SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

GREEKS COMPUTATION WITH AUTOMATIC DIFFERENTIATION

MSc. Probability & Finance
Numerical Probability Project

Feb. - Apr. 2020

Author
Matthieu CHARRIER
Student at MSc. Probability & Finance
matthieucharrier1994@gmail.com

Contents

1 Abstract **2**

2 Introduction **2**

3 Pathwise Differentiation Method **2**

4 Automatic Differentiation **3**
4.1 Theory 4
4.2 Tangent Mode 4
4.2.1 Tangent unimode 5
4.2.2 Tangent multimode 5
4.3 Adjoint Mode 6

5 Implementation strategies **6**
5.1 Tangent Mode Implementation - operator overloading 7
5.2 Adjoint Mode implementation - tape based implementation 8

6 Application **10**
6.1 Basket Option 10
6.2 Best Of Asian Option 15

7 Correlation greeks **19**

8 Conclusion **21**

1 Abstract

From February to April 2020, as part of the course "Numerical Probability for Finance" taught by Gilles Pages and Vincent Lemaire, I have chosen to work on the fast greeks computation using automatic differentiation tools which the guidelines was given in french by the paragraph below.
Last year during my experience as a quantitative researcher intern in a fintech, my supervisor told me that one of their former interns, as he was working on path dependent volatility models, had to use algorithmic differentiation tools.
Interesting discussions around this technology attracted my curiosity and that is why I used this opportunity to explore its performance.

"Calcul des sensibilites par differentiation automatique."

- Etudier la methode numerique introduit dans [1] (cf. aussi [2]) pour calculer des sensibilites par la methode du flot (ou processus tangent) en utilisant une technique de differentiation automatique (algorithm differentiation).
- Verifier numeriquement l'approche en insistant sur le cout numerique (la complexite). Il est possible d'appeler ou d'adapter des librairies existantes de differentiation automatique (cf. le site http://www.autodiff.org)

2 Introduction

In financial modeling and under complex models, closed formula hardly ever occur to price derivatives.
Hence, one has to rely on either approximations or Monte Carlo method whose several acceleration techniques have been provided along the years.
However, to hedge their positions, the traders need to assess not only the price but also the sensitivities with respect to each model parameter.
In the case of options on several names, the computational cost strongly increases with the number of assets.
Nowadays, financial engineering provides structured derivatives composed by hundred of names and quantitative models include a lot of parameters.
A naive use of this technique can make the computational efficiency not convincing enough to roll it out.
In this paper, one introduces algorithmic differentiation principles.
How can it be implemented in an efficient way ?
How useful it is to speed Monte Carlo sensitivities computation ?
Eventually, one shows with some examples how fast greeks calculation is when you involve algorithmic differentiation tools to your program.

3 Pathwise Differentiation Method

For a given financial model, prices of European options usually writes as:

$$V(\theta) = \mathbb{E}^{\mathbb{Q}}\left[P(\theta, X_{T_1}(\theta), X_{T_2}(\theta), ..., X_{T_M}(\theta)\right]$$

where:

- $\mathbb{Q}$ denotes the risk neutral measure, that is to say, a probability under which the discounted assets $\tilde{X}$ are martingales.
- $\theta = (\theta_n)_{1 \leq n \leq N}$ denotes the vector of model parameters.
- $X = (X^d)_{1 \leq d \leq D}$ denotes the random variables that model the price vector of the assets
- $T = (T_m)_{1 \leq m \leq M}$ denotes the set of fixing dates
- $P : \mathbb{R}^N \times (\mathbb{R}^D)^M \longrightarrow \mathbb{R}_+$ denotes the payout function.

However, in most of financial models, one does not have access to a closed form formula.
Monte Carlo method seems to be the right tool to deal with this issue.
One remains that the Monte Carlo estimator is written as:

$$\hat{P}^{N_{MC}} = \frac{1}{N_{MC}} \sum_{i=1}^{N_{MC}} P(\theta, X_{T_1}^i(\theta), X_{T_2}^i(\theta), ..., X_{T_M}^i(\theta))$$

With a confidence interval given by:

$$\hat{I}_{N_{MC}}(\alpha) = \left[\hat{P}_{N_{MC}} - \frac{\phi^{-1}(1-\frac{\alpha}{2})}{\sqrt{N_{MC}}} \Sigma, \hat{P}_{N_{MC}} + \frac{\phi^{-1}(1-\frac{\alpha}{2})}{\sqrt{N_{MC}}} \Sigma \right]$$

where $\Sigma = \mathbb{V}ar^{\mathbb{Q}} \left[P(\theta, X_{T_1}(\theta), X_{T_2}(\theta), ..., X_{T_M}(\theta) \right]$

But in order to hedge positions against the variations of market parameters, financial modeling theory require the access to $\nabla V(\theta)$.

The first solution is well known as "bumping":
For each $k = 1...N$, one compute:

$$\frac{V(\theta_1, \theta_2, ..., \theta_k + \delta, ..., \theta_N) - V(\theta_1, \theta_2, ..., \theta_k, ..., \theta_N)}{\delta}.$$

Which implied the calculation of two expectations and an overuse of the random number generator.
The second solution is called pathwise differentiation method:
If one assume that P has good properties of integrability and differentiability, it yields:

$$\nabla V(\theta) = \mathbb{E}^{\mathbb{Q}} \left[\nabla P(\theta, X_{T_1}(\theta), X_{T_2}(\theta), ..., X_{T_M}(\theta) \right]$$

In the following lines, let us denote $\bar{\theta}_k = \partial_{\theta_k} V = \mathbb{E}^{\mathbb{Q}} \left[\partial_{\theta_k} P \right]$

According to the chain rule, one can write:

$$\partial_{\theta_k} P(\theta, X_{T_1}(\theta), X_{T_2}(\theta), ..., X_{T_M}(\theta)) = \sum_{i=1}^{M} \partial_{x_i} P \times \partial_{\theta_k} X_{T_i} + \partial_{\theta_k} P$$

On the one hand, in most of models one does not have access to the tangent state vector/matrix $\partial_{\theta_k} X_{T_i}$ and generally one must use finite difference method, its computational cost strongly depends on the model. For the sake of simplicity, one lead the calculus under Black Scholes model and leave the case of other model as further works.

On the other hand, in the previous expression, P is analytically known but one can encounter in financial industry some payout function that involves hundred of assets and which the analytical expression of the derivative can be cumbersome to compute. For this reason, one is used to process finite differences method:

$$\begin{cases} \partial_{x_i} P \approx \frac{P(\theta, x_1, x_2, ..., x_i + \delta, ..., x_D) - P}{\delta} \\ \partial_{\theta} P \approx \frac{P(\theta + \delta, x_1, x_2, ..., x_D) - P}{\delta} \end{cases}$$

The main drawbacks lies on the multiple evaluation of the payout function, keeping in mind that the option can be written on hundred names, the computational burden of this part might fall the pathwise differentiation method apart.
Furthermore, it strongly lacks of accuracy, the traders has to set the δ parameters which is not user friendly for them.
The tools one explains in the next part represents a natural and simple manner to overcome these disadvantages.

4 Automatic Differentiation

Automatic Differentiation (AD) defines a class of algorithms which aim to compute accuratly the derivatives of a computer program with respect to its inputs. As one will discuss later, there exists two main methods to proceed the calculation: the tangent (forward) mode (TM) and the adjoint (backward) mode (AM).

4.1 Theory

For real-valued $\mathcal{C}^1$ function which analytical expression is explicitly known, the differentiation is straightforward and can be computed analytically. Let us suppose that all we have is a computer program that takes n inputs parameters $(x_1, x_2, ..., x_n) \in \mathbb{R}^n$ and one output $y \in \mathbb{R}$:

$$F : \begin{cases} \mathbb{R}^n \longrightarrow \mathbb{R} \\ (x_1, x_2, ..., x_n) \longrightarrow y = F(x_1, x_2, ..., x_n) \end{cases}$$

Our goal is to compute for a given input $(x_1, x_2, ..., x_n)$, $y = F(x_1, x_2, ..., x_n)$ and $\partial_{x_i} F(x_1, x_2, ..., x_n)$ for every $i \in [1, n]$ simultaneously.

One can represent the execution of the program F as the following steps: Let:

$$w_1 = x_1, w_2 = x_2, ..., w_n = x_n$$

For a given computer program F, there exists a sequence of functions $(\phi_i : \mathbb{R}^{n+i-1} \longrightarrow \mathbb{R})_{1 \leq i \leq N}$ such as:

$$\begin{cases} w_{n+1} = \phi_1(w_1, w_2, ..., w_n) \\ w_{n+2} = \phi_2(w_1, w_2, ..., w_n, w_{n+1}) \\ ... \\ w_{n+i} = \phi_i(w_1, w_2, ..., w_{n+i-1}) \\ ... \\ y = w_N = \phi_N(w_1, w_2, ..., w_{N-1}) \end{cases}$$

Here is a simple example with the function $f(x, y) = x \times y + \sin(x)$:

Example

- $w_1 = x$
  $w_2 = y$
- $w_3 = w_1 \times w_2$
  $w_4 = \sin(w_2)$
- $w_5 = w_3 + w_4$

4.2 Tangent Mode

The tangent mode of AD is based on the forward propagation of the chain rule, more precicely: For all $i \in [1, ..., N]$:

$$\nabla w_i = \sum_{j=1}^{i-1} \partial_{w_i} w_j \times \nabla w_j$$

If we denote by $D_{i,j} = \partial_{w_i} w_j = \partial_{w_i} \phi_j(w_1, w_2, ..., w_{j-1})$ the arc derivatives with respect to the link (i, j), it yields:

$$\begin{cases} \forall 1 \leq i \leq n, \nabla w_i = e_i \\ \forall n+1 \leq i \leq N, \nabla w_i = \displaystyle\sum_{j=1}^{i-1} D_{i,j} \times \nabla w_j \end{cases}$$

where e_i denotes the i-th vector of canonical basis of $\mathbb{R}^n$.

In fact, one simply applies progressively the usual rules to compute the derivative of the function. First, we need to compute the arc derivatives $D_{i,j}$ for each link.

Example

- $w_1 = x$
  $w_2 = y$
- $w_3 = w_1 \times w_2$, $D_{1,3} = w_2$, $D_{2,3} = w_1$ $w_4 = \sin(w_2)$, $D_{2,4} = -\cos(w_2)$
- $w_5 = w_3 + w_4$, $D_{3,5} = 1$, $D_{4,5} = 1$

Once we get them, we apply the chain rule forwardly to obtain the derivatives with the standard tangent mode:

Example

- $\nabla w_1 = e_1$
  $\nabla w_2 = e_2$
- $\nabla w_3 = D_{1,3}e_1 + D_{2,3}e_2 = w_2 e_1 + w_1 e_2$
  $\nabla w_4 = D_{2,4}e_2 = -\cos(w_2)e_2$
- $\nabla w_5 = D_{3,5}\nabla w_3 + D_{4,5}\nabla w_4 = \nabla w_3 + \nabla w_4 = w_2 e_1 + (w_1 - \cos(w_2)e_2$

However, one rather use two variants of the tangent mode:

4.2.1 Tangent unimode

Suppose one wish to get as output a linear combination of the partial derivatives, that is to say $\sum_{i=1}^{n} \lambda_i \partial_{x_i} F = \lambda \cdot \nabla F$ for a given vector $\lambda \in \mathbb{R}^n$. If we denote the tangents $\dot{w_i} = \lambda \cdot \nabla w_i$, it yields:

$$\begin{cases} \forall 1 \leq i \leq n, \dot{w_i} = \lambda_i \\ \forall n+1 \leq i \leq N, \bar{w_i} = \sum_{j=1}^{i-1} D_{i,j} \times \dot{w_j} \end{cases}$$

This way of presenting has the advantage to work with scalar data types and provide a linear combinaison of the derivatives which is precisely what we need to compute the sensitivity of the payout function with respect to one parameter.

In our example, suppose one has chosen $\lambda = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$.

Example

- $\dot{w_1} = 2$
  $\dot{w_2} = 1$
- $\dot{w_3} = 2D_{1,3} + D_{2,3} = 2w_2 + w_1$
  $\dot{w_4} = D_{2,4} = -\cos(w_2)$
- $\dot{w_5} = D_{3,5}\dot{w_3} + D_{4,5}\dot{w_4} = \dot{w_3} + \dot{w_4} = 2w_2 + w_1 - \cos(w_2)$

4.2.2 Tangent multimode

In order to compute in one shot the sensitivity of the payout with respect to multiple parameters, one relies on the tangent multimode. If we denote the multi tangent $\Omega_i = \Lambda \cdot \nabla w_i$ for a given matrix $\Lambda \in \mathcal{M}_{n,p}(\mathbb{R})$, one has:

$$\begin{cases} \forall 1 \leq i \leq n, \Omega_i = C_i(\Lambda) \\ \forall n+1 \leq i \leq N, \Omega_i = \sum_{j=1}^{i-1} D_{i,j} \times \Omega_j \end{cases}$$

In this way, one has in a row any linear combinaison of the derivatives one wants.
Let us get back to our example, one would like to add a linear combination with weights $\begin{pmatrix}3\\4\end{pmatrix}$. On set $\Lambda = \begin{pmatrix}1 & 3\\2 & 4\end{pmatrix}$

Example

- $\Omega_1 = \begin{pmatrix}1\\2\end{pmatrix}$
  $\Omega_2 = \begin{pmatrix}3\\4\end{pmatrix}$
- $\Omega_3 = \begin{pmatrix}2D_{1,3} + D_{2,3}\\4D_{1,3} + 3D_{2,3}\end{pmatrix}$
  $\Omega_4 = \begin{pmatrix}D_{2,4}\\3D_{2,4})\end{pmatrix} = \begin{pmatrix}-\cos(w_2)\\-3\cos(w_2)\end{pmatrix}$
- $\Omega_5 = D_{3,5}\Omega_3 + D_{4,5}\Omega_4 = \Omega_3 + \Omega_4 = \begin{pmatrix}2w_2 + w_1 - \cos(w_2)\\3w_2 + 2w_1 - 2\cos(w_2)\end{pmatrix}$

Clearly, the multi tangent mode of AD finds all its interests when the aim is to compute several linear combinaison of derivatives. In particularly when one wishes to get several sensitivities in a row.

4.3 Adjoint Mode

In the adjoint mode of AD, one defines the adjoints as $\bar{w}_i = \lambda\partial_{w_i} y$ for a given parameter $\lambda \in \mathbb{R}$. The idea is to propagate the chain rule backward. it gives:

$$\begin{cases}\bar{y} = \lambda\\ \forall 1 \leq i \leq N-1, \bar{w}_i = \sum_{j=i+1}^{N} D_{i,j}\bar{w}_j\end{cases}$$

The algorithm outputs $\bar{x}_i = \lambda\partial_{x_i} y$ for all the input variables.

Example

- $\bar{w}_5 = \lambda$
- $\bar{w}_3 = D_{3,5}\dot{w}_5 = \lambda$
  $\bar{w}_4 = D_{4,5}\dot{w}_5 = \lambda$
- $\bar{w}_1 = D_{1,3}\bar{w}_3 = \lambda w_2$
  $\bar{w}_2 = D_{2,3}\bar{w}_3 + D_{2,4}\bar{w}_4 = \lambda w_1 - cos(w_2)\lambda$

One notices that the adjoint mode is well adapted for payout function with several inputs. However, unlike tangent mode, a forward sweep is required to store the arc derivatives before moving backward and propagate the chain rule. One will see that it implies additional challenges when it deals with efficient implementation strategy.

5 Implementation strategies

The website http://www.autodiff.org proposes external libraries to use algorithmic differentiation tools. Nethertheless, for the sake of transparency and challenge, one decides to implement our own AD tools library.
This part holds the main challenge of the project. Indeed, I was particularly dedicated to produce efficient and user friendly automatic differentiation algorithms and organise a clean and rough source code.
There exists two manners to build automatic differentiation tools: source code transformation and operator overloading.
Although source code transformation can accelerate the processing time, for the sake of flexibility, one decided

to use operator overloading.
Furthermore, this choice is all the more coherent that C++ language fully unleash the power of operator overloading and oriented object programming.

Before implement the classes *tgt_double* and *adj_double*, in order to be coherent with the notations, one has decided to encapsulate the standard data type *double* in a kind of a type *std_double*.

5.1 Tangent Mode Implementation - operator overloading

What makes tangent mode implementation so innate is the forward propagation of the chain rule: One simply applies step by step the rules of differentiation taught in high school. This is a kind of standard data type overloading in the sense that one create a class with two attributes: a real data x and its tangent dx. Indeed, the inputs of the function to differentiate are replaced by a *tgt_double* which contains the current value and its current derivative. In order to use the power of multi mode, one imposes the attributes dx to be a vector-valued data type: each element represents the coefficient of the linear combination associated to the variable x.

***tgt_double* class example**

```
class tgt_double
{
    using value_type = double;
    using derivative_type = vec;
    private:
        value_type x;
        derivative_type dx;
}
```

From this stage, all one need to do is to overload the binary arithmetic operations +, -, ×, / not only between instances of the class *tgt_double* but also between *std_double* and *tgt_double*. It consists in re-implementing these operations and add its counterparts for the tangents. The same thing was carried out with the unary arithmetic operations such as *+=, -=, *=* and unary functions such as *sin, cos, exp*

Binary operation example

```
inline friend tgt_double operator* (tgt_double const& u,
tgt_double const& v)
{
        return tgt_double (u.x * v.x, v.x * u.dx + u.x * v.dx)
}
```

Unary operation example

```
tgt_double operator+= (tgt_double const& v)
{
        x += v.x;
    dx += v.dx;
    return *this;
}
```

It remains to overload the standard operations on vector such as *dot, mean, max, ...* provided one has defines what is a vector/matrix of *tgt_double*.
For that purpose, one used the well known library *armadillo* and its data type *field<object>*.
Eventually, one added a standard way to declare objects of type *tgt_double*, *vector<tgt_double>* and *matrix<tgt_double>*, for instance:

Make example

```
template<typename tgt_double>
tgt_double make0 (double f, vec const& df)
{
    return tgt_double (f, df);
}

template<typename tgt_double>
vector<tgt_double> make1 (vec const& f, mat const& df)
{
    unsigned n (f.n_elem);
    vector<tgt_double> res (n);
        for (unsigned i (0); i < n; ++i)
                res (i) = make0<tgt_double> (f (i), df.col (i));
    return res;
}

template<typename tgt_double>
matrix<tgt_double> make2 (mat const& f, cube const& df)
{
    unsigned m (f.n_cols);
    matrix<tgt_double> res (m);
        for (unsigned i (0); i < m; ++i)
                res (i) = make1<tgt_double> (f.col (i), df.slice (i));
    return res;
}
```

Not all unary and binary expressions have been overloaded in the present header files but just the one used in the following payoff examples. The reader is free to do it and test the efficiency of the tangent mode of AD throughout the simple interactive file dedicated.

5.2 Adjoint Mode implementation - tape based implementation

The main obstacle here is that one need to store the information concerning the arc derivatives before proceeding to the backward sweep. First I did not see other way but the naive building of the computational graph and store the arc derivatives in each node. The reader can see the implementation of this idea in the commentaries below the source code in the header file associated. I gave up this hint because too many useless nodes are stored and it does not reflect the efficiency of adjoint mode of AD.
One starts by defining a node object by the list of its parents p and the list of its arc derivatives w, that is to say the arc derivatives that links the node to its parents.

Node structure example

```
struct Node
{
        uvec p;
        vec w;
};
```

As unary and binary operations proceeds, a tape encoded by a *vector<Node*>* data type stores the children nodes that the operations bear. At the beginning of the calculation, a tape is declared and the nodes associated to the input variables are pushed back to the tape.
additionally, for the needs one returns the current index in the tape.

Node pushing example

```
unsigned push0 ()
{
        unsigned len (nodes.size ());
        nodes.push_back (new Node ());
        return len;
}
unsigned push (uvec const& p, vec const& w)
{
        unsigned len (nodes.size ());
        nodes.push_back (new Node (p, w));
        return len;
}
```

On the other hand, one defines the object *adj_double* that overload the notion of *double*. Each *adj_double* object contains a pointer towards the tape, stores the value of the node and its index in the tape.

***adj_double* class example**

```
struct adj_double
{
        Tape* tape;
        unsigned index;
        double value;
}
```

Unary operation example

```
friend adj_double exp (adj_double const& u)
{
        double exp_u (exp (u.value));
        return
                adj_double (u.tape, u.tape->push ({u.index}, {exp_u}), exp_u);
}
```

As for the *tgt_double* class, one creates the method to generate a vector or matrix of *adj_double* object but these methods now belong to the class *Tape* In this way, the tape fills up of nodes as the operations are running on. Once one arrives at the terminal node, one has access to the value. It remains to carry out the backward sweep to get the derivatives. This action is made by the method *grad* that returns the list of the adjoints of each node, calculated step by step backwardly:

Backward propagation example

```
Grad grad (double seed = 1) const
{
vec derivatives (tape->size (), arma::fill::zeros);
auto nodes = tape->nodes;
derivatives (index) = seed;
for (unsigned i (index); i > 0; i--)
{
        auto node = nodes [i];
        auto derivative = derivatives (i);
        auto indexes = node->p;
        auto weights = node->w;
        auto it_indexes = indexes.begin ();
        auto it_weights = weights.begin ();
        compute (derivatives);
```

```
}
```

Here again, the reader is free to add any adjoints of functions that he would need and test the performance of adjoint mode of AD in the dedicated file.

6 Application

6.1 Basket Option

A basket option written on the names $S = (S_1, S_2, ..., S_d)$ with maturity T, strike K and weights w is an option that gives the right to its holder to receive the w-weighted average of S at price K.
The payout function associated writes:

$$P(r, x) = e^{-rT}((w, x) - K)_+$$

Not to make things more complex, one works within the Black Scholes framework, which allows us to use closed form formula for the tangent state vectors. Our goal here is to compare the time processing to compute the price and sensitivities with Monte Carlo method using:

- Finite Difference Method
- Tangent Multi Mode of AD
- Adjoint Mode of AD

The header files dedicated to the Monte Carlo engine, random variable and Black-Scholes class are highly inspired by the one proposed during C++ training sessions with Vincent Lemaire.
The Black Scholes tools was adapted to generate the tangent state vector, that is to say the vectors $\partial_r S = (\partial_r S_i)_{1 \leq i \leq d}$, $\partial_x S = (\partial_{x_i} S_i)_{1 \leq i \leq d}$ and $\partial_\sigma S = (\partial_{\sigma_i} S_i)_{1 \leq i \leq d}$. These data are gathered through a *features_assets* structure. On the other hand, as our objective is to compute the price, delta, vega and rho in a row, one creates a structure named *features_option* and one takes that window to overload the arithmetic operators necessary to run a Monte Carlo inspector. At this occasion, one 'templates' the *mean_var* class to deal with any data types which the corresponding operations are overloaded.

Above all, one needs to create a virtual templated class named *basketOption*. Its children corresponds to the different manners to deal with the derivatives of the payout function. From the outset, one makes sure that each method works fine and produce the same results:

Market parameters	Finite difference method	Tangent Mode of AD	Adjoint Mode of AD
Price	23.5996	23.5014	23.4674
Δ_1	0.0495	0.0495	0.0495
Δ_2	0.0496	0.0494	0.0495
γ_1	1.3503	1.3380	1.3364
γ_2	1.3458	1.3382	1.3320
ρ	29.1427	29.1012	29.1382

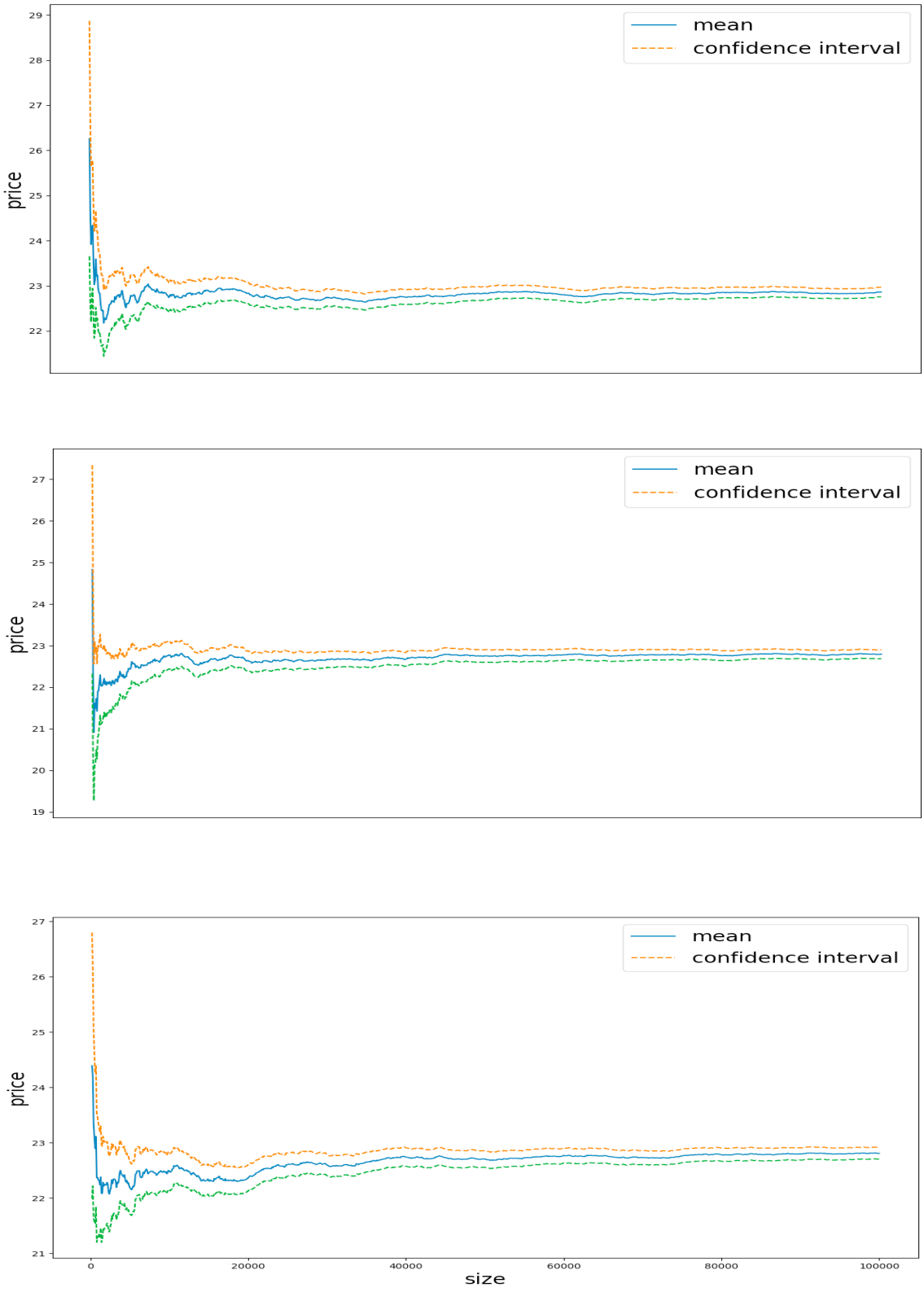


Figure 0.1: **Basket option price using (top to bottom) finite difference method, tangent mode and adjoint mode**
Number of assets: 2, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 100, *volatilities*: 30%, *initial spots*: 100, *risk free rate*: 10%, *weights*: equally distributed, *Correlation*: 25%

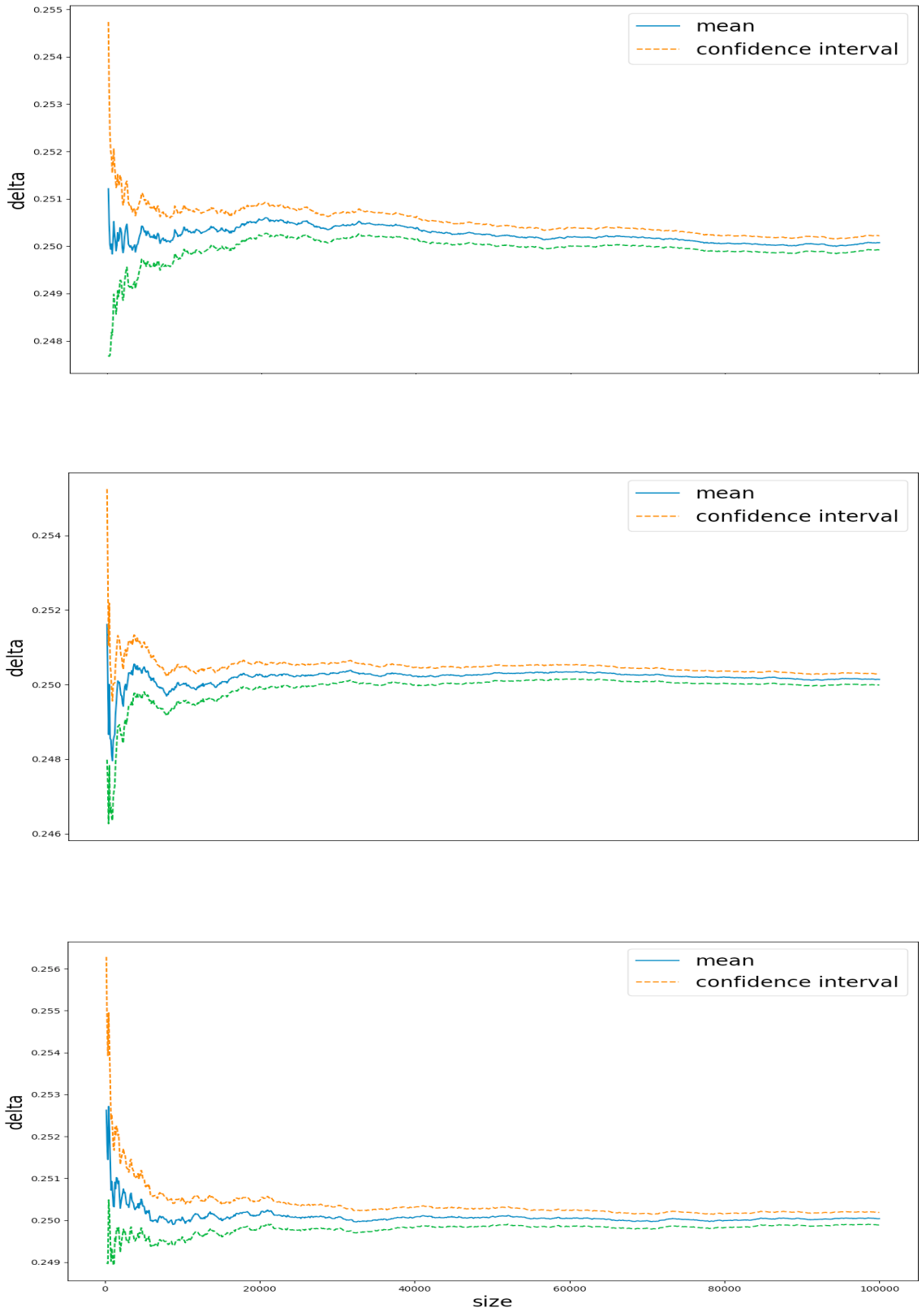


Figure 0.2: **Basket option delta using (top to bottom) finite difference method, tangent mode and adjoint mode**
Number of assets: 4, *Size*: 10^5 samples, *maturity*: 2 yr, *strike*: 95, *volatilities*: 10%, *initial spots*: 120, *risk free rate*: 20%, *weights*: equally distributed, *Correlation*: 25%

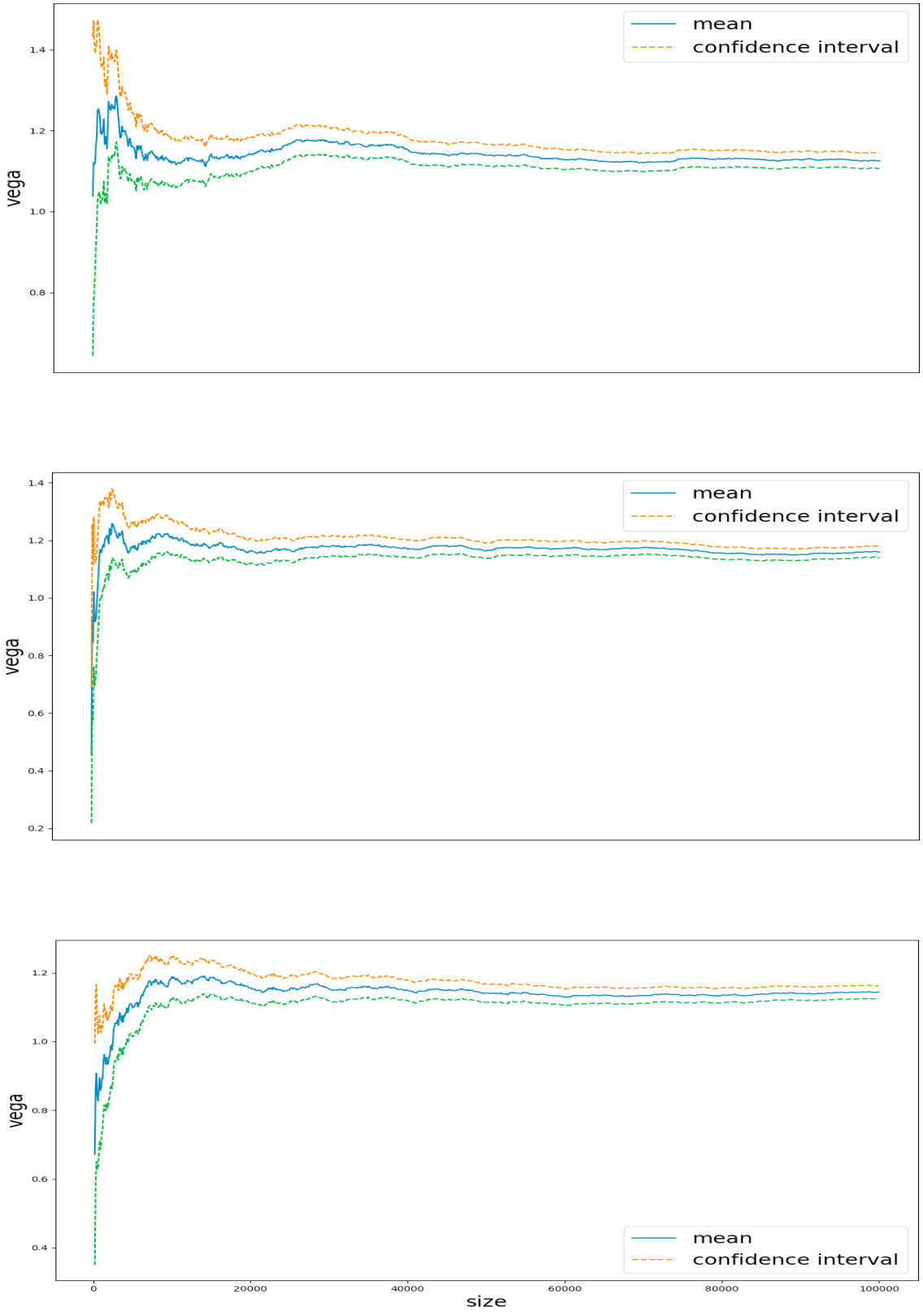


Figure 0.3: **Basket option vega using (top to bottom) finite difference method, tangent mode and adjoint mode**
Number of assets: 8, *Size*: 10^5 samples, *maturity*: 6 mth, *strike*: 110, *volatilities*: 10%, *initial spots*: 100, *risk free rate*: 1%, *weights*: equally distributed, *Correlation*: 75%

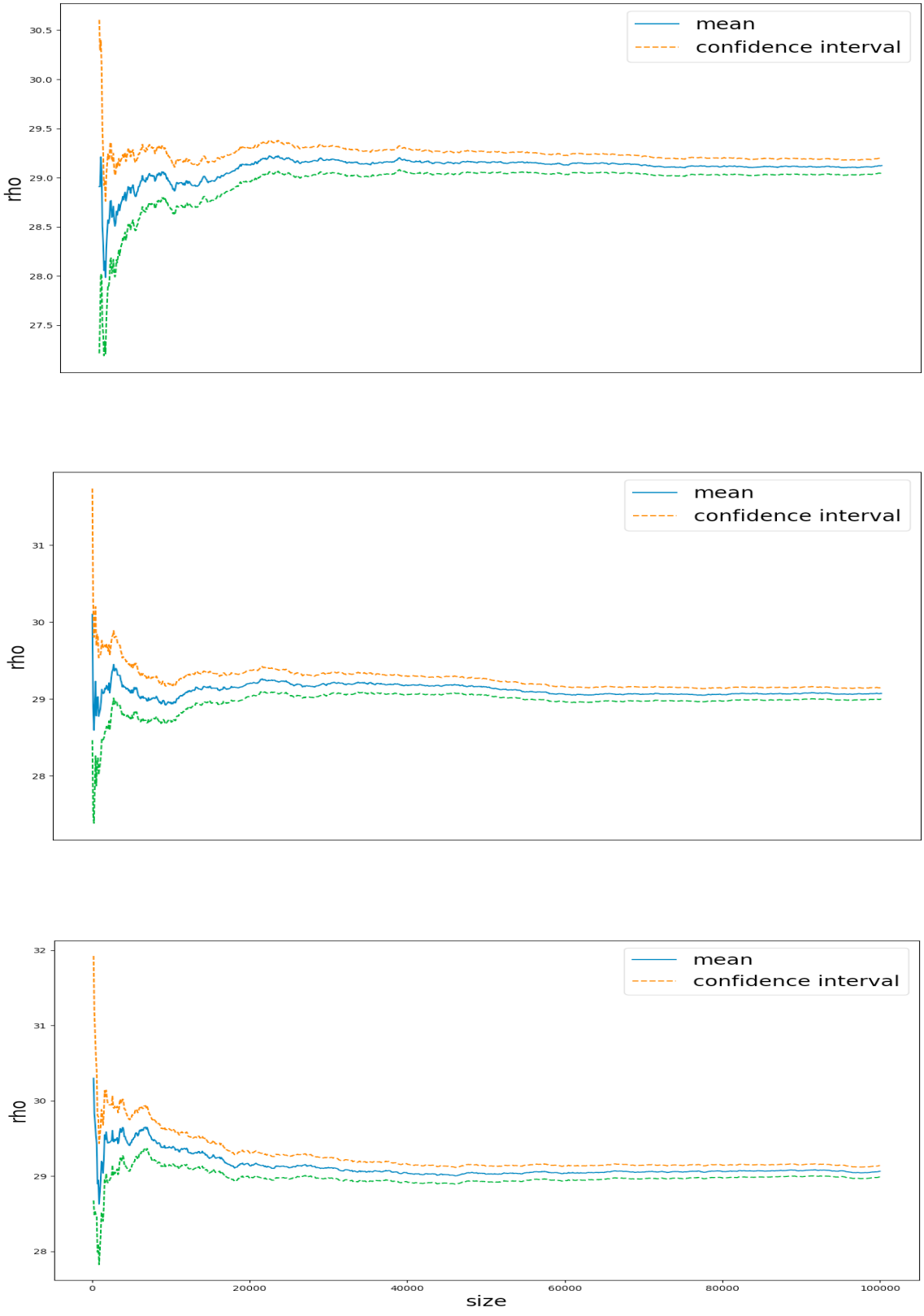


Figure 0.4: **Basket option rho using (top to bottom) finite difference method, tangent mode and adjoint mode**
Number of assets: 16, *Size*: 10^5 samples, *maturity*: 3 mth, *strike*: 120, *volatilities*: 40%, *initial spots*: 140, *risk free rate*: -1%, *weights*: equally distributed, *Correlation*: 85%

The most remarquable graph of this section is the following. It shows clearly the power of adjoint mode versus tangent mode versus finite difference method.
Indeed, one can notice that several order of magnitude in the processing time are gained.

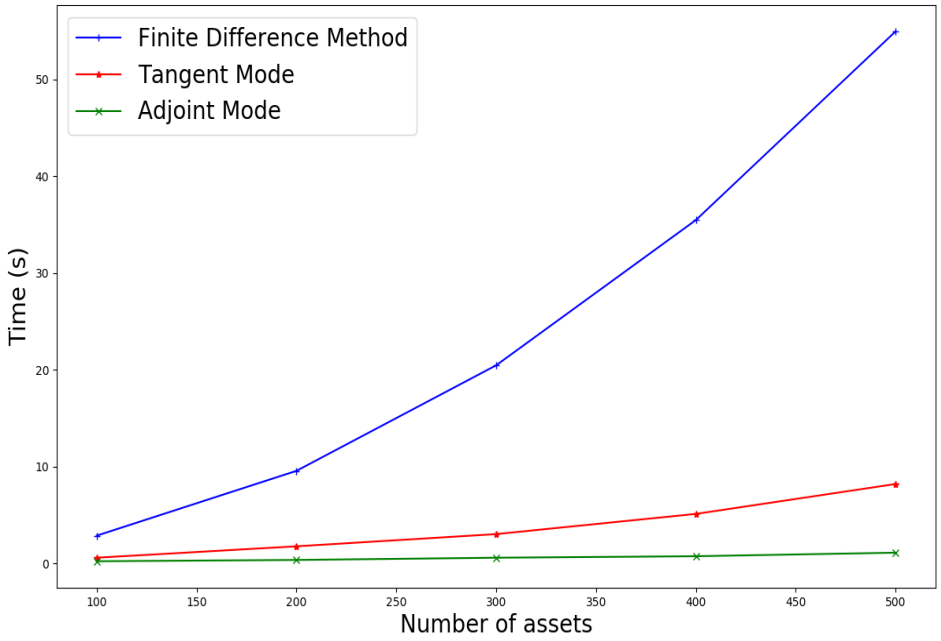


Figure 0.5: **Processing time comparison to compute basket option price, deltas, vegas and rho**
Number of assets: from 100 to 500 assets, *Size*: 10^3 samples, *maturity*: 1 yr, *strike*: 100, *volatilities*: 30%, *initial spots*: 100, *risk free rate*: -1%, *weights*: equally distributed, *Correlation*: 50%

Number of assets	Finite difference method	Tangent Mode of AD	Adjoint Mode of AD
100 assets	2.85378s	0.574918s	0.210927s
200 assets	9.55165s	1.75251s	0.344089s
300 assets	20.4614s	3.01471s	0.573359s
400 assets	35.4771s	5.10348s	0.728007s
500 assets	54.9782s	8.19274s	1.09516s

6.2 Best Of Asian Option

One now consider the case of a path-dependent option such as best of asian option. It consists for its holder to receive the mean of the maxima of the asset returns on fixing dates $0 = T_0 < T_1 < ... < T_N = T$ at price K, practically, the payout function writes:

$$P(r, (x_{i,j})_{1\leq i\leq d; 1\leq j\leq N}) = e^{-rT}\left(\frac{1}{N}\sum_{j=1}^{N}\max_{i=1...d}\left(\frac{S^i_{T_j}}{S^i_0}\right) - K\right)_+$$

The advantage to work under Black-Scholes framework is that one can assume without loss of generality that $S^i_0 = 1$ for all i, indeed this option is written on the returns of each asset and does not depend on the current value of the assets.

One adds to the Black Scholes tools class a *multiBlackScholes* class which produce a sample of the asset spot at the fixing dates. As one has built our own automatic differentiation library, there is no use to implement by hand the tangent and the adjoint mode of the payout as it is done in [1]. Instead, one needs to carry out the overloading of the vector/matrix input function max and mean in each class.

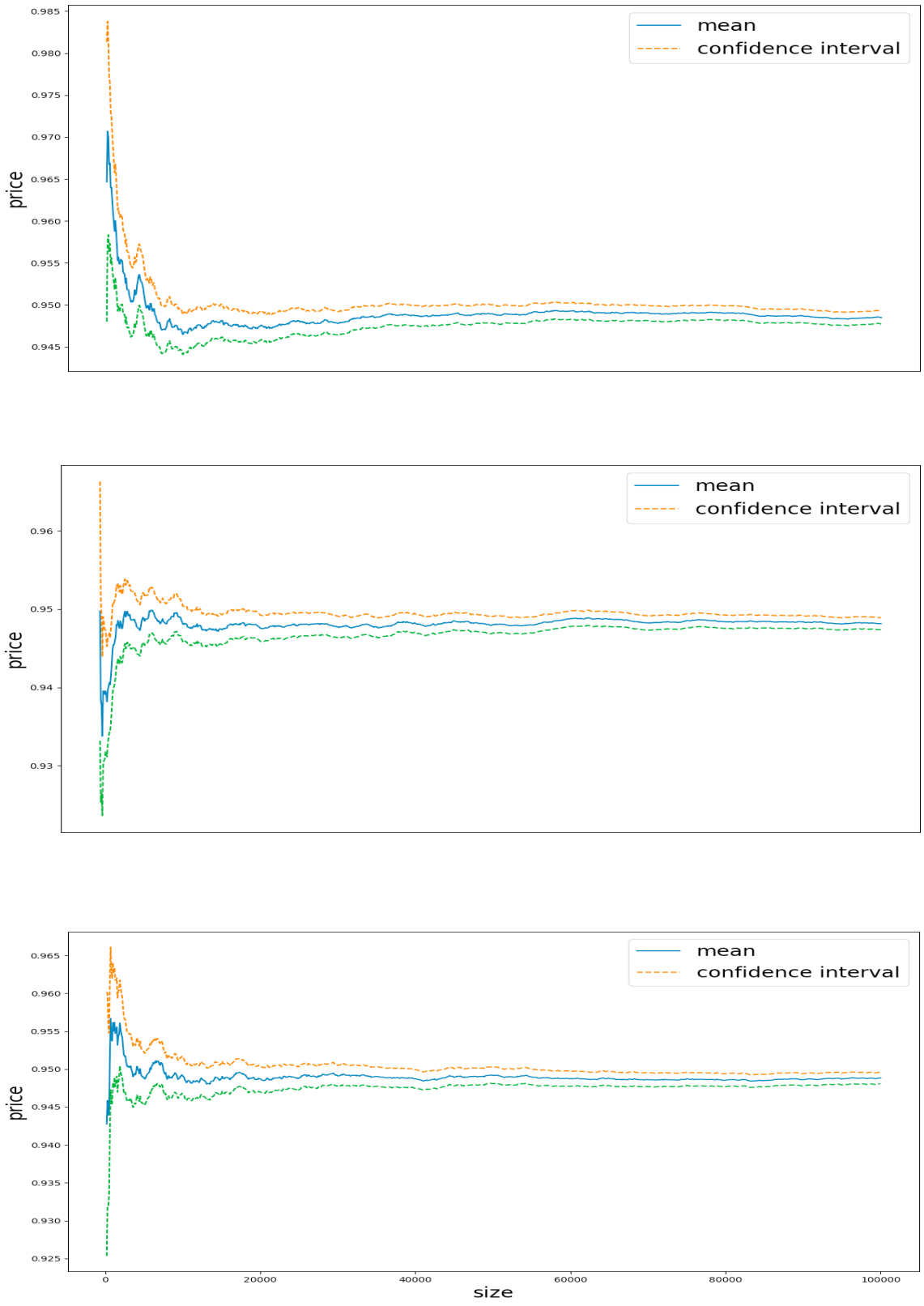


Figure 0.6: **Best Of Asian Option price using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 2, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50%

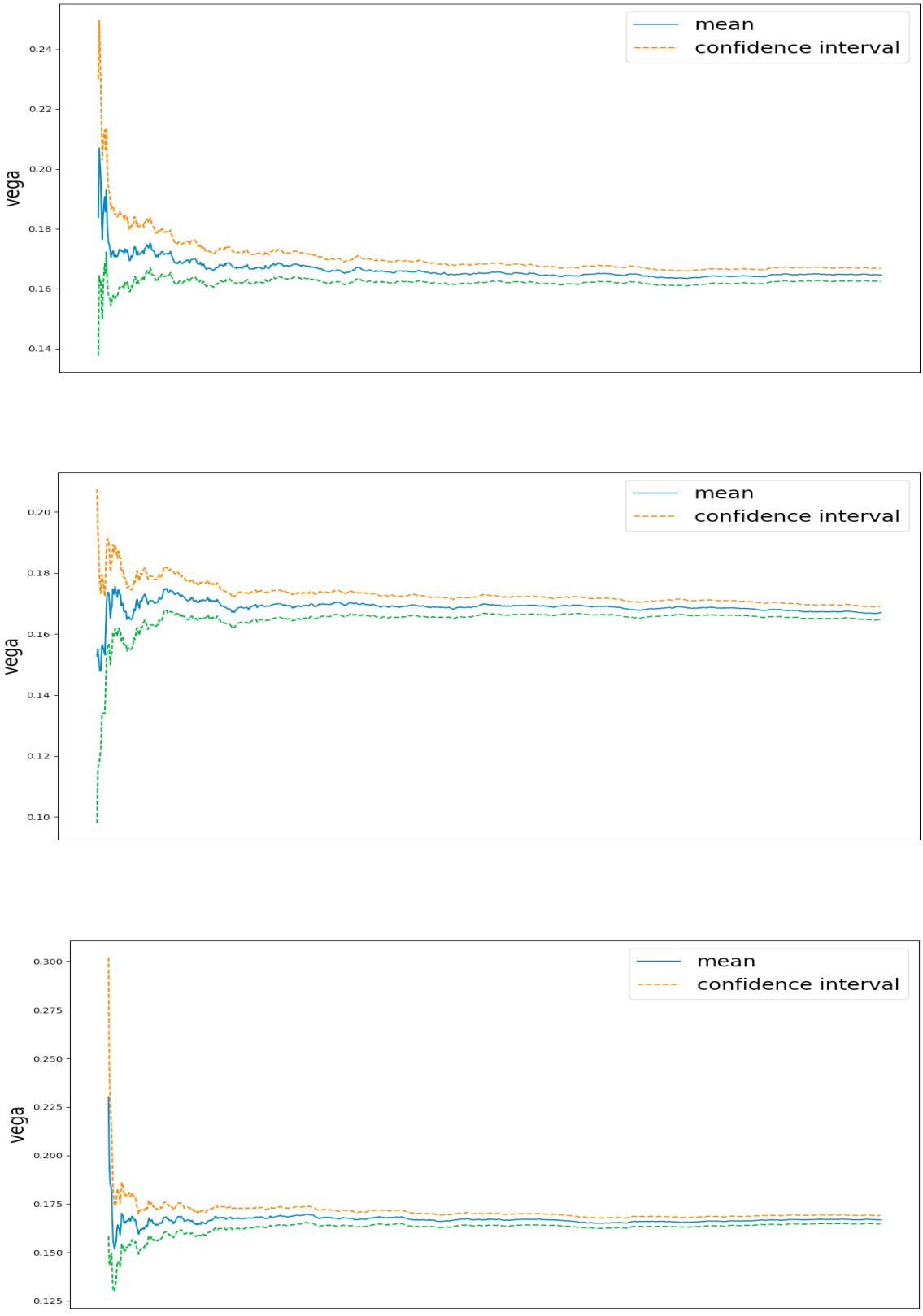
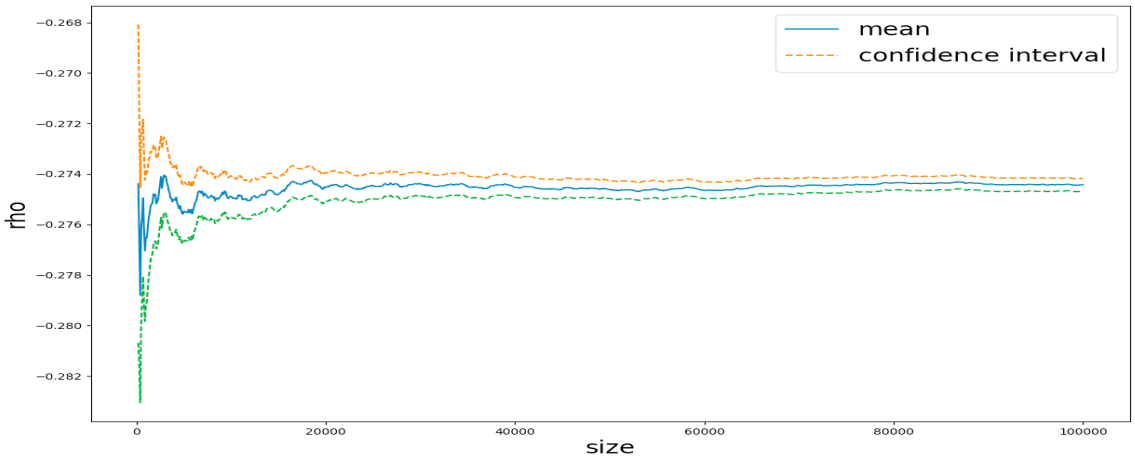
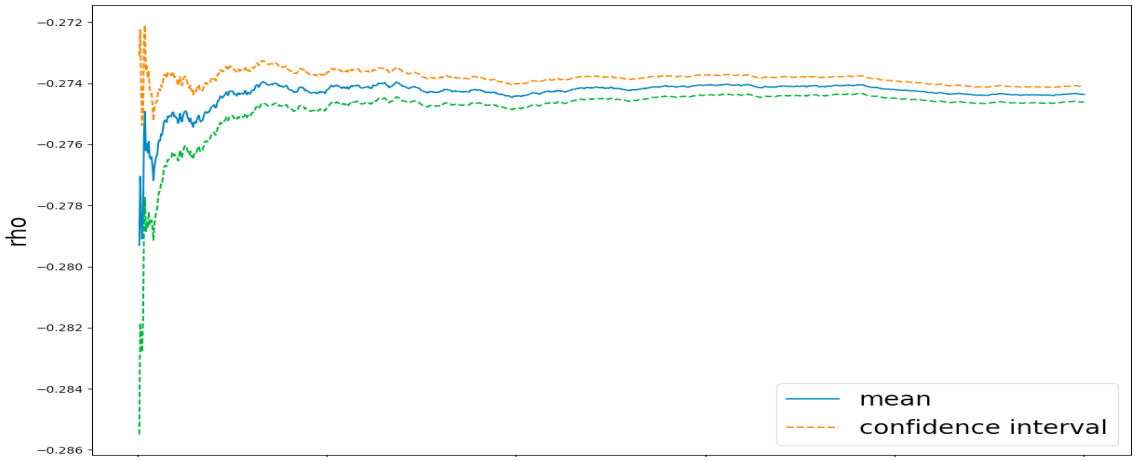


Figure 0.7: **Best Of Asian option vega using (top to bottom) finite difference method, tangent mode and adjoint mode**
Number of assets: 2, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50%





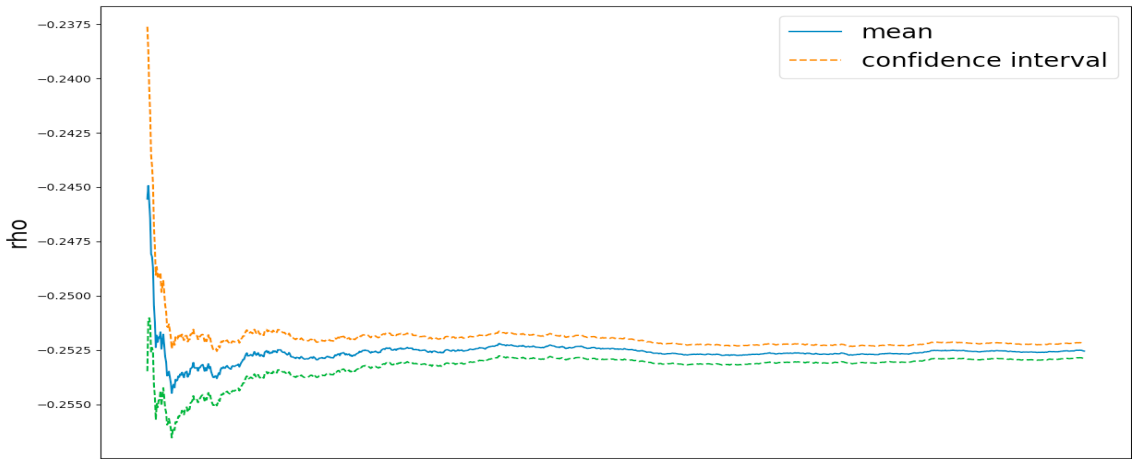


Figure 0.8: **Best Of Asian Option rho using (top to bottom) finite difference method, tangent mode and adjoint mode**
Number of assets: 2, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50%

For this kind of contingent claim, the complexity is increases linearly with $d \times N$ when one uses finite difference method. According to the resuts presented below, adjoint mode of AD holds highly accurate and speed compare to others and one notices that for large number of assets, it becomes the main tool to produce results quickly.

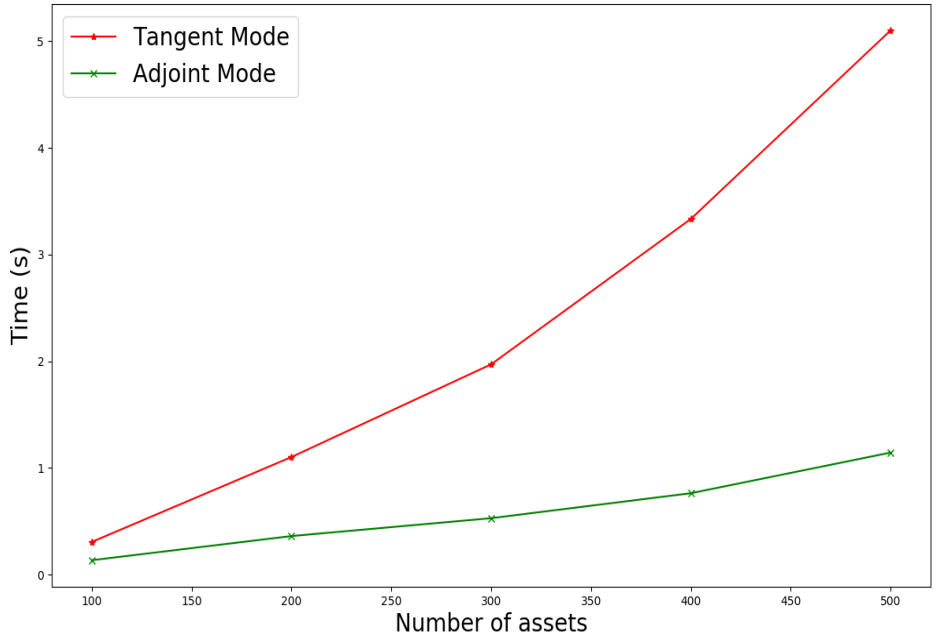


Figure 0.9: **Processing time comparison to compute best of asian option price, vegas and rho**
Number of assets: from 100 to 500, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50%

Market parameters	Finite difference method	Tangent Mode of AD	Adjoint Mode of AD
Price	1.04858	1.05031	1.0492
γ_1	0.1638	0.1673	0.1662
γ_2	0.1668	0.1691	0.1670
ρ	-0.252635	-0.274637	-0.274441

Number of assets	Finite difference method	Tangent Mode of AD	Adjoint Mode of AD
100 assets	28.0003s	0.306534s	0.135332s
200 assets	110.797s	1.10229s	0.361844s
300 assets	247.113s	1.97121s	0.529366s
400 assets	457.815s	3.33394s	0.763426s
500 assets	702.954s	5.10094s	1.14418s

7 Correlation greeks

One decides to make the correlation sensitivities computation in another framework. Here, one seeks to use automatic differentiation tools to compute the correlation part of the tangent state vector. It involves the calculation of the adjoint of cholesky decomposition. In [2], the author implement it by hand but one can proceed in an automatic way as one built the structure to do so.

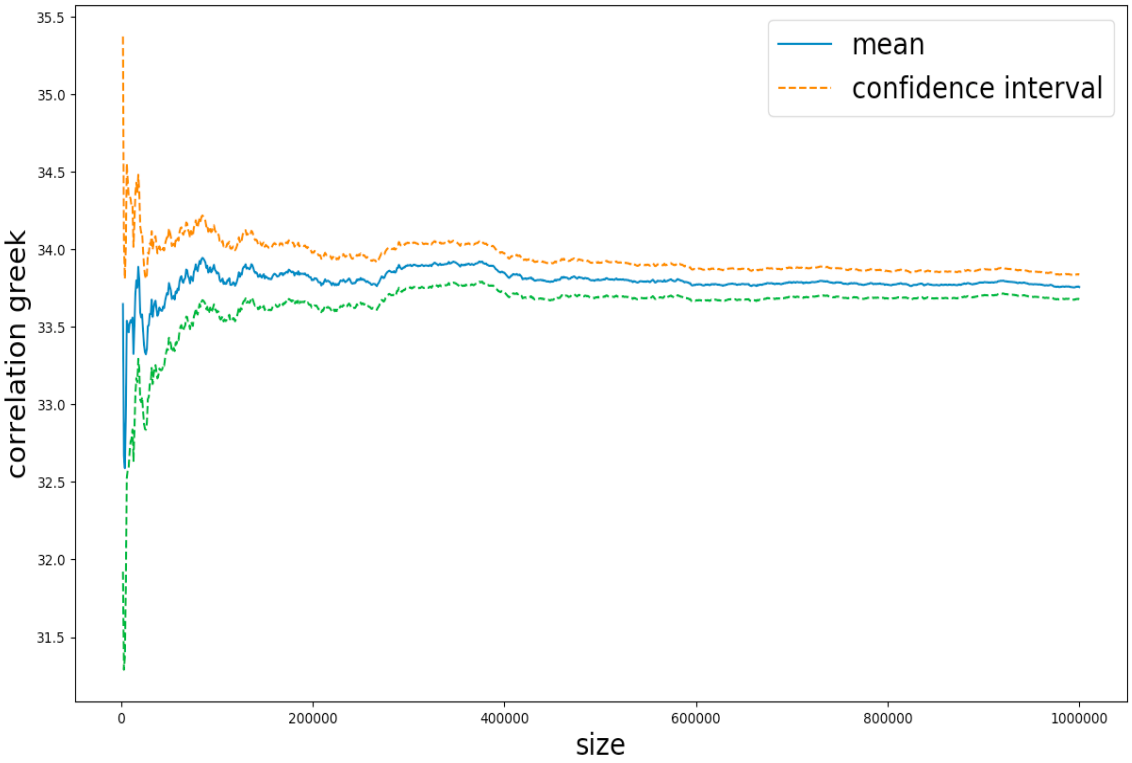


Figure 0.10: **Basket Option correlation sensitivity**
Number of assets: 2, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 100, *volatilities*: 30%, *initial spots*: 100, *risk free rate*: 10%, *weights*: equally distributed, *Correlation*: 25%

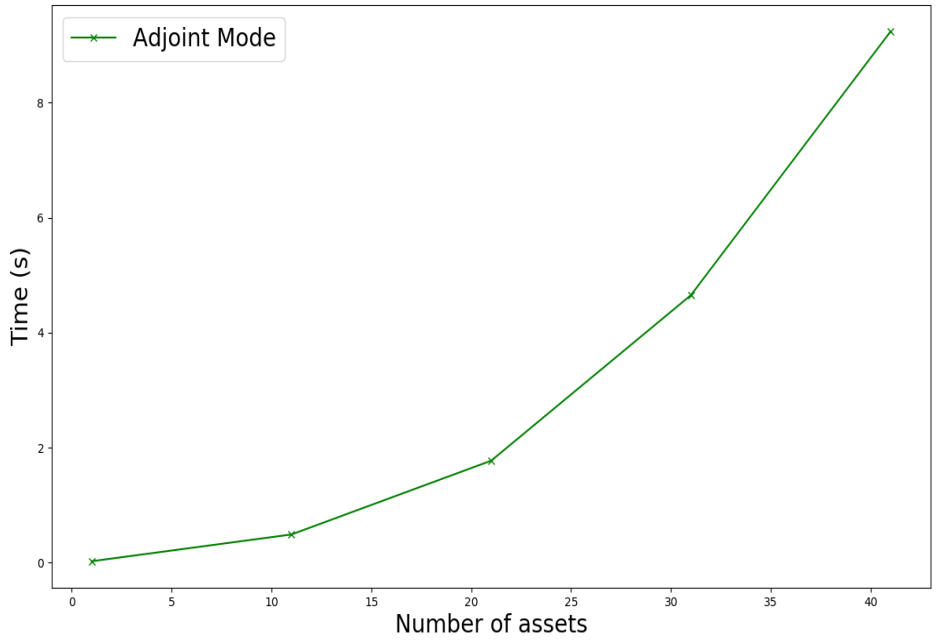


Figure 0.11: **Processing time comparison to compute basket option correlation greeks**
Number of assets: from 100 to 500, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50%

8 Conclusion

In this short report, I was seeking to explain as clearly as possible how algorithmic differentiation works and shows throughout simple examples of correlation products its efficiency and in particular the power of adjoint mode, which provides several orders of magnitude and speeds significantly the greeks calculation. I hope I have conviced the reader to use this technology in its future implementation works. As further works, one can apply it to the computation of the tangent state vector under more complex model with for example, given a diffusion scheme, seeking to diffuse the derivatives of the state vector with respect to the parameters involved.

List of Figures

0.1 **Basket option price using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 2, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 100, *volatilities*: 30%, *initial spots*: 100, *risk free rate*: 10%, *weights*: equally distributed, *Correlation*: 25% 11
0.2 **Basket option delta using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 4, *Size*: 10^5 samples, *maturity*: 2 yr, *strike*: 95, *volatilities*: 10%, *initial spots*: 120, *risk free rate*: 20%, *weights*: equally distributed, *Correlation*: 25% 12
0.3 **Basket option vega using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 8, *Size*: 10^5 samples, *maturity*: 6 mth, *strike*: 110, *volatilities*: 10%, *initial spots*: 100, *risk free rate*: 1%, *weights*: equally distributed, *Correlation*: 75% 13
0.4 **Basket option rho using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 16, *Size*: 10^5 samples, *maturity*: 3 mth, *strike*: 120, *volatilities*: 40%, *initial spots*: 140, *risk free rate*: -1%, *weights*: equally distributed, *Correlation*: 85% 14
0.5 **Processing time comparison to compute basket option price, deltas, vegas and rho** *Number of assets*: from 100 to 500 assets, *Size*: 10^3 samples, *maturity*: 1 yr, *strike*: 100, *volatilities*: 30%, *initial spots*: 100, *risk free rate*: -1%, *weights*: equally distributed, *Correlation*: 50% 15
0.6 **Best Of Asian Option price using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 2, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50% 16
0.7 **Best Of Asian option vega using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 2, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50% 17
0.8 **Best Of Asian Option rho using (top to bottom) finite difference method, tangent mode and adjoint mode** *Number of assets*: 2, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50% 18
0.9 **Processing time comparison to compute best of asian option price, vegas and rho** *Number of assets*: from 100 to 500, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50% 19
0.10 **Basket Option correlation sensitivity** *Number of assets*: 2, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 100, *volatilities*: 30%, *initial spots*: 100, *risk free rate*: 10%, *weights*: equally distributed, *Correlation*: 25% 20
0.11 **Processing time comparison to compute basket option correlation greeks** *Number of assets*: from 100 to 500, *Fixing*: monthly, *Size*: 10^5 samples, *maturity*: 1 yr, *strike*: 1%, *volatilities*: 30%, *risk free rate*: 10%, *Correlation*: 50% 20

References

[1] Luca Capriotti. Fast greeks by algorithmic differentiation. *The Journal of Computational Finance*, Volume 14, 2011.

[2] Luca Capriotti and Mike Giles. Fast correlation greeks by adjoint algorithmic differentiation. 2018.