Computational Differentiation : Finance

Simulation Sciences Seminar

Kumari Shikha

04 July 2018

Supervisor : Dr. Johannes Lotz

Software and Tools for Computational Engineering | RWTH AACHEN UNIVERSITY

Main reference paper

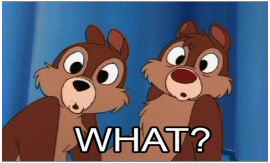
U. Naumann and J. du Toit.
Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance.
Journal of Computational Finance, 2016

Flow of the Presentation

- Use Case
  - Basic Terminologies
- Mathematical formulation of the problem
- Introduce dco/c++
- Checkpointing
- Performance Comparison

Use Case

Consider a simple European call option on an underlying driven by a local volatility process



Consider a simple European call option on an **underlying** driven by a local volatility process

Underlying

An underlying security is a stock, index, bond, interest rate, currency or commodity on which derivative instruments, such as futures and options, are based.

Consider a simple European **call option** on an underlying driven by a local volatility process

Call Option

Right to buy / sell

An **options contract** is an agreement between two parties to facilitate a potential transaction on the underlying security at a preset price, referred to as the strike price, prior to the expiration date **(maturity)**. The two types of contracts are **put** and **call** options.

Consider a simple **European call option** on an underlying driven by a local volatility process

Types of Option

Mainly two types of option : American and European
Name has nothing to do with geographic location.

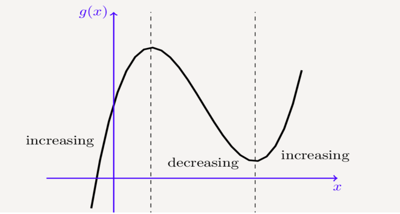
An **American option** is an option that can be exercised anytime during its life.
A **European option** is an option that can only be exercised at the end of its life, at its maturity.

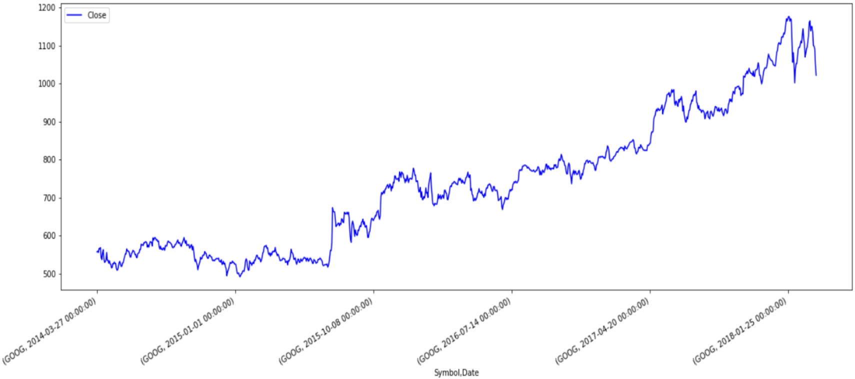
Consider a simple European call option on an underlying driven by a **local volatility process**

Volatility

Volatility refers to the **amount of uncertainty or risk** about the size of changes in a security's value.
Volatility can either be measured by using the **standard deviation or variance** between returns from that same security or market index.

Commonly, **the higher the volatility, the riskier the security** .





Google stock price close values : 1/4/2013 - 25/3/2018

Consider a simple European call option on an underlying driven by a local volatility process

Monte Carlo Pricing

Let $S = (S_t)_{t \geq 0}$ be the solution to the SDE :

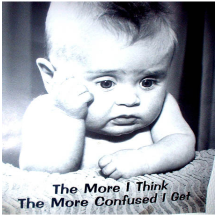
$dS_t = rS_t dt + \sigma(log(S_t), t)S_t dW_t$

where $W = (W_t)_{t \geq 0}$ is a standard Brownian motion
r>0 is the risk free interest rate
σ is the local volatility function

The **price of the call option** is then given by :
$V = e^{-rT} E(S_T - K)^+$
T : Final time / maturity / expiration time

sigma σ

In practice σ will typically be computed from the market observed implied volatility surface and is often represented either as a bicubic spline or as a series of one-dimensional splines.



sigma σ

To keep things simple , we choose to represent σ as
$\sigma(x, t) = g(x).t$

$g : R- > R_+$ is given by :

$g(x) = \frac{p_m(x)}{q_n(x)} = \frac{a_0+a_1x+...+a_mx^m}{b_0+b_1x+...+b_nx^n}$

p_n and q_m are polynomials of order n and m respectively .

Sensitivities

Basically relevant derivatives

	Spot price (S)	**Volatility (σ)**	**Time to expiry (τ)**
Value (V)	Δ Delta	$\mathcal{V}$ Vega	Θ Theta
Delta (Δ)	Γ Gamma	Vanna	Charm
Vega ($\mathcal{V}$)	Vanna	Vomma	Veta
Theta (Θ)	Charm	Veta	
Gamma (Γ)	Speed	Zomma	Color
Vomma		Ultima	

Active and Passive variables

Active outputs : We are interested in their rate of change
Active inputs : We are interested in rate of change wrt them

Passive outputs : We are NOT interested in their rate of change
Passive inputs : We are NOT interested in rate of change wrt them

eg : Active input : S0
Passive input : a_i in calculation of sigma

How to differentiate

$z = f(a, b, c)$

Tangent mode / Forward mode : Derivative of output of every "layer" wrt input of that layer is calculated / stored . $\frac{\partial f}{\partial a}$, $\frac{\partial f}{\partial b}$, $\frac{\partial f}{\partial c}$

Reverse / Adjoint mode : Derivative of input of every "layer" wrt output of that layer is calculated / stored . $\frac{\partial a}{\partial f}$, $\frac{\partial b}{\partial f}$, $\frac{\partial c}{\partial f}$

How to differentiate : An example

Expression : $\mathbf{z = x * y + sin(x)}$

We are interested in the derivatives of output wrt input , i.e. $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$

Steps to calculate z :

x = input x

y = input y

a = x * y

b = sin(x)

z = a + b

How to differentiate : Tangent / Forward mode

Chain rule : $\frac{\partial z}{\partial t} = \Sigma\left(\frac{\partial z}{\partial u_i}.\frac{\partial u_i}{\partial t}\right)$ where t is some input variable like x .

Our example : $\mathbf{z = x * y + sin(x)}$

$\frac{\partial x}{\partial t} = ?$

$\frac{\partial y}{\partial t} = ?$

$\frac{\partial a}{\partial t} = y * \frac{\partial x}{\partial t} + x * \frac{\partial y}{\partial t}$

$\frac{\partial b}{\partial t} = cos(x) * \frac{\partial x}{\partial t}$

$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}$

Now , if we want $\frac{\partial z}{\partial x}$, $t = x$ and therefore $\frac{\partial x}{\partial x} = 1$ and $\frac{\partial y}{\partial x} = 0$.

Once we "seed" these value , everything else is taken care of .

How to differentiate : Tangent / Forward mode

Chain rule : $\frac{\partial z}{\partial t} = \Sigma\left(\frac{\partial z}{\partial u_i}.\frac{\partial u_i}{\partial t}\right)$ where t is some input variable like x .

Our example : **z = x * y + sin(x)**

$\frac{\partial x}{\partial t} = 1$

$\frac{\partial y}{\partial t} = 0$

$\frac{\partial a}{\partial t} = y * 1 + x * 0$

$\frac{\partial b}{\partial t} = cos(x) * 1$

$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}$

$\frac{\partial z}{\partial t} = y + cos(x)$

$\frac{\partial z}{\partial x} = y + cos(x)$

How to differentiate : Tangent / Forward mode

Advantage : The differential variables depend on the intermediate variables, calculate them together , no need to hold on to the the intermediate variables until later, **saving memory**.

Expression : z = x * y + sin(x)

x = input value

$\frac{\partial x}{\partial t} = ?$

y = input value

$\frac{\partial y}{\partial t} = ?$

a = x * y

$\frac{\partial a}{\partial t} = y * \frac{\partial x}{\partial t} + x * \frac{\partial y}{\partial t}$

b = sin(x)

$\frac{\partial b}{\partial t} = cos(x) * \frac{\partial x}{\partial t}$

z = a + b

$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}$

How to differentiate : Tangent / Forward mode

Disadvantages : Since we set t = x to calculate $\frac{\partial z}{\partial x}$,
we will need to set t = y for $\frac{\partial z}{\partial y}$. This means number of passes = number of input variables .

Problem if number of input variables is large !

How to differentiate : Reverse / Adjoint mode

Turn the chain rule upside down : $\frac{\partial s}{\partial u} = \Sigma\left(\frac{\partial z_i}{\partial u} \cdot \frac{\partial s}{\partial z_i}\right)$

z_i : output variables of interest
u : input variables (x and y)
s : some output variable (either of the z_i)

For example problem : $\mathbf{z = x * y + \sin(x)}$

$\frac{\partial s}{\partial z} = ?$
$\frac{\partial s}{\partial b} = \frac{\partial s}{\partial z}$
$\frac{\partial s}{\partial a} = \frac{\partial s}{\partial z}$
$\frac{\partial s}{\partial y} = x * \frac{\partial s}{\partial a}$
$\frac{\partial s}{\partial x} = y * \frac{\partial s}{\partial a} + cos(x) * \frac{\partial s}{\partial b}$

Put s = z and in just 1 pass get $\frac{\partial s}{\partial x}$ and $\frac{\partial s}{\partial y}$

How to differentiate : Reverse / Adjoint mode

s = z ; $\frac{\partial s}{\partial z} = 1$

For example problem : **z = x * y + sin(x)**

$\frac{\partial s}{\partial z} = 1$

$\frac{\partial s}{\partial b} = 1$

$\frac{\partial s}{\partial a} = 1$

$\frac{\partial s}{\partial y} = x * 1$

$\frac{\partial s}{\partial x} = y * 1 + cos(x) * 1$

$\frac{\partial z}{\partial y} = x$

$\frac{\partial z}{\partial x} = y + cos(x)$

How to differentiate : Reverse / Adjoint mode

Disadvantage : Now calculations and differential calculation cannot be interleaved . We need to save the intermediate variables also , and then calculate the differentials , thus leading to more memory use .

Disadvantage : If number of active output variables is large and number of active input variables is small , say 1 , tangent mode would be better since it requires only 1 pass .

Open source Automatic differentiation tools :

`www.autodiff.org`

`http://www.autodiff.org/?module=Tools`

Click on language preference -> List of tools

dco/c++

An Algorithmic Differentiation tool developed by Numerical Algorithms Group
Link for download :

https://www.nag.co.uk/content/downloads-dco-c-versions

dco/c++: Tangent Mode

```
#include "dco.hpp"
typedef dco::gt1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;

   ACTIVE_INPUTS<DCO_TYPE> X;
   PASSIVE_INPUTS XP;
   ACTIVE_OUTPUTS<DCO_TYPE> Y;
   PASSIVE_OUTPUTS YP;

   dco::derivative(X.S0)=1;
   price(X,XP,Y,YP);
   cout << "Y=" << dco::value(Y.V) << endl;
   cout << "dY/dX.S0=" << dco::derivative(Y.V) << endl;

   dco::derivative(X.S0)=0;
   dco::derivative(X.r)=1;
   price(X,XP,Y,YP);
   cout << "dY/dX.r=" << dco::derivative(Y.V) << endl;
```

[1]

Notice : price function called twice

dco/c++: Adjoint Mode

```
#include "dco.hpp"
typedef dco::ga1s<double> DCO_MODE;
typedef DCO_MODE::type DCO_TYPE;
typedef DCO_MODE::tape_t DCO_TAPE_TYPE;
DCO_TAPE_TYPE* & DCO_TAPE_POINTER=DCO_MODE::global_tape;

  ACTIVE_INPUTS<DCO_TYPE> X;
  PASSIVE_INPUTS XP;
  ACTIVE_OUTPUTS<DCO_TYPE> Y;
  PASSIVE_OUTPUTS YP;

  DCO_TAPE_POINTER = DCO_TAPE_TYPE::create();
  DCO_TAPE_POINTER->register_variable(X.S0);
  DCO_TAPE_POINTER->register_variable(X.r);
  ...

  price(X,XP,Y,YP);

  DCO_TAPE_POINTER->register_output_variable(Y.V);
  dco::derivative(Y.V)=1;
  DCO_TAPE_POINTER->interpret_adjoint();

  cout << "Y=" << dco::value(Y.V) << endl;
  cout << "dY/dX.S0=" << dco::derivative(X.S0) << endl;
  cout << "dY/dX.r=" << dco::derivative(X.r) << endl;
  ...

  DCO_TAPE_TYPE::remove(DCO_TAPE_POINTER);
```

[1]

Notice : price function called once

Memory

input................intermediate variables / values...........output

Problem : Since the memory requirements of the tape scale more or less linearly with the number of sample paths, this leads to infeasible peak memory requirements.
Solution : Store some values "elsewhere" , use them for calculations later.

Checkpointing

A checkpoint is a set of data which is stored (either to disk or to memory) at a point during a computation, and which allows the computation to be restarted (at some later time) from that point.

Advantage : Peak memory requirement reduces (A lot !) .

Disadvantage : Amount of computation increases (Not a lot).

$$F : (\mathbf{x}, \tilde{\mathbf{x}}) \xrightarrow{f_1} (\mathbf{u}, \tilde{\mathbf{u}}) \, , \begin{pmatrix} (\mathbf{u}, \tilde{\mathbf{u}}_1) \xrightarrow{g} (\mathbf{v}_1, \tilde{\mathbf{v}}_1) \\ (\mathbf{u}, \tilde{\mathbf{u}}_2) \xrightarrow{g} (\mathbf{v}_2, \tilde{\mathbf{v}}_2) \\ \vdots \\ (\mathbf{u}, \tilde{\mathbf{u}}_N) \xrightarrow{g} (\mathbf{v}_N, \tilde{\mathbf{v}}_N) \end{pmatrix} , (\mathbf{v}_1, \ldots, \mathbf{v}_N, \tilde{\mathbf{v}}_1, \ldots, \tilde{\mathbf{v}}_N) \xrightarrow{f_2} (y, \tilde{\mathbf{y}})$$

[1]

N : sample paths
N evaluations of g : explosion in tape size
Mutual independence of evaluation of g : work in parallel

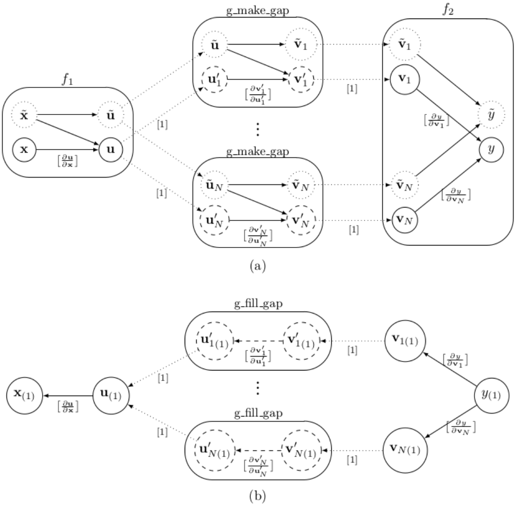
$X_t = log(S_t)$
$X_{t0} = log(S_0)$
Z_i : standard normal random number
$\Delta = T/M$ some integer M
$t_i = i\Delta, i = 1, 2, ...M =$ Monte Carlo time steps

$$dX_t = \left(r - \tfrac{1}{2}\sigma^2(X_t, t)\right)dt + \sigma(X_t, t)\, dW_t.$$

$$X_{t_{i+1}} = X_{t_i} + \left(r - \tfrac{1}{2}\sigma^2(X_{t_i}, t_i)\right)\Delta + \sigma(X_{t_i}, t_i)\sqrt{\Delta}Z_i$$

N sample paths generated to use in MC integrator to calculate V

Pathwise adjoint calculation :



[1]

Performance

n	`mc/primal`	`mc/cfd`	`mc/a1s`	`mc/a1s_ensemble`	$\mathcal{R}$
10	0.3s	6.1s	1.8s (2GB)	1.3s (1.9MB)	4.3
22	0.4s	15.7s	- ($>$ 3GB)	2.3s (2.2MB)	5.7
34	0.5s	29.0s	- ($>$ 3GB)	3.0s (2.5MB)	6.0
62	0.7s	80.9s	- ($>$ 3GB)	5.1s (3.2MB)	7.3
142	1.5s	423.5s	- ($>$ 3GB)	12.4s (5.1MB)	8.3
222	2.3s	1010.7s	- ($>$ 3GB)	24.4s (7.1MB)	10.6

[1]

N = 10000

R : Runtime of AD code / Runtime of primal code

R is sensitive to compiler flags , memory hierarchy , cache sizes , level of optimization of primal code

References

[1] U. Naumann and J. du Toit.
Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance.
Journal of Computational Finance, 2016

[2]https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation
[3]https://www.probabilitycourse.com/chapter4/4_1_3_functions_continuous_var.php
[4]http://www.picturequotes.com/the-more-i-think-the-more-confused-i-get-quote-20923
[5]https://www.wikiwand.com/en/Greeks_(finance)

Questions ?