin x_2 & x_1 \cos x_2 \\ x_2^2 & 2x_1 x_2 \\ 0 & 1 \end{pmatrix}$$

which we can think of as representing a linear map $\mathbb{R}^2 \multimap \mathbb{R}^3$. Thus, if we move away from (x_1, x_2) in direction (h_1, h_2), the approximate change in f can be found as

$$\begin{pmatrix} \sin x_2 & x_1 \cos x_2 \\ x_2^2 & 2x_1x_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} = \begin{pmatrix} h_1 \sin x_2 + h_2 x_1 \cos x_2 \\ h_1 x_2^2 + h_2 2x_1x_2 \\ h_2 \end{pmatrix}$$

3.3. **Higher order derivatives.** If $f\colon U \to V$ is a function and the derivative $f'(x)$ of f at x is defined for all $x \in U$, we can define the general derivative f' of f as the map $f'\colon U \to (U \multimap V)$ which takes a vector $x \in U$ and returns the linear map $f'(x)\colon U \multimap V$. We can now ask for derivatives of the function f' and repeat the process. The second order derivative $f''(x)$, if it exists, should be a linear map $U \multimap (U \multimap V)$, which takes a vector $h \in U$ and gives a linear map $f''(x)h\colon U \multimap V$, which can be used as a linear approximation of $f'(x+h)$ in relation to $f'(x)$:

$$f'(x+h) \approx f'(x) + f''(x)h.$$

Since $U \simeq \mathbb{R}^k$ and $V \simeq \mathbb{R}^l$ for some k and ℓ, we can think of this approximation as taking place in the vector space of $\ell \times k$-matrices or, equivalently, the vector space of linear maps $U \multimap V$ whose vector space structure is induced by the vector space structure on V. One can think of $f''(x)$ either as a linear map $U \multimap (U \multimap V)$ or as a three-dimensional $\ell \times k \times k$-matrix. The general second order derivative is a map $f''\colon U \to (U \multimap (U \multimap V))$.

Repeating this indefinitely, we determine the types of all the higher order derivatives:

$$\begin{aligned} f\colon U &\to V \\ f'\colon U &\to (U \multimap V) \\ f''\colon U &\to (U \multimap (U \multimap V)) \\ f'''\colon U &\to (U \multimap (U \multimap (U \multimap V))) \\ &\vdots \end{aligned}$$

The interpretation is always the same: the n'th order derivative $f^{(n)}(x)$ at x is a linear map that can be used to approximate the local behavior of $f^{(n-1)}$ around x as

$$f^{(n-1)}(x+h) \approx f^{(n-1)}(x) + f^{(n)}(x)h,$$

where the approximation becomes better the smaller $\|h\|$ is.

Continuing the example from before with the function $f\colon \mathbb{R}^2 \to \mathbb{R}^3$, we saw that its derivative was a function $f'\colon \mathbb{R}^2 \to (\mathbb{R}^2 \multimap \mathbb{R}^3)$ given by

$$f'(x_1, x_2) = \begin{pmatrix} \sin x_2 & x_1 \cos x_2 \\ x_2^2 & 2x_1x_2 \\ 0 & 1 \end{pmatrix}.$$

The derivative of f' at the point (x_1, x_2) can be represented by a three-dimensional $3\times 2\times 2$-matrix. Splitting the matrix up into parts corresponding to the three coordinate functions f_1, f_2, f_3, we can depict the matrix by its three 2×2-submatrices:

$$\begin{pmatrix} \frac{\partial^2 f_1}{\partial x_1^2} & \frac{\partial^2 f_1}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f_1}{\partial x_2 \partial x_1} & \frac{\partial^2 f_1}{\partial x_2 \partial x_2} \end{pmatrix} = \begin{pmatrix} 0 & \cos x_2 \\ \cos x_2 & -x_1 \sin x_2 \end{pmatrix}$$

$$\begin{pmatrix} \frac{\partial^2 f_2}{\partial x_1^2} & \frac{\partial^2 f_2}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f_2}{\partial x_2 \partial x_1} & \frac{\partial^2 f_2}{\partial x_2 \partial x_2} \end{pmatrix} = \begin{pmatrix} 0 & 2x_2 \\ 2x_2 & 2x_1 \end{pmatrix}$$

$$\begin{pmatrix} \frac{\partial^2 f_3}{\partial x_1^2} & \frac{\partial^2 f_3}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f_3}{\partial x_2 \partial x_1} & \frac{\partial^2 f_3}{\partial x_2 \partial x_2} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

When applying the $3\times 2\times 2$-matrix to a point (h_{11}, h_{12}), the result is a 3×2-matrix:

$$f''(x_1, x_2)\begin{pmatrix} h_{11} \\ h_{12} \end{pmatrix} = \begin{pmatrix} h_{12}\cos x_2 & h_{11}\cos x_2 - h_{12}x_1 \sin x_2 \\ h_{12}2x_2 & h_{11}2x_2 + h_{12}2x_1 \\ 0 & 0 \end{pmatrix}.$$

This matrix represents a change in $f'(x_1, x_2)$ when (x_1, x_2) is changed by (h_{11}, h_{12}) to $(x_1 + h_{11}, x_2 + h_{12})$. When applied to a direction (h_{21}, h_{22}) the result is a vector in $\mathbb{R}^3$ representing the change of the change in f when using the derivative $f'(x_1 + h_{11}, x_2 + h_{12})$ to approximate f compared to the change in f when using the derivative $f'(x_1, x_2)$ to approximate f:

$$f''(x_1, x_2)\begin{pmatrix} h_{11} \\ h_{12} \end{pmatrix}\begin{pmatrix} h_{21} \\ h_{22} \end{pmatrix} = \begin{pmatrix} h_{12}h_{21}\cos x_2 + h_{11}h_{22}\cos x_2 - h_{12}h_{22}x_1 \sin x_2 \\ h_{12}h_{21}2x_2 + h_{11}h_{22}2x_2 + h_{12}h_{22}2x_1 \\ 0 \end{pmatrix}.$$

We have now seen the definition of higher order derivatives for multivariate functions. In practice, one would only apply these definitions directly to a few, basic "building block" functions. For other and more complex functions, one would break down the function into simple components whose derivatives can be found, and then combine these simple derivatives into a derivative for the entire function. The remainder of this section discusses how this is possible using differentiation rules for sums, products and function composition.

3.4. **The sum rule.** Given two functions $f, g\colon U \to V$ their sum $f + g\colon U \to V$ is defined by $(f + g)(x) = f(x) + g(x)$. If f and g are both differentiable at a point x, then so is $f + g$. From the previous approximations, we see that, for $h \in U$,

$$\begin{aligned} (f+g)(x+h) &= f(x+h) + g(x+h) \\ &\approx f(x) + f'(x)h + g(x) + g'(x)h \\ &= (f+g)(x) + (f'+g')(x)h \end{aligned}$$

where the approximation becomes better the smaller h is. Thus, we have obtained the *sum rule*:

$$(f+g)' = f' + g'. \tag{3.4}$$

3.5. **The product rule.** Suppose that $*\colon U \times V \to W$ is a continuous, bi-linear function: that is, linear in both variables. If $U = V = W = \mathbb{R}$ this could, for example, be ordinary multiplication. Given two functions $f\colon A \to U$ and $g\colon A \to V$, we can define $f * g\colon A \to W$ by $(f * g)(x) = f(x) * g(x)$. If f and g are both differentiable at a point x, then so is $f * g$. From the previous approximations, we see that, for $h \in A$,

$$\begin{aligned}(f * g)(x+h) &= f(x+h) * g(x+h)\\ &\approx (f(x) + f'(x)h) * (g(x) + g'(x)h)\\ &= f(x) * g(x) + f(x) * g'(x)h +\\ &\qquad f'(x)h * g(x) + f'(x)h * g'(x)h\\ &= (f * g)(x) + (f(x) * g'(x) + f'(x) * g(x))h + f'(x)h * g'(x)h\end{aligned}$$

where the approximation becomes better the smaller $\|h\|$ is. Here, we have exploited continuity of $*$ to get the approximation, and we have overloaded the $*$ symbol by thinking of $f(x)$ and $g(x)$ as constant functions $A \to U$ and $A \to V$, respectively. Since $f'(x)h * g'(x)h$ has two occurrences of h, the limit for this term divided by $\|h\|$ will tend to 0 as $\|h\|$ tends to 0. Consequently we can (and should) ignore this term, and so we obtain

$$(f * g)(x+h) \approx (f * g)(x) + (f(x) * g'(x) + f'(x) * g(x))h$$

where the approximation becomes better the smaller $\|h\|$ is. Thus, thinking of f and g as functions $A \to (A \to U)$ and $A \to (A \to V)$, respectively, where the last argument from A is ignored, we have

$$(f * g)' = f * g' + f' * g \tag{3.5}$$

When "$*$" denotes ordinary multiplication on $\mathbb{R}$, this is known as the *product rule.*

3.6. **The chain rule.** Given two functions $f\colon U \to V$ and $g\colon V \to W$, we can define the composition $g \circ f\colon U \to W$ by $(g \circ f)(x) = g(f(x))$. If f is differentiable at x and g is differentiable at $f(x)$, then $g \circ f$ is differentiable at x. From the previous approximations, we see that, for $h \in U$,

$$\begin{aligned}(g \circ f)(x+h) &= g(f(x+h))\\ &\approx g(f(x) + f'(x)h)\\ &\approx g(f(x)) + g'(f(x))f'(x)h\\ &= (g \circ f)(x) + (g' \circ f)(x)f'(x)h\end{aligned}$$

where the approximation becomes better the smaller $\|h\|$ is. Here, the first approximation exploits differentiability of f at x as well as continuity of g at $f(x)$ (which follows immediately from differentiability of g at $f(x)$), and the second approximation exploits differentiability of g at $f(x)$ as well as linearity of $f'(x)$. Note that we use the multiplicative notation $(g' \circ f)(x)f'(x)$ to compose the linear maps $(g' \circ f)(x)\colon V \multimap W$ and $f'(x)\colon U \multimap V$ instead of writing $(g' \circ f)(x) \circ f'(x)$. If we, for functions $p\colon A \to (B \to C)$ and $q\colon A \to (C \to D)$, define $q \,\hat{\circ}\, p\colon A \to (B \to D)$ to be the function $(q \,\hat{\circ}\, p)(x) = q(x) \circ p(x)$, then we can write the above as

$$(g \circ f)' = (g' \circ f) \,\hat{\circ}\, f'. \tag{3.6}$$

This is known as the *chain rule.*

The chain rule is actually the only general rule we ever need, because every function can be expressed as compositions of more elementary functions. In particular, the sum and the product rules from above are instances of the chain rule. For example, the sum $f + g$ of two functions $f, g\colon U \to V$ can be considered the result of a series of function applications $x \mapsto (x, x) \mapsto (f(x), g(x)) \mapsto f(x) + g(x)$: that is,

$f+g$ is the composition of a map given by $x \mapsto (x,x)$; coordinate-wise applications of f and g given by $(x,y) \mapsto (f(x), g(y))$; and addition given by $(x,y) \mapsto x+y$. If we know the derivatives of these three elementary functions, the chain rule gives us the combined derivative of $f+g$, namely $f'+g'$.

3.7. **A toolbox of rules.** Our goal is to be able to find the derivative of any formula appearing in the world of finance. The chain rule is a major step toward achieving this goal. With this rule at hand, we can combine expressions whose derivatives we already know into expressions whose derivatives we can find using the rule. As mentioned, this includes sums and products, and hence also differences and quotients. In order to be able to find higher order derivatives, we must make sure that we can apply the chain rule repeatedly. In (3.6) we encountered the "lifted" function composition "$\hat{\circ}$", so we need to make sure that we can differentiate expressions involving that symbol. Fortunately, since we are only interested in finding the derivatives of $g \mathbin{\hat{\circ}} f$ when f and g are in the form $f\colon A \to (U \multimap V)$ and $g\colon A \to (V \multimap W)$, we can exploit that composition of linear functions is a bi-linear operation. Thus, finding the derivatives of $g \mathbin{\hat{\circ}} f$ is already covered by the product rule in (3.5).

With the chain rule at hand, all we need is a small set of elementary functions whose derivatives we know and can express using the same set of functions and the chain rule. These building block functions should include the constant functions, the identity function, injection functions ($\mathbb{R} \multimap \mathbb{R}^\ell$), projection functions ($\mathbb{R}^\ell \multimap \mathbb{R}$), the arithmetic operations ($\mathbb{R} \times \mathbb{R} \to \mathbb{R}$) as well as all the well-known functions $\mathbb{R} \to \mathbb{R}$ such as the exponential function, the logarithmic function, the reciprocal function, square root, sine, cosine, tangent and so on. Note that we can build arbitrary polynomials as sums and products of the identity function and the constant functions. Note also that the injection and projection functions allow us to put together expressions to form multivariate functions $\mathbb{R}^\ell \to \mathbb{R}^k$. Using these building blocks we can, for example, construct the function $f\colon \mathbb{R}^2 \to \mathbb{R}^3$ given by $f(x_1, x_2) = (x_1 \sin x_2, x_1 x_2^2, 1 + x_2)$ which was used as example in the above.

Finding the derivatives of the elementary functions requires a closer mathematical analysis which we will not go further into here. The important thing is that they are well-known and can be expressed in terms of each other. Thus, we now have the tools needed for finding all higher order derivatives of most multivariate functions, and we can begin differentiating in practice.

4. Differentiation in practice

4.1. **Symbolic differentiation.** *Symbolic differentiation* is the act of applying the differentiation rules described in Section 3 to a function in order to obtain a formula for the derived function. The derived function can then be applied to the point(s) we are interested in. This is the technique normally taught in high-school and would probably be the first choice for many people.

From a practical point of view, however, symbolic differentiation is quite inefficient. The method produces a function which can be used to compute the derivative at any point, and this may involve a lot of redundant computation, if we are just interested in the derivative at a specific point. For example, the real-valued function $x \mapsto \max(2, x^3 \exp x)$ clearly has derivative 0 for negative values of x, but symbolic differentiation would, nevertheless, compute a complete specification for the derivative even if we are only interested in its value at $x = -2$

If one wants to implement symbolic differentiation as software, it has an even larger drawback than bad performance: in order to find the derivative of an expression such as $\max(2, x^3 \exp x)$, it is necessary to access and transform the source code. Not only is this inconvenient, but it also places restrictions on the source code and may be straight out impossible.

There are solutions to this, of course. For example, one could overload all the primitive functions and function constructs so that they carry with them information about their derivatives. But this is no longer pure *symbolic* differentiation then, since we are doing more than just investigating the specification of a function. Automatic differentiation uses the overloading technique and will be discussed in Section 4.3.

4.2. **Numerical differentiation.** The traditional way of computing derivatives of functions in the financial sector is by *numerical differentiation*, sometimes also called *shocking*. The idea is to approximate the derivative $f'(x)$ by calculating function values in a neighborhood of x and using these to obtain $f'(x)$. The simplest version of this is a two-point estimation for a function $f\colon \mathbb{R} \to \mathbb{R}$, where we "shock" x with a small value h and find the slope of the secant line through $(x, f(x))$ and $(x+h, f(x+h))$:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{4.1}$$

Note that this is simply an approximation to the limit in (3.1). Note also that h can be negative which would mean that we are shocking x to the left rather than the right. We could also shock x equidistantly to both sides in order to get an approximation error that is not biased to one side:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

If f is multivariate, the same approach is still possible. In order to find the partial derivative $\partial f_i/\partial x_j$, we look at the i'th coordinate function f_i and keep all variables but x_j fixed. This leads to a one-dimensional function for which we can then apply (4.1). If we are looking for a *directional* derivative at x in direction h (corresponding to the matrix multiplication $f'(x)h$ from Section 3.2), we can consider instead the one-dimensional function $r \mapsto f(x + rh)$, $r \in \mathbb{R}$, to which we can apply (4.1).

The fact that the computation of a derivative by numerical differentiation involves two computations of function values means that the number of computations needed to compute an n'th derivative, à priori, grows exponentially with n. For example, the approximation to the second order derivative of a function $f\colon \mathbb{R}^2 \to \mathbb{R}$ is given by

$$\frac{\partial^2 f}{\partial x \partial y} \approx \frac{f(x+h, y+h) - f(x+h, y) - f(x, y+h) + f(x, y)}{h^2}$$

which requires calculation of four different function values.

If we differentiate multiple times by the *same* variable, however, it is possible to re-use some of the intermediate function values. For example, if we differentiate twice with respect to x, the above formula simplifies to

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2}$$

which only requires three function values to be calculated. In general, the number of function values needed to compute the n'th order derivative for the same variable grows only linearly with n.

Most financial key ratios are defined as first, second or third order derivatives, so the exponential growth in the computational complexity of finding n'th order derivatives is not as big a problem as it may sound. The largest problem is accuracy: too large values of h in (4.1) lead to inaccurate results; too small values of h lead to rounding errors. It is impossible to find a generic h that gives the optimal amount of accuracy for *all* possible functions. Thus, one is required to use different approximations for different functions, which either limits the set of such functions or requires a time-consuming, dynamic analysis of the function. Alternatively. one is required to use a better (but then probably also more time-consuming) approximation than the one in (4.1).

Although numerical differentiation may not be the best method for finding derivatives, it is quite standard in the financial sector. There are multiple reasons for this. One reason is speed: it is plain and simple fast—at least as long as we are only interested in lower order derivatives. The inaccuracy and limitations incurred by the use of numerical differentiation, as described above, are acceptable in many situations. Another reason is that, in some situations, we are truly interested in a formula such as (4.1) with a given fixed h, say $h = 1$, rather than the actual derivative. This could be for historic reasons: some key ratios have been defined before the time of computers using a formula such as (4.1), and hence it would simply be *wrong* to use a better approximation for the derivative, even though the true derivative is the original intent of the formula. It could also be that it is simply more descriptive for the fluctuations of a pricing function to know the change in value over a larger amount of time, rather than at an instant. For example, it may be more informative to know the value of a portfolio tomorrow rather than in a split second. Such "coarse" calculations are also less vulnerable to local anomalies and non-differentiable points.

4.3. **Automatic differentiation.** *Automatic differentiation* is a technique to evaluate the derivative of a function specified by a computer program by exploiting the observation in Section 3.7 that most functions are composed of a small set of elementary arithmetic operations and elementary functions.

The traditional way [12] of doing automatic differentiation is by a dynamic programming technique where the partial derivatives of an expressions are found one at a time using the chain rule at each step. If we use the chain rule starting with the innermost functions and work our way out, this is known as *forward accumulation*; if we start from the outside and work our way in, this is known as *reverse accumulation*. See [9, 13] for further details.

As an example, consider again the function $f\colon \mathbb{R}^2 \to \mathbb{R}^3$ given by

$$f(x_1, x_2) = (x_1 \sin x_2, x_1 x_2^2, 1 + x_2).$$

The *straight line* implementation of this function is an ordered sequence of primitive steps:

$$\begin{aligned} t_1 &= x_1 \\ t_2 &= x_2 \\ t_3 &= \sin t_2 \\ t_4 &= t_2^2 \\ t_5 &= t_1 t_3 \\ t_6 &= t_1 t_4 \\ t_7 &= 1 + t_2 \end{aligned}$$

The triple (t_5, t_6, t_7) is the output $f(x_1, x_2)$ of the function. The complete derivative $f'(x_1, x_2)$ is given by the set of partial derivatives $\partial f_i / \partial x_j$ for $i = 1, 2, 3$ and $j = 1, 2$, which corresponds to the partial derivatives $\partial t_m / \partial t_n$ for $m = 5, 6, 7$ and $n = 1, 2$. We already know that $\partial t_k / \partial t_k = 1$ for all k, and we can use the primitive differentiation rules to find $\partial t_\ell / \partial t_k$ for those t_k that are variables in the definition of t_ℓ.

Forward accumulation uses the rule

$$\frac{\partial t_m}{\partial t_n} = \sum_{n\le k<m} \frac{\partial t_m}{\partial t_k}\frac{\partial t_k}{\partial t_n}$$

where the sum is over the k's such that t_k is a variable in the definition of t_m. The rule is an immediate consequence of the chain rule. Using this rule, we can compute the sequence

$$\frac{\partial t_n}{\partial t_n}, \frac{\partial t_{n+1}}{\partial t_n}, \dots, \frac{\partial t_m}{\partial t_n}$$

which allows us to compute the derivatives $\partial f_i/\partial x_j$ for all i in one sweep. So, for example, if we want to compute $\partial f_i/\partial x_2$ for $i = 1, 2, 3$, which is the same as $\partial t_m/\partial t_2$ for $m = 5, 6, 7$, we would compute as follows:

$$\begin{aligned}
\frac{\partial t_2}{\partial t_2} &= 1\\
\frac{\partial t_3}{\partial t_2} &= \frac{\partial t_3}{\partial t_2}\frac{\partial t_2}{\partial t_2} = \cos t_2 \cdot 1 = \cos t_2\\
\frac{\partial t_4}{\partial t_2} &= \frac{\partial t_4}{\partial t_2}\frac{\partial t_2}{\partial t_2} = 2t_2 \cdot 1 = 2t_2\\
\frac{\partial t_5}{\partial t_2} &= \frac{\partial t_5}{\partial t_3}\frac{\partial t_3}{\partial t_2} = t_1 \cos t_2 = x_1 \cos x_2\\
\frac{\partial t_6}{\partial t_2} &= \frac{\partial t_6}{\partial t_4}\frac{\partial t_4}{\partial t_2} = t_1 \cdot 2t_2 = 2x_1x_2\\
\frac{\partial t_7}{\partial t_2} &= \frac{\partial t_7}{\partial t_2}\frac{\partial t_2}{\partial t_2} = 1 \cdot 1 = 1
\end{aligned}$$

In order to find the remaining partial derivatives $\partial f_i/\partial x_1$ for $i = 1, 2, 3$, we would have to repeat the process using t_1 in place of t_2. In general, finding the complete derivative of a map $\mathbb{R}^k \to \mathbb{R}^\ell$ requires k sweeps.

Reverse accumulation uses the rule

$$\frac{\partial t_m}{\partial t_n} = \sum_{n<k\le m} \frac{\partial t_m}{\partial t_k}\frac{\partial t_k}{\partial t_n}$$

where the sum this time is over the k's such that t_n is a variable in the definition of t_k. Using this rule, we can compute the sequence

$$\frac{\partial t_m}{\partial t_m}, \frac{\partial t_m}{\partial t_{m-1}}, \dots, \frac{\partial t_m}{\partial t_n}$$

which allows us to compute the derivatives $\partial f_i/\partial x_j$ for all j in one sweep. So, for example, if we want to compute $\partial f_1/\partial x_j$ for $j = 1, 2$, which is the same as $\partial t_5/\partial t_n$ for $n = 1, 2$, we would compute as follows:

$$\begin{aligned}
\frac{\partial t_5}{\partial t_5} &= 1\\
\frac{\partial t_5}{\partial t_4} &= 0\\
\frac{\partial t_5}{\partial t_3} &= \frac{\partial t_5}{\partial t_3}\frac{\partial t_5}{\partial t_5} = t_1 \cdot 1 = t_1\\
\frac{\partial t_5}{\partial t_2} &= \frac{\partial t_3}{\partial t_2}\frac{\partial t_5}{\partial t_3} + \frac{\partial t_4}{\partial t_2}\frac{\partial t_5}{\partial t_4} = \cos t_2 \cdot t_1 + 2t_2 \cdot 0 = x_1 \cos x_2\\
\frac{\partial t_5}{\partial t_1} &= \frac{\partial t_5}{\partial t_1}\frac{\partial t_5}{\partial t_5} = t_3 \cdot 1 = t_3 = \sin t_2 = \sin x_2
\end{aligned}$$

In order to find the remaining partial derivatives $\partial f_i/\partial x_j$ for $i = 1, 2$ and $j = 1, 2$, we would have to repeat the process starting with t_6 and t_7 in place of t_5. In general, finding the derivative of a map $\mathbb{R}^k \to \mathbb{R}^\ell$ requires ℓ sweeps.

The forward and reverse accumulation are two extreme methods of filling out the Jacobian matrix of derivatives one column or one row at a time, respectively, requiring k or ℓ sweeps for an $\ell \times k$-matrix. In general, the chain rule can be traversed with any combination of forward or reserve accumulation. The problem of computing the full Jacobian with a minimum number of operations is known as the *optimal Jacobian accumulation* problem and is NP-complete [10].

The remainder of this section presents two examples of automatic differentiation using forward accumulation.

4.4. **Dual numbers.** The *dual numbers* [11] are an extension $\mathbb{D}$ of the real numbers $\mathbb{R}$ obtained by adjoining an element ε with the property that $\varepsilon^2 = 0$. The construction is similar to that of the complex numbers, where one adjoins an element i with the property that $i^2 = -1$. A dual number is in the form $a + a'\varepsilon$, and addition and multiplication are given by

$$\begin{aligned}(a + a'\varepsilon) + (b + b'\varepsilon) &= (a + b) + (b + b')\varepsilon\\ (a + a'\varepsilon)(b + b'\varepsilon) &= ab + (ab' + a'b)\varepsilon\end{aligned}$$

where $a, a', b, b' \in \mathbb{R}$. The dual numbers form a ring (an algebraic construction with addition and multiplication) but not a field (a ring in which multiplication has an inverse operation), since $a'\varepsilon$ does not have an inverse for $a' \neq 0$. We will not go further into the algebraic details of the construction of dual numbers, but just take for granted that the definition works and produces a ring.

Dual numbers enable a forward accumulation automatic differentiation technique in a quite remarkable way. Given a differentiable function $f\colon \mathbb{R} \to \mathbb{R}$ with some known specification of $f(x)$, if we are able to extend the specification to a function $\mathbb{D} \to \mathbb{D}$, the result will be a function satisfying

$$f(x + \varepsilon) = f(x) + f'(x)\varepsilon. \tag{4.2}$$

For example, if $f(x) = x^2 + 3x + 1$, then

$$\begin{aligned}f(x + \varepsilon) &= (x + \varepsilon)^2 + 3(x + \varepsilon) + 1\\ &= x^2 + 2x\varepsilon + 3x + 3\varepsilon + 1\\ &= (x^2 + 3x + 1) + (2x + 3)\varepsilon\\ &= f(x) + f'(x)\varepsilon\end{aligned}$$

The same holds for functions where we do not have a closed formula as specification, e.g. the trigonometric functions, or the exponential or logarithmic functions. For these, we can just consider the Taylor expansions of the functions and do the same computations. In an implementation, the algorithm behind such functions is of course hidden, so we would have to manually specify how these functions should operate on dual numbers.

It is not pure magic, of course, that derivatives pop out of the computations with dual numbers: the ε corresponds to the h in (3.2), and the fact that $\varepsilon^2 = 0$ corresponds to the fact that $\lim_{h\to 0} h^2/h = 0$.

The idea of using dual numbers to compute derivatives can easily be extended to higher order derivatives. To do this, simply remove the equality $\varepsilon^2 = 0$ and generalize $\mathbb{D}$ to consist of power series in ε: that is, numbers in the form $a + a'\varepsilon +$

$a''\varepsilon^2 + \cdots$, where $a, a', a'', \ldots \in \mathbb{R}$. In place of (4.2) we then get

$$f(x+\varepsilon) = f(x) + f'(x)\varepsilon + \frac{1}{2!}f''(x)\varepsilon^2 + \frac{1}{3!}f'''(x)\varepsilon^3 + \cdots .$$

This is simply the Taylor expansion of f about x.

We can extend this even further to the multivariate case. Here we will need to keep track of a collection of ε_i's, one for each variable, and we then get

$$f(x_1+\varepsilon_1, \ldots, x_k+\varepsilon_k) = \sum_{i_1,\ldots,i_k} \frac{1}{i_1!\cdots i_k!} \frac{\partial^{i_1+\cdots+i_k} f}{\partial x_1^{i_1}\cdots \partial x_k^{i_k}} \varepsilon^{i_1}\cdots\varepsilon^{i_k}.$$

This formula also holds when the range of f is multi-dimensional; in this case, the sum is coordinate-wise over vectors over $\mathbb{D}$. The preceding section mentioned that forward accumulation required k sweeps for functions $\mathbb{R}^k \to \mathbb{R}^\ell$; this is reflected in the extra bookkeeping required to keep track of $\varepsilon_1, \ldots, \varepsilon_k$.

Dual numbers provide an elegant way of determining derivatives from function specifications. However, the elegance is mostly mathematical: an implementation to compute higher order derivatives of multivariate functions requires a lot of bookkeeping to keep track of all the ε_i's and the factorials $i_1!\cdots i_n!$ with which we need to multiply in order to obtain the derivatives. A better solution is to go back to the original definition in Section 3 of derivatives as linear maps and keep all these in one single data structure. This is exactly the idea pursued in the next section and in the implementation part of this project.

4.5. **Beautiful differentiation.** The computation of $f(x+\varepsilon)$ for dual numbers involves a computation of $f(x)$ (the real part) simultaneously with a computation of $f'(x)$ (the dual part) as well as higher order derivatives in the general case. Another way of obtaining this is by keeping function values and derivatives together in a single data structure and overloading all functions to work on such data structures.

Conal Elliot's *beautiful differentiation* [3] introduces a data structure in which any function value $f(x)$ for a function $f\colon U \to V$ at a point $x \in U$ is replaced by a *derivative tower* for f at x containing $f(x), f'(x), f''(x), \ldots$. We use the notation $U \triangleright V$ for this data type. Thus, an element of type $U \triangleright V$ will contain an element from U (corresponding to $f(x)$), a linear map $U \multimap V$ (corresponding to $f'(x)$), a linear map $U \multimap (U \multimap V)$ (corresponding to $f''(x)$), and so on. If we just want an ordinary value in V which is not the result of a function application, we will think of it as the result of an application of the identity function $V \to V$.

A function $g\colon V \to W$ must be overloaded to a function $U \triangleright V \to U \triangleright W$, so that the overloaded function applied to the derivative tower for f at x produces the derivative tower for $g \circ f$ at x containing $(g\circ f)(x), (g\circ f)'(x), (g\circ f)''(x), \ldots$. The key to this overloading is, of course, the chain rule. Further details are provided in the next section where an implementation (somewhat modified compared to Elliot's implementation in [5]) of beautiful differentiation is described. See also Section 6.1 for a discussion on the correctness of Elliot's implementation.

5. Implementation

The implementation consists of of five Haskell modules:

`Functions:` contains instances to lift functions so that we can work with them as values.

`Vectors:` contains the definition and instances of a new class to model vector spaces over a scalar field.

`MCSim:` contains the machinery to construct Monte Carlo simulated functions.

`AutoDiff:` contains an implementation of automatic differentiation. This is the central part of this project.

`Finance:` contains examples of financial pricing functions and key ratios calculated using automatic differentiation and Monte Carlo simulation.

This section describes these modules, focusing primarily on the implementation of automatic differentiation. Note that some of the ASCII characters in the code examples of this section have been replaced by more human-friendly symbols. The real code in the associated Haskell files is, of course, written in pure ASCII.

5.1. **Functions.** The `Functions` module is not directly related to automatic differentiation, Monte Carlo simulation or any of the other themes of this project. It is, in fact, just there for convencience, and this is the reason why it has been moved to its own separate module.

The module contains instances of `Num`, `Fractional`, `Floating` and `Ord` for the type `a → b`. This allows us, for example, to add, multiply and take reciprocals and maximums of functions whenever these operations are defined for the function values. Also, it allows us to apply mathematical functions to other mathematical functions by defining this as simple function composition. Thus, we can, for example, write an expression such as `exp*sin(cos+2*id+1)`, which is both easier to read and less verbose than what we would normally write:

```
*> exp*sin(cos + 2*id + 1) $ 3
-5.418932433500644
*> (\x → exp x*sin(cos x + 2*x + 1)) 3
-5.418932433500644
```

Note how the `id` function plays the role of the unknown variable when writing functions in this way. For multivariate functions, we can replace `id` by projection functions, and we can collect tuples of results using injection functions; we will use this trick in the `AutoDiff` module.

We have implemented maximum and minimum functions for the `Ord` instance, so that we, for example, can define $\max(\sin, \cos)$ to be the function whose value at x is $\max(\sin x, \cos x)$. The implementation of `Ord` is not complete, however, since we cannot make a general comparison operator for functions (because there is no reasonable way to define, say, if $\sin < \cos$ should be true or false).

5.2. **Vectors.** The `Vectors` module introduces the class `Vector v s` which we will use to model vectors of type `v` over a scalar field of type `s`:

```
class Fractional s ⇒ Scalar s

class Scalar s ⇒ Vector v s | v → s where
  zeroV :: v
  (^+^) :: v → v → v
  (*^) :: s → v → v
```

(Note that Elliot's implementation [5] also has a negation function for vector spaces; this is superfluous, since negation of a vector can be obtained by multiplication by the scalar -1.) The `| v → s` in the class definition means that the scalar type `s` is uniquely determined by the vector type `v`. In practice we will only use `Double` as scalars.

If U and V are vector spaces, then their direct sum $U \oplus V$ is a vector space, too, with the vector space operations defined component-wise. In general, for any collection $U_1, \ldots, U_n$ of vector spaces, the direct sum $\bigoplus_{i=1}^{n} U_i$ is a vector space. If all U_i's are identical and equal to U, we write U^n in place of $\bigoplus_{i=1}^{n} U$. In this case, we can model U^n as a set of functions $\{1, \ldots, n\} \to U$ with the vector space structure induced by that on U. Thus, addition and scalar multiplication are defined on functions from the same operations on function values. (This is exactly what the `Num` instance of `a → b` did in the `Functions` module.) We therefore define

```
instance Vector v s ⇒ Vector (a → v) s where
  zeroV   = const zeroV
  (^+^)   = liftA2 (^+^)
  (*^) s  = fmap (s *^)
```

Note that we have used the `liftA2` function from the `Control.Applicative` module to lift the binary function of addition.

To get started we just need a single vector space corresponding to the vector space $\mathbb{R}$ over itself:

```
instance Vector Double Double where
  zeroV = 0
  (^+^) = (+)
  (*^) = (*)
```

From this instance, we can, in one single step, build vector spaces of arbitrary dimension, even infinite, by using the type `a → Double` for some indexing set of type `a`. In practice, we will often use `Int` or `String` as indexing set, even though we only need a finite set of indexes, for example $\{1, \ldots, n\}$ or $\{$`"x1"`$, \ldots,$ `"xn"`$\}$; we must then keep in mind only to evaluate the indexing functions on this set.

We could also have made an instance of `Vector` for pairs of vector spaces in order to model $U \oplus V$, but then, to be able to build vector spaces of arbitrary dimension, we would need to nest pairs indefinitely, which would make it cumbersome to work with. Just having the simple instance for functions is much more general, and it is all we ever need. as long as we are only working with vector spaces in the form $\mathbb{R}^\ell$.

For convenience, the helper function `toC` takes a list of vectors of type `v` and turns them into a vector of type `Int → v`. (The "`C`" is for *column* since we think of these maps as index maps into a column vector.) The function `fromC` makes the opposite conversion, where one has to point out the requested indexes of the column. We can now do things such as

```
*> let u = toC [3,1,4 :: Double]
*> let v = toC [2,7,1 :: Double]
*> let w = u ^+^ v
*> fromC w [1..3]
[5.0,8.0,5.0]
```

Note that we need to specify the type of (at least one of) the elements in the lists. Otherwise Haskell cannot know what instance of `Vector` these should be interpreted as (they could, for instance, have been constant functions of type `a → Double` rather than just `Double`, due to the `Num` instance for functions).

The functions `toMap` and `fromMap` do exactly the same as `toC` and `fromC` except that they use an arbitrary set for indexing. Thus, we can, for example, index elements by variable names:

```
*> let u = toMap ["sigma", "x", "rho"] [3,1,4 :: Double]
*> let v = toMap ["x", "rho", "sigma"] [7,1,2 :: Double]
*> let w = u ^+^ v
*> fromMap w ["sigma", "x", "rho"]
[5.0,8.0,5.0]
```

5.3. **MCSim.** The `MCSim` module contains the tools needed to define Monte Carlo simulated functions: that is, functions in the form (2.1). The main function is

```
sim :: (Floating a) ⇒ [r] → (r → a) → a
sim ws f = sum (map (\w → f w) ws) / fromIntegral (length ws)
```

which takes an element of type `[r]` (corresponding to a list $(w_1, \ldots, w_N)$ of randomly generated variables) and a function of type `r → a` (corresponding to the function $\hat{f}(x, -)$) and gives a result of type `a` (namely the average of the $\hat{f}(x, w_i)$'s). What list of variables and what function to use is up to the user, but we provide two functions, `simUniforms` and `simNormals`, that as the list of variables use a list of n independent, randomly generated variables from the standard uniform and standard normal distributions, respectively. These two functions require a random generator as input in order to keep the implementation pure.

As a small test, let us try out the example with π from Section 2:

```
simPi :: (Num a, Ord a) ⇒ [a] → a → a
simPi [w1,w2] _ = if w1*w1 + w2*w2 < 1 then 4 else 0

testPi :: RandomGen g ⇒ g → Double
testPi g = sim (take 10000 $ tuples 2 $ randoms g) simPi 0
```

Here, `tuples 2` collects the elements in a list in pairs of two. After compilation, we can now estimate π with 10000 randomly generated variables, uniformly distributed on $[0; 1]^2$:

```
*> g ← getStdGen
*> testPi g
3.1376
```

Not the best approximation in the world—but this is as expected.

5.4. **AutoDiff.** We now proceed to the `AutoDiff` module which is the main part of the implementation. It contains the implementation of automatic differentiation, based on beautiful differentiation described in Section 4.5

Derivative towers. We implement derivative towers with a recursive data type in Haskell:

```
data u ▷ v = D v (u ▷ (u ⊸ v))
```

In the implementation as well as in this text, the ⊸ symbol is just a type synonym for →, which we use to distinguish linear maps from ordinary maps. This distinction could come in handy in future optimizations of operations on linear maps; see the discussion concerning optimization of linear maps in Section 6.3 on page 27.

An element of type `u ▷ v` is in the form

```
D fx D f'x D f''x D ···
```

where `fx` has type `v`, `f'x` has type `u ⊸ v`, `f''x` has type `u ⊸ (u ⊸ v)` and so on. The value of f and its derivatives at x is all we need to know in order to compute the derivatives of arbitrary expressions involving $f(x)$. Thus, the idea is that, in an expression involving $f(x)$, we replace $f(x)$ by its entire derivative tower and lift all the functions in the expression to work on derivative towers. This requires lifting all the functions from Haskell's `Num`, `Fractional` and `Floating` classes and is discussed further below.

Before we proceed, we create a few "fixed" derivative towers, namely those for the constant functions and for the identity function:

```
constD :: Vector v s ⇒ v → u ▷ v
constD x = D x (constD zeroV)

idD :: (Num v, Vector v s) ⇒ v → v ▷ v
idD x = D x (constD id)
```

Note that the derivative of the identity function at any point, as we can see above, is the identity function itself.

Derivative towers as vectors. We must ensure that vector space structures are preserved when going from values to derivative towers:

```
instance Vector v s ⇒ Vector (u ▷ v) s where
   zeroV  = constD zeroV
   (*^) s = \(D fx dfx) → D (s *^ fx) (fmapD (s *^) dfx)
   (^+^) = liftA2 (^+^)
```

The function `fmapD` lifts a function to act on values inside a derivative tower, so that scalar multiplication applies to all levels of a derivative tower. The use of `liftA2` means that we have implemented an instance of `Control.Applicative`. The result, in this case, is that addition of derivative towers is simply done coordinate-wise.

The sum rule and the product rule. The rule for `^+^` in the preceding states that the derivative of a sum is the sum of the derivatives. This is the sum rule! The preceding rule for `*^` is a special instance of the product rule in which one factor is a scalar. The real product rule, generalized to bilinear operators as it was introduced in (3.5), is obtained by lifting the operator to derivative towers:

```
bilinearD :: Vector w s ⇒
             (u → v → w) → (t ▷ u) → (t ▷ v) → (t ▷ w)
bilinearD op dfx@(D fx df'x) dgx@(D gx dg'x) =
  let u' `opl` v  = liftA2 op u' (const v)
      u `opr` v'  = liftA2 op (const u) v'
      df'gx = bilinearD (opl) df'x dgx
      dfg'x = bilinearD (opr) dfx dg'x
  in  D (fx `op` gx) $ (df'gx ^+^ dfg'x)
```

This is probably the most complicated function in the entire implementation—which is not so bad, after all.

Derivative towers as random variables. In order to integrate automatic differentiation with Monte Carlo simulation, we need to be able to generate random derivative towers. This can be obtained with an instance of the `Random` class:

```
instance (Random v, Vector v s) ⇒ Random (u ▷ v) where
  random g = let (z, g') = random g
             in (constD z, g')
  randomR (D x _,D y _) g = let (z,g') = randomR (x,y) g
                             in (constD z, g')
```

For simplicity, we have also made a `Random` instance for the type `t ▷ u → v ▷ w` of functions between derivative towers. This instance simply creates random derivative towers and turns them into constant functions.

These instances are all we need in order to use the Monte Carlo simulation framework in `MCSim` on derivative towers!

The chain rule. The chain rule will be “hidden” inside an operation used to overload functions to work on derivative towers. We begin by implementing the lifted function composition $\hat{\circ}$ from (3.6). As discussed in Section 3.7, we will only use the operation in the bilinear case, meaning that we can implement it quite easily:

```
infix 2 ^.^
(^.^) :: (Vector v s, Vector w s) ⇒
          (a ▷ (v ⊸ w)) → (a ▷ (u ⊸ v)) → (a ▷ (u ⊸ w))
(^.^) = bilinearD (.)
```

We next define an operator that can be used to construct overloaded versions of functions. The operator takes the usual function as its first argument and, recursively, the overloaded derivative as its second argument. The derivative should work according to the chain rule (3.6):

```
infix 1 ⋈
(⋈) :: (Vector v s, Vector w s)  ⇒
          (v → w) → (u ▷ v → u ▷ (v ⊸ w)) → (u ▷ v) → (u ▷ w)
(g ⋈ g’) dfx@(D fx df’x) = D (g fx) $ (g’ dfx) ^.^ df’x
```

Compared to Elliot’s implementation in [5], not only is this a simplification—it is also correct! See Section 6.1 for further details on the problems with Elliot’s implementation.

We finally also define a special version of ⋈ that can be used for one-dimensional functions, which is what we encounter most often:

```
infix 1 |>-<|
(|>-<|) :: (Vector s s, Vector v s)  ⇒
          (s → s) → (u ▷ s → u ▷ s) → (u ▷ s) → (u ▷ s)
g |>-<| g’ = g ⋈ (liftD . g’)
```

See further below for examples of how to use these functions.

Lifting functions to derivative towers. With the sum, product and chain rule in place, it only remains to implement overloaded versions of the elementary functions. Our elementary functions consist mostly of those defined by the `Num`, `Fractional` and `Floating` classes, so we need to make new instances of these for the `u ▷ v` type. Using the `|>-<|` operation from above, this is quite straightforward:

```
instance (Num s, Vector v s, Vector s s) ⇒ Num (v ▷ s) where
  fromInteger = constD . fromInteger
  (+) = liftA2 (+)
  (*) = bilinearD (*)
  negate = negate |>-<| -1
  abs = abs |>-<| signum
  signum = signum |>-<| 0

instance (Fractional s, Vector v s, Vector s s)
          ⇒ Fractional (v ▷ s) where
  fromRational = constD . fromRational
  recip        = recip |>-<| -recip sqr

instance (Floating s, Vector v s, Vector s s)
          ⇒ Floating (v ▷ s) where
  pi    = constD pi
```

```
exp   = exp   |>-<| exp
log   = log   |>-<| recip
sqrt  = sqrt  |>-<| recip (2 * sqrt)
sin   = sin   |>-<| cos
cos   = cos   |>-<| -sin
 ⋮               ⋮
```

This implementation is not completely correct! The absolute and signum functions are not differentiable at 0, but are here, nonetheless, implemented with derivative 0 at this point. The logarithm function is not even defined for negative values, but is here implemented with a derivative that is. All this is for simplicity only and could easily be fixed in various ways, but which solution to choose is a bit unclear and depends on the usage; see the discussion in Section 6.2. Note that the reciprocal and square root functions are defined in terms of themselves, meaning that the points where the functions are not defined automatically become points where the derivatives are not defined.

There are still a few functions missing before we can construct all the expressions that we want. The maximum and minimum functions will certainly be needed when creating pricing functions. These can be obtained by simply creating an instance of `Ord` for derivative towers:

```
instance (Vector u s, Vector s s, Ord s) ⇒ Ord (u ▷ s) where
  compare dfx dgy = compare (val dfx) (val dgy)
```

Note, however, that the maximum and minimum functions are not differentiable on the line (x, x); see Section 6.2.

The injection and projection functions are implemented according to how we model multi-dimensional vector spaces in the `Vectors` module:

```
injD :: (Eq a, Vector v s) ⇒ a → u ▷ v → u ▷ (a → v)
injD i = let inject v k = if k == i then v else zeroV
         in inject ⋈ (const . constD) inject

projD :: Vector v s ⇒ a → u ▷ (a → v) → u ▷ v
projD i = ($ i) ⋈ (const . constD) ($ i)
```

Note that the injection and projection functions do not know what indexing set they inject into or project from, respectively, so we need to keep track of that ourselves (which should not be too hard to do). The helper functions `makeVars` and `setVars` make it easy to set up multiple variables as projections and collect multiple results as injections; see the implementation for further details.

We now have all the functions we need to start playing. Let us first try out the simple one-dimensional function $f\colon \mathbb{R} \to \mathbb{R}$ given by $\sin(\cos x)$ at the point $x = 3$.

```
*> let D fx (D f'x (D f''x _)) = sin(cos) (idD (3 :: Double))
*> fx
-0.8360218615377305
*> f'x 1
-7.743200279648704e-2
*> f''x 1 1
0.5598543107302792
```

Here we have extracted the derivative $f'(x)$ applied to 1 and second derivative $f''(x)$ applied to 1 and 1. These are the values that would traditionally constitute the first and second order derivatives. The function `derivatives` extracts these values for us and puts them into a list:

```
*> take 3 $ derivatives $ sin(cos) (idD (3 :: Double))
[-0.8360218615377305,-7.743200279648704e-2,0.5598543107302792]
```

As a test, we can compare these values to the theoretical (symbolically derived) values:

```
*> [sin(cos) 3, (-cos(cos)*sin) 3, (-sin(cos)*sin*sin-cos(cos)*cos) 3]
[-0.8360218615377305,-7.743200279648704e-2,0.5598543107302792]
```

A perfect match! And certainly a better result than would have been possible with numerical differentiation.

The multivariate case is slightly more complicated to use. As an example, let us consider the function $f\colon \mathbb{R}^2 \to \mathbb{R}^3$ given by $f(x_1, x_2) = (x_1 \sin x_2, x_1 x_2^2, 1 + x_2)$, which we have already encountered in many previous examples:

```
*> :{
*| let { [x1,x2] = makeVars [1,2];
*|       y = [x1 * (sin x2), x1 * sqr(x2), 1+ x2];
*|       df = setVars [1,2,3] y;
*|       [x1r, x2r] = [3, 4 :: Double];
*|       dfx = df $ idD (toC [x1r, x2r]);
*|       D fx (D f'x (D f''x _)) = dfx
*|     }
*| :}
```

First we check that the function actually computes ordinary values correctly, by comparing to the theoretical results we have obtained in the previous sections:

```
*> fromC fx [1,2,3]
[-2.2704074859237844,48.0,5.0]
*> [x1r*sin x2r, x1r*sqr(x2r), 1 + x2r]
[-2.2704074859237844,48.0,5.0]
```

Next we check the first derivatives:

```
*> fromC (f'x (toC [1,0])) [1,2,3]
[-0.7568024953079282,16.0,0.0]
*> fromC (f'x (toC [0,1])) [1,2,3]
[-1.960930862590836,24.0,1.0]
*> [sin x2r, sqr(x2r), 0]
[-0.7568024953079282,16.0,0.0]
*> [x1r*cos x2r, 2*x1r*x2r, 1]
[-1.960930862590836,24.0,1.0]
```

Finally, we check the second derivatives:

```
*> fromC (f''x (toC [1,0]) (toC [1,0])) [1,2,3]
[0.0,0.0,0.0]
*> fromC (f''x (toC [1,0]) (toC [0,1])) [1,2,3]
[-0.6536436208636119,8.0,0.0]
*> fromC (f''x (toC [0,1]) (toC [1,0])) [1,2,3]
[-0.6536436208636119,8.0,0.0]
*> fromC (f''x (toC [0,1]) (toC [0,1])) [1,2,3]
[2.2704074859237844,6.0,0.0]
```

```
*> [cos x2r, 2*x2r, 0]
[-0.6536436208636119,8.0,0.0]
*> [-x1r*sin x2r, 2*x1r, 0]
[2.2704074859237844,6.0,0.0]
```

And so on. For each higher order derivative, there are more and more values to pick out in order to verify that the entire derivative is correct. Note, however, that we could also just have obtained a directional derivative straight away by choosing vectors different from `[0,1]` and `[1,0]` in the above. See the test function `test23` in the implementation for examples of this.

5.5. **Finance.** The module `Finance` contains some examples of financial calculations based on automatic differentiation and Monte Carlo simulation.

We have hard-coded some of the well-known Greeks [16] as derivatives with respect to variables indexed by certain names such as `sigma`, `tau` or `S`. This makes it easy to retrieve specific partial derivatives. For example, Vega is the first order partial derivative of an option's value with respect to the volatility of the underlying asset. In other words, Vega measures the *sensitivity* to volatility. Volatility is most often denoted σ, and the `vega` function computes the first order derivative with respect to the variable indexed by `sigma`. A more advanced example is Vera, which is the second order partial derivative with respect to the volatility σ and the interest rate r. The `vera` function computes the second order derivative with respect to the variables indexed by `sigma` and `r`. And so on.

We can now define the mathematical functions that we are interested in. It is important to note that there is no need to modify the function definition in order to use it on derivative towers instead of ordinary values. For example, the following function is a standard payoff function for call options, based on the simulation framework provided by the `MCSim` module:

```
simCallOptionPayOff g s k r t sigma =
  let f w = exp(-r*t) * max (p-k) 0 where
        p = s * exp((r - sqr(sigma)/2.0) * t + sigma * w * sqrt t)
  in simNormals g (10^4) f
```

We can use this function in the usual way:

```
*> g ← getStdGen
*> simCallOptionPayOff g 48 45 0.05 (7/12) 0.2
5.412133199131258
```

Or we can do it using derivative towers:

```
*> :{
*| let { input = ["s","k","r","t","sigma"];
*|        [s,k,r,t,sigma] = makeVars input;
*|        df = simCallOptionPayOff g s k r t sigma;
*|        [sr,kr,rr,tr,sigmar] = [48, 45, 0.05, 7/12, 0.2 :: Double];
*|        dfx = df $ idD (toMap input [sr,kr,rr,tr,sigmar])
*|     }
*| :}
*> val dfx
5.412133199131258
```

This result should be compared with the correct value 5.4374... calculated by hand using the Black–Scholes closed form. To get a more precise result, we would have to use more iterations (10^4 is not very many) or more evenly spread out random numbers; see the discussion in Section 6.3 on page 28. (Note that it is possible to get identical results even when random simulations are involved, since Haskell does not generate new random variables the second time we call for them.) Now that we are working with derivative towers instead of values, we can also obtain any desired derivatives. For example, we can ask for Greeks using the predefined functions:

```
*> vega dfx
11.436902358084298
*> vera dfx
-5.167952437518108e-16
```

The test function `testCallOptionPayOff` compares these values to corresponding values obtained by numerical differentiation.

6. Discussion

6.1. **Correctness.**

Conal Elliot uses a set of specifications for automatic differentiation to derive the implementation in [3]. In doing so, he is certain that the implementation will satisfy the specifications, and no further proof of correctness should ever be needed. Unfortunately, Elliot's implementation is built in stages (starting with one-dimensional, first order derivatives, then adding higher order derivatives and then adding higher dimensions), and the derivations are only described for some stages. In effect, the end result, which should be an implementation capable of handling multivariate, higher order derivatives, is not equipped with a complete derivation of its implementation and hence does not come with any guarantee for correctness. Indeed, the complete implementation is not thoroughly described in [3], especially with regards to how to combine the extension to higher order derivatives with the extension to multivariate functions.

To get the full story, I consulted Elliot's implementation in the `vector-space` library on Hackage [5]. The library comes in a multitude of versions, starting with version 0.0 and ending (at the time of writing) with version 0.8.4. The later versions are rather different from the description in [3], so I started with the earlier versions. As it turned out, many of the implementations are flawed. Version 0.0 contains the following implementation of the $\bowtie$ operator:

```
(>*<) :: (b → c) → (b → (b ⊸ c)) → (a ▷ b) → (a ▷ c)
f >*< f' = \ (D u u') → D (f u) ((f' u .) <$> u')
```

The type of this is wrong, since the second argument `f'` is just an ordinary function rather than the recursively defined overloaded version that works on derivative towers. The result is that derivatives of order higher than 1 are not computed correctly. The following example computation shows that the second order derivative of $\exp(x)$ is computed to be 0 for $x = 3$, which is wrong:

```
*> let D fx (D f'x (D f''x _)) = exp(dId 3) :: (▷) Double Double
*> fx
20.085536923187668
*> f'x 1
20.085536923187668
*> f''x 1 1
0.0
```

In version 0.1, this has been fixed, and a slightly modified definition of the type `u ▷ v` has been introduced. The implementation now looks as follows:

```
(>-<) :: VectorSpace b s ⇒ (b → b) → ((a ▷ b) → (a ▷ s))
      → (a ▷ b) → (a ▷ b)
```

```
    f >-< f' = \ b@(D b0 b') → D (f b0) ((f' b *^) . b')
```

The types are now correct and the example from before gives the correct second order derivative. But there is another problem: the function composition "`.`" is not itself overloaded, and hence derivatives of non-trivial function compositions will not be correct. The following example shows that the first-order derivative of $\exp(\exp x)$ is computed to be 0 for $x = 3$, which is wrong:

```
    *> let dfx = exp(exp 3) :: (▷) Double Double
    *> dVal dfx
    5.284913114854943e8
    *> dVal $ (dDeriv dfx) 1
    0.0
```

In general, later versions of Elliot's implementation seem to get more and more details correct but are, at the same time, further and further away from the description in [3]. The conclusion must be that, even though the idea of deriving an implementation from a set of specifications is good, it must be carried out to the full extend for all parts of the code to be reliable. In practice, code evolves over time and one cannot derive implementations from scratch at every step. A set of tests that could be run at every step would have been of much more value than an incomplete derivation of the implementation of a first version.

In my implementation, I have tried to follow the descriptions in [3] as closely as possible while keeping the system as simple as possible, postponing, for now, issues such as performance. I have fixed the problems in Elliot's early implementations and have done a significant amount of testing to see if the system works as intended. Of course, even thorough testing does not provide the same sort of 100 % correctness guarantee as a derivation from specifications, but on the other hand, humans make mistakes, and a wide collection of tests are more likely to catch errors than a single, human-made derivation on paper, especially when the derivation is not updated along with every update of the implementation.

A few of the tests I have made during the implementation can be found in the source code. In the `AutoDiff` module, the tests compare calculated derivatives to theoretical results, whereas in the `Finance` module, the tests compare calculated derivatives (key ratios) to values found by numerical differentiation (shocking). The reader is encouraged to consult these tests and validate their correctness.

There is one problem in my implementation (as well as in Elliot's) concerning non-differentiable points. This is discussed below.

6.2. **Non-differentiable points.** The previous discussions (and the implementation) have completely ignored the fact that not all functions are differentiable at all points. The signum and absolute functions, for instance, are not differentiable at $x = 0$. The reciprocal function is not even defined for $x = 0$, and the logarithm is not even defined for non-positive values.

There are several things we could do to handle such problems, some of which will be discussed below. In the implementation, we have ignored these problems, not because they are hard to fix, but because ignoring them yields simpler code and because it is not quite clear what would be the desired way to fix the problems. Said differently: it depends on the situation how such points should be handled. Thus, to make the code ready for production, non-differentiable points should not only be handled, but it should be possible for the user to choose between many different alternatives for how to handle them. The remainder of this section contains some suggestions for how to handle such points.

Hard-coding values. One way of handling non-differentiable points is to simply hard code a value for the derivative at such points. For example, the derivative of the signum function is 0 for all points $x \neq 0$, so it would make sense to set it to 0 at $x = 0$ too. This is, in fact, what we have done in the implementation (although not because it is the best solution but because it is the simplest). It is less clear how to handle the reciprocal function, since its value goes to ∞ and its derivative to $-\infty$ as we approach 0 from the right, whereas its value goes to $-\infty$ and its derivative to $-\infty$ as we approach 0 from the left. We cannot really fix the discontinuity of the reciprocal function, and the only reasonable value for its derivative at $x = 0$ is $-\infty$ which may not be what we want.

Smoothing. Another solution would be to smoothen functions at critical points. The signum function, for instance, could be smoothened by an arbitrarily often differentiable function that, for some small $\varepsilon > 0$, is constantly equal to -1 on $]-\infty; -\varepsilon]$, grows rapidly from -1 to 1 on $]-\varepsilon; \varepsilon[$ and is constantly equal to 1 on $[\varepsilon; \infty[$. This is a somewhat dangerous solution, since the values of the derivative around 0 will depend heavily on the choice of ε and of the smoothening function, which makes results hard to predict. In some situations, however, smoothing is quite desirable, because it allows a sudden and abrupt change in value to be predicted beforehand, since the change after smoothing occurs gradually over a longer interval. This makes it possible to "look into the future" and predict changes at an earlier stage, which is of value when analyzing risks of financial products. Note that an implementation of numerical differentiation would not necessarily have problems with non-differentiable points but would approximate a derivative in a way similar to what we would obtain by smoothing.

Throwing errors or special values. The perhaps best and certainly the safest solution would be simply to throw an error or a special value when a point of non-differentiability is encountered. In a sense, this behavior is already built into the reciprocal function, since Haskell returns `Infinity` when evaluating 1/0. (Note that all derivatives of the reciprocal function are defined in terms of the reciprocal function, so that we do not run into problems with higher order derivatives either.) It would be straightforward to define the signum function such that its overloaded derivative throws an error when passed 0 as argument.

6.3. **Performance.** Performance has not been a real consideration during this project. We have overloaded functions so that they work on derivative towers and this, of course, means that we need to carry around some more baggage when evaluating functions. However, as long as we just evaluate function values without asking for any derivatives, the extra baggage does not have any large influence on performance in a lazy language like Haskell. In the implementation, function values are wrapped inside a derivative tower, but only at the top level of the tower, and as long as we just ask for function values and not any derivatives, no derivative tower will need to be evaluated beyond the first level. Further, it will never be necessary to look up any of the overloaded derivatives of functions, meaning that it will be straightforward to evaluate a function value simply by using the old, non-overloaded functions on the unwrapped values at the first level of the derivative towers. In conclusion, the only prize we pay for simple function evaluation is the prize of wrapping functions and values inside data structures, and that will only impact performance by a constant factor.

When we look at derivatives of first order, we need to look up the overloaded derived functions, and we need to move to the second level of the derivative towers to find the derivative values. The latter is not so bad and only contributes a constant factor to the time complexity as the number of function compositions grows. What really matters is the chain rule in (3.6) in the overloaded derived functions. A small induction proof shows that, for a function in the form $f = f_1 \circ \cdots \circ f_n$,

$$f' = (f_1' \circ f_2 \circ \cdots \circ f_n) \,\hat{\circ}\, (f_2' \circ f_3 \circ \cdots \circ f_n) \,\hat{\circ} \cdots \hat{\circ}\, (f_{n-1}' \circ f_n) \,\hat{\circ}\, f_n' \tag{6.1}$$

When applied to an element x, this gives

$$f'(x) = ((f_1' \circ f_2 \circ \cdots \circ f_n)(x))((f_2' \circ f_3 \circ \cdots \circ f_n)(x)) \cdots ((f_{n-1}' \circ f_n)(x)) f_n'(x),$$

which is a composition of n linear maps constructed from $\binom{n}{2}$ function compositions. Thus, the computation of the derivative of a composition of n functions involves composing $\binom{n+1}{2} = \mathcal{O}(n^2)$ functions. This can be optimized quite a bit, however: first of all, we can re-use the computation of $(f_{k+1} \circ \cdots \circ f_n)(x)$ when computing $(f_k \circ \cdots \circ f_n)(x)$; second, we can optimize the (outer) composition of linear maps by representing them as matrices; third, we can compute the (inner) function compositions in parallel and combine the resulting matrices in a divide-and-conquer manner. See the discussions below on memoization, optimization of linear maps and parallelization for more detail on these optimization techniques.

Although the number of function compositions has substantial impact on the time complexity, it is nothing compared to that of the order of the derivatives. Here we consult Faà di Bruno's formula [15] for a higher order chain rule:

$$\frac{\partial^n}{\partial x_1 \cdots \partial x_n}(g \circ f) = \sum_{p \in P(n)} g^{|p|}(f(x)) \cdot \prod_{B \in p} \frac{\partial^{|B|} f}{\prod_{j \in B} \partial x_j}$$

where $P(n)$ is the set of partitions of $\{1, \ldots, n\}$ (that is, the set of non-empty, disjoint subsets whose union is $\{1, \ldots, n\}$) and $|A|$ denotes the cardinality of the set A. In the usual notation where derivatives are linear maps, this becomes

$$(g \circ f)^{(n)} = \sum_{p \in P(n)} (g^{(|p|)} \circ f) \,\hat{\circ} \hat{\bigcirc}_{B \in p} f^{(|B|)} \tag{6.2}$$

where the big overloaded composition "$\hat{\bigcirc}$" runs over the subsets B of the partitions p in $P(n)$ ordered according to their *largest* element; for example, the partition $p = 1|24|3$ (which is the short way of writing $\{\{1\}, \{2, 4\}, \{3\}\}$) in $P(4)$ with this ordering should be arranged as $1|3|24$ and will therefore yield the overloaded composition $f' \,\hat{\circ}\, f' \,\hat{\circ}\, f''$. When applied to an element x, formula (6.2) gives

$$(g \circ f)^{(n)}(x) = \sum_{p \in P(n)} (g^{(|p|)} \circ f)(x) \prod_{B \in p} f^{(|B|)}(x)$$

where the product is, in fact, a big composition of linear maps which can be represented as a product of matrices in the same order as in (6.2).

The total number of partitions in $P(n)$ is the n'th *Bell number* [14], B_n, defined recursively as

$$B_0 = 1 \qquad \text{and} \qquad B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k.$$

Bell numbers are asymptotically lower bounded [2] by $(n/e \log n)^n$, and hence the number of summands in the sum in (6.2) grows superpolynomially with n, which is very bad news from a performance point of view. The big product of each summand is over at most n matrices and hence grows linearly with n which certainly is acceptable. Overall, the performance is really bad, but again there is quite a bit room for optimization: first of all, we can re-use computations (for example, the overloaded composition $f' \,\hat{\circ}\, f' \,\hat{\circ}\, f''$ appears several times in the computation of a fourth derivative); second, we can optimize the composition of linear maps by representing them as matrices and combining them in a divide-and-conquer manner; third, we can compute each summand in the sum in parallel and compute the sum in a divide-and-conquer manner. See the discussions below on memoization, optimization of linear maps and parallelization for further details on these optimization techniques.

The first order derivative of n function compositions and the n'th order derivative of one function composition are the two extremes. We should, of course, investigate how the m'th order derivative of n function compositions looks and how its implementation can be optimized and, perhaps, parallelized. This will be left for future studies. The conclusion, for now, is that the implementation part of this project, asymptotically, does not perform that bad when it comes to normal function evaluation, and that lower order derivatives can be computed in a reasonable amount of time whereas higher order derivatives offer a somewhat larger performance challenge. The former is good news for financial calculations, because the functions there tend to be complicated (composed of many smaller functions), whereas the order of the derivatives we consider rarely exceeds 3. In any case, there is ample opportunity for optimizations.

We conclude this section with a collection of suggestions for performance improvements.

Optimizing zeros. The derivatives of a constant function is zero which itself is a constant function. Thus, there is no need to compute any (higher order) derivatives for a constant function as we already know they will be zero. A way to avoid unnecessary computation is therefore to change the type of derivative towers to

```
u ▷ v = D v (Maybe (u ▷ (u ⊸ v)))
```

with the derivative being `Nothing` whenever the function is constant. It does not require many changes to the implementation to accommodate for this changed type, but the gain in performance may be significant. (Note that, in an expression such as $3\sin(2x)$, both 3 and 2 are considered to be constant functions.)

Optimizing linear maps. There are a lot of linear maps involved in computing derivatives and hence much to be gained from an efficient implementation of them. A linear map can be represented as a matrix, and it may in some situations be more efficient to represent linear maps as matrices rather than maps. Because one can evaluate a matrix, it is, in a sense, possible to evaluate the linear map before applying it, which allows the evaluation of the map to be re-used. In particular, it is possible to evaluate the composition of linear maps before it is applied to an element, simply by multiplying the corresponding matrices. Since matrix multiplication is associative, the composition of many linear maps can therefore be performed in parallel in a divide-and-conquer way; see further below.

The later versions of Elliot's implementation [5] try to optimize linear maps. The implementation in this project does not, but it is prepared for such an optimization, having used the different type `u ⊸ v` for linear maps instead of `u → v`. All that remains to be done is to change the implementation of how such linear maps should be created, applied, composed etc.

Memoization. Memoization is a technique in which the performance of a program is improved by storing the results of function calls, thereby avoiding recalculation of previously calculated values. The later versions of Elliot's implementation [5] contain some kind of memoization. There may be complications if memoization is to be used in combination with parallelization; see below.

Parallelization. As discussed above there are many opportunities for running some of the computations in this project's implementation in parallel. Which to pursue will depend mainly on the usage of the system, but since we are mostly concerned with financial calculations, it will most likely be more fruitful to consider parallelizations that speed up computations of low order derivatives of complex functions rather than high order derivatives of simple functions. (What makes a function "complex" in this phrasing is simply that it is composed of many functions.) There are challenges with respect to shared memory: if we allow memory to be shared between processes, then we can re-use computations between processes and exploit memoization. On the other hand, this may require some processes to wait for others to finish which defeats the purpose of running things in parallel. The optimal solution should probably investigate exactly which tasks should be spawned as subprocesses and what information should be shared between processes. It may be difficult or even impossible to find a work-efficient parallel algorithm, but it is certainly worth investigating further.

Commutativity of differentiation. Differentiation with respect to different variables does not depend on the order in which we differentiate. For example, $\partial^2 f/\partial x\partial y = \partial^2 f/\partial y\partial x$, meaning that, if one has already computed $\partial^2 f/\partial x\partial y$, there is no need to also compute $\partial^2 f/\partial y\partial x$. In the setting of linear maps, this means that the three-dimensional matrix representing $f''(x, y)$ is symmetric around a diagonal plane. This information could help save some computations in combination with optimizing linear maps or using memoization (see above).

Striking a balance between forward and reverse accumulation. Section 4.3 discussed the NP-complete optimal Jacobian accumulation problem of finding the optimal balance between forward and reverse accumulation. It will most likely not improve performance of the implementation to try to solve an NP-complete problem during execution, but perhaps there are fast approximation algorithms or randomized algorithms that could help determine a better balance between forward and reverse accumulation than the plain forward-mode implementation we have used. How exactly to incorporate reverse accumulation into the beautiful differentiation setup is not very clear, but it may be something worth investigating in the future.

Quasi-Monte Carlo simulation. The definition of Monte Carlo simulation requires random elements to be drawn independently from a distribution. In some situations it may be more efficient (meaning that it leads to a faster convergence) to draw samples dependently or even deterministically. A *low discrepancy sequence* [17] is, roughly speaking, a sequence with the property that any initial segment of points in the sequence is evenly distributed, meaning that the probability of a point falling into a specific set is about proportional to the size (or measure) of the set. Using a low discrepancy sequence for Monte Carlo simulations is normally referred to as quasi-Monte Carlo simulation.

Whether or not it is beneficial to use quasi-Monte Carlo simulation depends on the function we want to simulate, and hence it should probably be left to the user to decide. Whatever type of simulation the user chooses, the implementation of the function needs not change very much, because the implementation of the function $\hat{f}$ from (2.1) can be kept independent of how the values of the w_i's are obtained. In other words, only the few calls to `MCSim` where the simulation is made need change. For example, in the implementation of `simCallOptionPayOff`, we can substitute the call to `simNormals` with any other call that returns function values simulated by drawing samples from a distribution, whether it is random or not.

In any case, quasi-Monte Carlo simulation is in use the financial world, and hence should also be covered by the system in a future version.

6.4. **Other languages.** The choice of Haskell as the language of implementation is not completely arbitrary. First of all, not all languages would allow a recursive type such as `u ▷ v = D v (u ▷ (u ⊸ v))`. Second, not all languages have lazy evaluation, meaning that it would not be possible to have infinite constructs such as derivative towers or the overloaded functions that work on them. The implementation depends heavily on these two features.

If we were to implement derivative towers in a language with a less tolerant type system, we would have to change the data model. Standard ML, for instance, does allow recursive types, including the above, but would not allow the construction of a function that can process such a type, because the definition of `u ▷ v` does not refer to the same instance of itself. Standard ML would, however, make it possible to work with the variant `u ▷ v = D v (u ⊸ (u ▷ v))`. This type is a version of the derivative tower, in which the arguments to a linear function at a specific level in the tower must also be given to the linear function at the next level. The later versions of Elliot's implementation [5] actually uses this type for derivative towers. In languages that do not support recursive types whatsoever, we can still get around the problems by throwing in a few "hacks". Even in an object-oriented language like Java, we could simulate (the above variant of) derivative towers by having each level of the tower be an object equipped with a method that returns the object for the next level.

If we were to implement the system in an eager language, we would also need to make changes to avoid infinitely many redundant computations. Standard ML is eager, but one can achieve lazy evaluation in various ways. The aforementioned implementation in Java could even be considered "lazy" in some sense, since we would not construct objects for the lower levels except if they are called for.

So, in general, anything is possible, but some languages are certainly more suited for certain tasks than others. In this case, Haskell is an excellent choice.

6.5. **Future work.** The previous discussion shows that there are still many things to do with respect to performance optimizations, handling non-differentiable points, and so on. The entire Monte Carlo functionality could certainly also be extended from just covering functions simulated as averages of other functions to handling entire simulation scenarios. To make things even more applicable to financial calculations, we could also include a calendar system and other mechanisms to handle real-world problems. Finally, the way we handle multi-dimensional vector spaces seems a bit clumsy: a nicer interface would be desirable.

However, all this was not the purpose of this project. The purpose was to provide a proof-of-concept that automatic differentiation can be combined with Monte Carlo simulation, which is very useful for valuating financial constructs. So if we set all the nice-to-have features aside, where else can we go from here?

With a library for automatic differentiation and Monte Carlo simulation in place, we can find key ratios for a multitude of financial products. Users of the system must, however, still implement the pricing functions themselves. Jones et al. [7] have introduced a combinator library that allows one to define financial products and from the definition automatically derive appropriate pricing functions. If such a library was added to the implementation, the only thing users would ever have to do would be to provide the definitions of the desired financial products as well as of the models to use for market observables and scenario calculations, after which prices and key ratios would be readily available. Add some of the performance optimizations above, and the user will not have to wait unnecessarily long for the results. Throw in a library for chain fractions, and we can even provide results with arbitrary precision and precise error estimates.

All this, and more, is left for future studies.

References

[1] *Computational differentiation: Techniques, applications and tools*, SIAM, 1996.

[2] Daniel Berend and Tamir Tassa, *Improved bounds on bell numbers and on moments of sums of random variables.*, Probability and Mathematical Statistics **30** (2010), no. 2, 185–205.

[3] Conal Elliott, *Beautiful differentiation*, International Conference on Functional Programming (ICFP), 2009.

[4] Andreas Griewank and Andrea Walther, *Evaluating derivatives: Principles and techniques of algorithmic differentiation*, 2nd ed., SIAM, Philadelphia, 2008.

[5] Hackage, *The vector-space package*, `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/vector-space`, 2008–2012, [Online; accessed 18 November 2012].

[6] Kenneth E. Iverson, *The derivative operator*, Proceedings of the international conference on APL: part 1 (New York, NY, USA), APL '79, ACM, 1979, pp. 347–354.

[7] Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward, *Composing contracts: an adventure in financial engineering, functional pearl*, ICFP, 2000, pp. 280–292.

[8] Jerzy Karczmarczuk, *Functional differentiation of computer programs*, Higher-Order and Symbolic Computation **14** (2001), no. 1, 35–57.

[9] John Mount, *Gradients via reverse accumulation*, Tech. report, Win-Vector LLC, 2010.

[10] Uwe Naumann, *The art of differentiating computer programs: An introduction to algorithmic differentiation*, SIAM, 2012.

[11] Barak A. Pearlmutter, *Lazy multivariate higher-order forward-mode ad*, In Proceedings of the 2007 Symposium on Principles of Programming Languages, 2007.

[12] R. E. Wengert, *A simple automatic derivative evaluation program*, Commun. ACM **7** (1964), 463–464.

[13] Wikipedia, *Automatic differentiation — wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Automatic_differentiation&oldid=522047439`, 2012, [Online; accessed 14 November 2012].

[14] ———, *Bell number — wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Bell_number&oldid=516743822`, 2012, [Online; accessed 19-November-2012].

[15] ———, *Faà di bruno's formula — wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Fa%C3%A0_di_Bruno%27s_formula&oldid=517416427`, 2012, [Online; accessed 19-November-2012].

[16] ———, *Greeks (finance) — wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Greeks_(finance)&oldid=513955623`, 2012, [Online; accessed 23-November-2012].

[17] ———, *Low-discrepancy sequence — wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Low-discrepancy_sequence&oldid=521966320`, 2012, [Online; accessed 21-November-2012].

[18] ———, *Monte carlo method — wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Monte_Carlo_method&oldid=520937170`, 2012, [Online; accessed 14 November 2012].