UNIVERSITY OF WATERLOO
Faculty of Mathematics

Do Not Drink and Derive; Smoke Adjoints and Fly!

Oliver Wyman
Toronto, ON

Jose Luis Avilez
4B Mathematical Finance
ID 20702106

December 21, 2018

MEMORANDUM OF SUBMITTAL

To:	Matt McPhail
From:	Jose Luis Avilez
Address:	120 Bremner Blvd, Toronto, ON M5J 3A6
Date:	December 21, 2018
Re:	Do Not Drink and Derive; Smoke Adjoints and Fly!

As agreed, I have prepared the enclosed report, "Do Not Drink and Derive; Smoke Adjoints and Fly!" for my 4B work report and for the ATLAS team. This report, the second of five work reports that the Co-operative Education Program requires that I successfully complete as part of my BMath Co-op degree requirements, has not received academic credit.

The ATLAS team builds software that clients use to price liabilities at Oliver Wyman. My job required that I handle the computation of derivatives of a price function. This paper is an analysis of the results of my investigation.

The Faculty of Mathematics requests that you evaluate this report for command of topic and technical content/analysis. This report is written for someone who has taken a first calculus course in mind, implying the key findings should be understandable by most audiences. Following your assessment, the report, together with your evaluation, will be submitted to the Math Undergrad Office for evaluation on campus by qualified work report markers. The combined marks determine whether the report will receive credit.

Thank you for your assistance in preparing this report.

Jose Luis Avilez

TABLE OF CONTENTS

List of Figures

EXECUTIVE SUMMARY

Determining financial instrument sensitivities—or partial derivatives—is one of the main interests in the field of quantitative finance and the ATLAS team at Oliver Wyman. One novel method to compute an instrument's sensitivities is to use algorithmic adjoint differentiation (AAD). This report considers a Python and C++ implementation of AAD to determine its potential use within ATLAS software.

In Python, AAD was implemented using Autograd to price and compute the sensitivities of European and Asian call options using Monte Carlo simulation. In C++, a variety of libraries listed in the Autodiff website were tested with the aforementioned two instruments, with the addition of guaranteed minimum accumulation benefit, a liability product.

The analysis revealed that AAD is able to yield anywhere between a two-fold and five-fold speed boost, depending on the computational complexity of the instrument's price. Of note, the experiments showed that no extra cost was incurred in computing multiple derivatives.

This report recommends leveraging the code produced during this investigation for a potential future implementation of AAD by the ATLAS team.

1 INTRODUCTION

The central question in the field of computational finance is to compute the price function $P : \mathbb{R}^n \to \mathbb{R}$ of a financial instrument and its derivative matrix $DP \in M_{1\times n}(\mathbb{R})$. In financial terminology, the entries of the matrix DP, which encode the partial derivatives with respect to each input parameter, are called the "Greeks" or sensitivities [2]. The Greeks include various partial derivatives, usually taken with respect to equity or underlying price (delta), time (theta), interest rate (rho), amongst other financial variables.

1.1 Mathematical Preliminaries

Recall that the derivative of a function P at the point a is defined the unique linear transformation T such that, for a vector $h \in \mathbb{R}^n$ [3]:

$$\lim_{h\to 0} \frac{\|P(a+h) - P(a) - T(h)\|}{\|h\|} = 0$$

Furthermore, if the function P is scalar valued and continuously differentiable, the matrix representation of the derivative at a is called the gradient of P and can be written as $DP(a) = \nabla P(a)$, where the j-th entry is the partial derivative with respect to the j-th coordinate, and is given by:

$$\frac{\partial P}{\partial x_j} = \lim_{h\to 0} \frac{P(a+he_j) - P(a)}{h} \tag{1}$$

where $h \in \mathbb{R}$ and $e_j = (0, \ldots, 0, 1, 0, \ldots, 0)$, where the 1 entry occurs at the j-th position.

1.2 Derivatives in Finance

The specific price function this paper is concerned with is the price of an option. An option is a financial instrument that gives the holder the right (but not obligation) to buy a

product (say, a stock) at a certain date [4]. Models for stock options usually arise by projecting the underlying instrument via stochastic differential equations [5]; in this case, the ATLAS team is interested in European options, whose underlying stochastic process gives rise to the Black-Scholes equation [6]:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{2}$$

where V is the option price, t is time, S is the stock price, r is the risk-free interest rate, and σ is the volatility of the stock. A popular method to solve this equation for V is simulating multiple paths S can take and taking the present value of the mean of the payouts of these paths; this method is called Monte Carlo simulation [7]. Formally, this requires computing a single-parameter function: $\alpha : \mathbb{R} \to \mathbb{R}$ where:

$$\alpha(\theta) = \mathrm{E}_{\mathbb{Q}}\left[P(x_1, \ldots, x_{j-1}, \theta, x_{j+1}, \ldots, x_n)\right]$$

where $\mathrm{E}_{\mathbb{Q}}$ is the risk-neutral[1] expectation, the sensitivity with respect to the parameter θ is $\alpha'(\theta)$ [8]. This sensitivity can be calculated by a suitable choice of h in equation (1) or, if a lower error is desired, a two-sided finite difference can be used as below:

$$\frac{\partial P}{\partial x_j} = \lim_{h \to 0} \frac{P(a + he_j) - P(a - he_j)}{2h} \tag{3}$$

The methods outlined in equations (1) and (3) are slow and expensive, as they required repeated computations of the price function at multiple points. Therefore, a novel method for computing sensitivities, algorithmic adjoint differentiation (AAD), is developed and discussed in this paper.

[1]The fundamental theorem of asset pricing allows us to define a risk-neutral world as one in which there are no players which can craft a strategy which guarantees them a profit with probability one (i.e. the market is arbitrage-free).

2 ANALYSIS

The philosophy behind AAD is to view computer programs as a sequence of elementary functions whose derivatives can be propagated as the target computation proceeds, rather than separately, by using the chain rule; this idea was born out of Giles' and Glasserman's paper "Smoking adjoints: Fast Monte Carlo Greeks" [9], whose title was borrowed and refined for this paper. It has been a topic of wide interest in computational finance [10] and the source for various theses conceived at the University of Waterloo [11, 12, 13]. This section examines the mathematics of AAD and detail several performance metrics when implementing it to three different financial products.

2.1 The Mathematics of AAD

Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$ is a continuously differentiable function, which is computed as $f(\vec{x}) = \vec{y}$, where $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_m)$. Suppose, additionally, that f can be computed sequentially as $\vec{x} \to \ldots \to \vec{u} \to \vec{v} \to \ldots \to \vec{y}$ where $\vec{u}$ and $\vec{v}$ are intermediate variables. Let $\overline{V}$ denote the adjoint of $\vec{v}$, and define it component-wise as [1]:

$$\overline{V}_k = \sum_{j=1}^{m} \overline{Y}_j \frac{\partial f_j}{\partial v_k}$$

where $\overline{Y}$ is a suitably selected vector in $\mathbb{R}^m$. Using the chain rule and the definition of an adjoint, the following is obtained by working backwards:

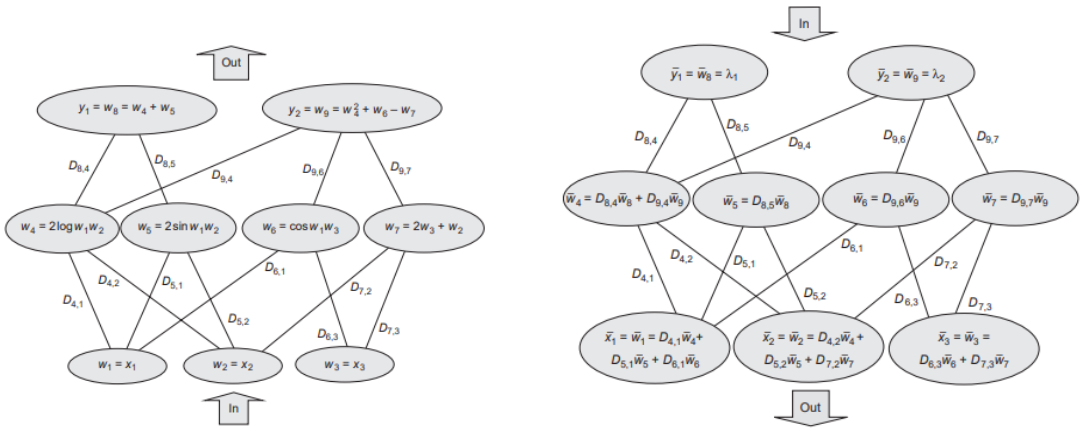
$$\overline{U}_i = \sum_{j=1}^{m} \overline{Y}_j \frac{\partial f_j}{\partial u_i} = \sum_{j=1}^{m} \overline{Y}_j \sum_k \frac{\partial f_j}{\partial v_k} \frac{\partial v_k}{\partial u_i} = \sum_k \overline{V}_k \frac{\partial V_k}{\partial U_i}$$

which allows for the computation of the sequence $\overline{Y} \to \ldots \to \overline{V} \to \overline{U} \to \ldots \to \overline{X}$.
Re-tracing these steps until the input variables shows that the adjoint of $\vec{x}$ is a linear

combination of the partial derivatives whose scalars arise from the selection of $\overline{Y}$:

$$\overline{X}_i = \sum_{j=1}^{m} \overline{Y}_j \frac{\partial Y_j}{\partial X_i}$$

In the formula above, setting $\overline{Y}_j = 1$, yields the derivatives for free. Sample computational graphs for a desired objective function and its derivatives are shown in Figure 1.



(a) Computational graph (b) Adjoint differentiation graph

Figure 1: Adjoint and computational graph for the function $f : \mathbb{R}^3 \to \mathbb{R}^2$ defined by $f(x_1, x_2, x_3) = \left(2\log x_1 x_2 + 2\sin x_1 x_2\,,\; 4\log^2 x_1 x_2 + \cos x_1 x_3 - 2x_3 - x_2\right)$. Adapted from Capriotti et al [1].

For a scalar-valued price function, the summation can be dropped to obtain the desired sensitivities. For this paper's class of price functions, it can be proven that the cost of computing arbitrarily many derivatives with AAD is at most four times the cost of computing the price function. This bound indicates that a robust implementation of AAD would not be affected by the need to compute many sensitivities at once [14].

2.2 Using AAD to Compute Option Sensitivities

The ATLAS team was interested in computing the value for delta, the derivative with respect to equity, for European and Asian options[2]. These options were chosen on the basis of their computational complexity: the former has a closed-form solution, whereas the latter is exotic and path-dependent [15].

To price both options, fully vectorised and optimised Numpy Python [16] code was written to perform Monte Carlo pricing on both options[3]. As a benchmark, two-sided finite difference Monte Carlo with initial stock price perturbations was performed. AAD was implemented using the Autograd Python library [17]. Furthermore, the ATLAS team tested whether Autograd's implementation was able to compute multiple partial derivatives in sublinear time.

Finally, all the C++ implementations of algorithmic implementation which had support for Linux and Windows systems were tested to determine whether the computation of option and liability sensitivities yielded the same performance changes as the Python implementation.

2.3 Results

For this experiment, all objective functions were computed using Monte Carlo simulation, even when closed-form solutions were available, as this allowed us to stress-test the algorithm with a high number of computations. For all options, two test scenarios with 100 000 and 1 000 000 path simulations were performed. For the European option simulation, a one-year option on a single stock was modelled having initial underlying price and strike price were $100.00, at a 5% risk-free interest rate, and 20% volatility (see Appendix

[2]A European option is a stock option which may only be exercised on the contract's expiration date. An Asian option is a stock option whose payoff is determined by the average underlying stock price. These tend to be classified under "vanilla" and "exotic" options, respectively, given their path dependency or payoff computation complexity.

[3]Code provided in Appendices A and B for reproducibility.

A). The compilation of the Autograd AAD wrapper function took 26.5μs. At derivative runtime, a value for the equity sensitivity was obtained in 3.61s using AAD with 1 000 000 simulations, compared with 8.79s with a two-sided finite-difference estimation, which equates to a 2.43x boost in speed. Additionally, derivative estimates with AAD are precise up to floating-point precision; finite-difference estimates are not. As expected, runtime complexity increased linearly with respect to number of Monte Carlo paths simulated.

For the second experiment, a model for both arithmetic and geometric Asian options was built. As initial parameters were a stock price of $45.00, a strike price of $42.00, with a 20% volatility, 1.5% dividend payments, and 1 year maturity, with averaging occurring at at ten time steps (see Appendix B). For this experiment, only 100 000 simulations were performed. As in the previous instrument, derivatives were computed with respect to the underlying price; in addition, the derivative with respect to strike was obtained using the methods outlined above. Using two-sided finite-difference estimation, delta was determined in 243ms, whereas AAD computed delta in 84.2ms, which equates to a 2.89x speed boost. Surprisingly, computing the two desired derivatives only added a 0.2ms overhead for a geometric option, and none for an arithmetic option, which suggests not only a sublinear complexity with respect to number of inputs, but rather a constant runtime complexity. Given that with two-sided finite-difference the complexity is linear in the number of derivatives computed, it is not surprising that, in this computation, users observed up to a 4.76x boost in speed using AAD. Models were then refined to take derivatives with respect to interest rates and volatility; however, Autograd's implementation failed to do so. It is suspected that this is because Autograd was highly strained by the large number of products involved in computing these derivatives.

When transferring these findings to C++, the language in which the ATLAS platform is built, seven different libraries were tested; all of them were downloaded from `www.autodiff.org`, a website which compiles the state-of-the-art in AAD research [18]. The two instruments above were modelled, with the addition of an insurance product called a guaranteed

minimum accumulation benefit. All the tools tested required significant source code modification and did not lead to performance boosts.

3 CONCLUSIONS

The computational analyses show that the ATLAS team was able to obtain anywhere between a two-fold and five-fold boost when computing derivatives using algorithmic adjoint differentiation, when benchmarked against two-sided finite-difference derivative estimation. Furthermore, AAD was able to provide estimates which were correct up to floating point precision; attempts to do that with finite-difference methods led to numerical instability. Moreover, the experiments were able to replicate the theoretical result that the cost of computing the derivative with respect to multiple inputs was sublinear in complexity.

With regards to the C++ implementation of the algorithm, similar boosts were not attained. The most likely explanation is that the significant source code modifications required for this algorithm to be implemented led to areas where performance could not be optimised in a fashion similar to the Autograd implementation. This finding reveals that although the algorithm delivers as promised, significant language barriers may be faced when used in production-ready code bases.

4 RECOMMENDATIONS

In a business where advantages in computation speed may lead to significant gains in revenue, research and development into these techniques has the potential to reap large returns on investment and produce an edge over a company's competitors. While the code developed for this investigation cannot be readily deployed in the ATLAS database, it is certainly valuable for posteriority in one of three cases: (*i*) there is significant development in compile-time source code modifications which allow an implementation as simple as the Python implementation in C++, (*ii*) the ATLAS team decides that these boosts are significant enough to rewrite the code base with this algorithm as its sensitivity basis, or (*iii*) the ATLAS team decides to build a Python interface for their users, which can use AAD

for derivative computations.

References

[1] Luca Capriotti and Michael Giles. Adjoint greeks made easy. *Risk*, 25(9):92, 2012.

[2] Marco Avellaneda and Roberta Gamba. Conquering the greeks in monte carlo: Efficient calculation of the market sensitivities and hedge-ratios of financial assets by direct numerical simulation. In *Mathematical Finance—Bachelier Congress 2000*, pages 93–109. Springer, 2002.

[3] Walter Rudin et al. *Principles of mathematical analysis*, volume 3. McGraw-hill New York, 1976.

[4] John C Hull. *Options futures and other derivatives*. Pearson Education India, 2003.

[5] Pao-Liu Chow. *Stochastic partial differential equations*. Chapman and Hall/CRC, 2007.

[6] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637–654, 1973.

[7] Phelim P Boyle. Options: A monte carlo approach. *Journal of financial economics*, 4(3):323–338, 1977.

[8] Jeffrey S Rosenthal. *A first look at rigorous probability theory*. World Scientific Publishing Company, 2006.

[9] Mike Giles and Paul Glasserman. Smoking adjoints: Fast monte carlo greeks. *Risk*, 19(1):88–92, 2006.

[10] Cristian Homescu. Adjoints and automatic (algorithmic) differentiation in computational finance. 2011.

[11] Wei Xu, Xi Chen, and Thomas Coleman. The efficient application of automatic differentiation for computing gradients in financial applications. 2014.

[12] Samuel Embaye. The determination of structured hessian matrices via automatic differentiation. 2014.

[13] Jiayi Zheng. Efficient greek estimation for variable annuities using monte carlo simulation. 2017.

[14] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[15] Justin London. *Modeling derivatives in C++*, volume 263. John Wiley & Sons, 2005.

[16] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[17] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[18] Autodiff Organisation. Autodiff, 1999.

Appendix A: AAD for a European Option

A.1 Pricing functions

Below we compute the Monte Carlo prices for European put and call options.

```
In [1]: import numpy as np
        import autograd.numpy as anp
        from autograd import grad

In [2]: def np_monte_carlo_call_price(S, K, r, v, T, num_sims, rnorms):
            S_adjust = S * anp.exp(T * (r - 0.5 * v * v))

            S_cur = S_adjust * anp.exp(anp.sqrt(v*v*T) * rnorms)
            pay_vec = S_cur - K
            pay_vec_max = anp.maximum(pay_vec, 0)

            return anp.exp(-r * T) * anp.mean(pay_vec_max)


        def np_monte_carlo_put_price(S, K, r, v, T, num_sims, rnorms):
            S_adjust = S * anp.exp(T * (r - 0.5 * v * v))

            S_cur = S_adjust * anp.exp(anp.sqrt(v*v*T) * rnorms)
            pay_vec = K - S_cur
            pay_vec_max = anp.maximum(pay_vec, 0)

            return anp.exp(-r * T) * anp.mean(pay_vec_max)
```

A.2 Monte Carlo Simulation

In the following cells we compute the MC estimates for Black-Scholes Call and Put options and their delta.

```
In [3]: num_sims = 100000   # Number of simulated asset paths
        S = 100.0  # Option price
        K = 100.0  # Strike price
        r = 0.05   # Risk-free rate (5%)
        v = 0.2    # Volatility of the underlying (20%)
        T = 1.0    # One year until expiry
        rnorms = anp.random.normal(size = num_sims)

        c = np_monte_carlo_call_price(S, K, r, v, T, num_sims, rnorms)
        # p = monte_carlo_put_price(S, K, r, v, T, num_sims, rnorms)

In [4]: %%time

        ## Delta calculation via numpy Monte Carlo

        num_sims = 100000000   # Number of simulated asset paths
        S = 100.0  # Option price
        K = 100.0  # Strike price
        r = 0.05   # Risk-free rate (5%)
        v = 0.2    # Volatility of the underlying (20%)
        T = 1.0    # One year until expiry
        rnorms = anp.random.normal(size = num_sims)
```

```
        equity_shock = 0.01

        c1 = np_monte_carlo_call_price(S, K, r, v, T, num_sims, rnorms)
        c2 = np_monte_carlo_call_price(S + equity_shock, K, r, v, T, num_sims, rnorms)

        delta = (c2 - c1) / equity_shock
        print("Call delta: {0}".format(delta))
```

```
Call delta: 0.6369429486317912
CPU times: user 7.11 s, sys: 1.68 s, total: 8.79 s
Wall time: 8.79 s
```

A.3 AAD for Black-Scholes European Option

Now we attempt to use Autograd's AAD to compute the delta.

```
In [5]: %%time
        delta_grad = grad(np_monte_carlo_call_price)
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 26.5 μs
```

```
In [6]: %%time
        delta_grad(S, K, r, v, T, num_sims, rnorms)
```

```
CPU times: user 1.96 s, sys: 1.64 s, total: 3.6 s
Wall time: 3.61 s
```

```
Out[6]: 0.6368491159470586
```

Appendix B: AAD for Arithmetic and Geometric Asian Options

Here we price an Asian option under Monte Carlo and compute the Greeks via Monte Carlo and Autograd. The first implementation is taken from the internet. The second and third are taken from London's Modelling Derivatives in C++.

B.1 Monte Carlo Pricing function for a Geometric Asian Option

```
In [57]: import numpy as np
         import autograd.numpy as anp
         from autograd import grad
         from math import exp, sqrt
```

```
In [58]: def np_mc_geom_asian_call(price, strike, vol, rate, div, T, rnorms):
             N = np.ma.size(rnorms, axis=1)
             M = np.ma.size(rnorms, axis=0)
             deltat = T/N
             mu = rate - div - 0.5*vol*vol
             transformed_norms = anp.exp(mu * deltat + vol * anp.sqrt(deltat) * rnorms)
             cumprod_norms = anp.power(anp.cumprod(transformed_norms, axis = 1), 1/N)

             G = price * anp.prod(cumprod_norms, axis = 1)
             payoff = anp.maximum(G - strike, 0)
             payoff_sum = anp.sum(payoff) / (M-1)

             return anp.exp(-rate * T) * payoff_sum
```

B.2 Monte Carlo Simulation

We shock the equity by $\epsilon = 0.01$.

```
In [59]: price = 45.0
         strike = 42.0
         vol = 0.2
         r = 0.055
         div = 0.015
         T = 1.0
         rnorms = np.random.normal(size = (100000, 10))
```

```
In [60]: %%time
         equity_shock = 0.01
         c1 = np_mc_geom_asian_call(price - equity_shock, strike, vol, r, div, T, rnorms)
         c3 = np_mc_geom_asian_call(price, strike, vol, r, div, T, rnorms)
         c2 = np_mc_geom_asian_call(price + equity_shock, strike, vol, r, div, T, rnorms)
         delta = (c2 - c1) / (2 * equity_shock)
         print("Asian Call MC delta: {0}".format(delta))
```

```
Asian Call MC delta: 0.7516358054711958
CPU times: user 232 ms, sys: 12 ms, total: 244 ms
Wall time: 243 ms
```

B.3 AAD for Geometric Asian Option

Now we use Autograd's AAD wrapper to compute delta.

```
In [61]: delta_grad = grad(np_monte_carlo_geom_asian_call_option)

In [62]: %%time
         delta_grad(price, strike, vol, r, div, T, rnorms)

CPU times: user 76 ms, sys: 8 ms, total: 84 ms
Wall time: 84.2 ms


Out[62]: 0.7516594727131297

In [63]: nabla_price = grad(np_mc_geom_asian_call, (0,1))

In [64]: %%time
         nabla_price(price, strike, vol, r, div, T, rnorms)

CPU times: user 80 ms, sys: 4 ms, total: 84 ms
Wall time: 84.4 ms


Out[64]: (array(0.75165947), array(-0.70134304))

In [65]: %%time
         eq_shock = 0.01
         st_shock = 0.01
         base = np_mc_geom_asian_call(price, strike, vol, r, div, T, rnorms)
         eq_shock1 = np_mc_geom_asian_call(price + eq_shock, strike, vol, r, div, T, rnorms)
         eq_shock2 = np_mc_geom_asian_call(price - eq_shock, strike, vol, r, div, T, rnorms)
         st_shock1 = np_mc_geom_asian_call(price, strike + st_shock, vol, r, div, T, rnorms)
         st_shock2 = np_mc_geom_asian_call(price, strike - st_shock, vol, r, div, T, rnorms)

         delta_MC = (eq_shock1 - eq_shock2) / (2 * eq_shock)
         dPdK = (st_shock1 - st_shock2) / (2 * st_shock)

         print("Delta = {0}".format(delta))
         print("dP/dK = {0}".format(dPdK))

Delta = 0.7516358054711958
dP/dK = -0.7013179239083289
CPU times: user 388 ms, sys: 16 ms, total: 404 ms
Wall time: 402 ms
```

B.4 AAD for Arithmetic Asian Option

Now we use Autograd's AAD wrapper to compute the gradient and test whether AAD is sublinear in the number of input parameters.

```
In [66]: def np_mc_arith_asian_call(price, strike, vol, rate, div, T, rnorms):
             N = np.ma.size(rnorms, axis=1)
             M = np.ma.size(rnorms, axis=0)
             deltat = T/N
             mu = rate - div - 0.5*vol*vol
             transformed_norms = anp.exp(mu * deltat + vol * anp.sqrt(deltat) * rnorms)
             cumprod_norms = anp.cumprod(transformed_norms, axis = 1) #k_1 ... k_N
```

```
            S_vec = price * cumprod_norms
            A_mean = anp.mean(S_vec, axis = 1)

            payoff = anp.maximum(A_mean - strike, 0)
            payoff_sum = anp.sum(payoff) / (M-1)

            return anp.exp(-rate * T) * payoff_sum
```

```
In [67]: np_mc_arith_asian_call(price, strike, vol, r, div, T, rnorms)
```

```
Out[67]: 4.47829537474829
```

```
In [68]: delta_arith_grad = grad(np_mc_arith_asian_call)
```

```
In [69]: %%time
         delta_arith_grad(price, strike, vol, r, div, T, rnorms)
```

```
CPU times: user 12 ms, sys: 16 ms, total: 28 ms
Wall time: 29.1 ms
```

```
Out[69]: 0.7596875715493492
```

```
In [70]: # Partial derivatives with respect to 1st and 2nd arguments
         nabla_arith = grad(np_mc_arith_asian_call, (0,1))
```

```
In [71]: %%time
         nabla_arith(price, strike, vol, r, div, T, rnorms)
```

```
CPU times: user 12 ms, sys: 16 ms, total: 28 ms
Wall time: 29.1 ms
```

```
Out[71]: (array(0.75968757), array(-0.70732489))
```

```
In [72]: %%time
         eq_shock = 0.01
         st_shock = 0.01
         base = np_mc_arith_asian_call(price, strike, vol, r, div, T, rnorms)
         eq_shock1 = np_mc_arith_asian_call(price + eq_shock, strike, vol, r, div, T, rnorms)
         eq_shock2 = np_mc_arith_asian_call(price - eq_shock, strike, vol, r, div, T, rnorms)
         st_shock1 = np_mc_arith_asian_call(price, strike + st_shock, vol, r, div, T, rnorms)
         st_shock2 = np_mc_arith_asian_call(price, strike - st_shock, vol, r, div, T, rnorms)

         delta_MC = (eq_shock1 - eq_shock2) / (2 * eq_shock)
         dPdK = (st_shock1 - st_shock2) / (2 * st_shock)

         print("Delta = {0}".format(delta))
         print("dP/dK = {0}".format(dPdK))
```

```
Delta = 0.7516358054711958
dP/dK = -0.7073384793111703
CPU times: user 44 ms, sys: 24 ms, total: 68 ms
Wall time: 67.7 ms
```