Monte Carlo Option Greeks by Hand Made Adjoint of Automatic Differentiation

March 1, 2016

1 Monte Carlo Option Greeks by Hand Made Adjoint of Automatic Differentiation

David Dallaire - linkedin https://ca.linkedin.com/in/ddallaire

2 Introduction

This IPython/Jupyter notebook shows a simple example of calculating option greeks in Monte Carlo by the adjoint method of automatic differentiation. The focus is on implementation in Python: background can be found in available books and papers. The goal is to demonstrate the mechanics of the adjoint method via a simple implementation–a supplement to a text description of the algorithm.

The adjoint or backward mode method of calculating option greeks is an alternative to the finite difference methodology (knows as "bumping") and to closed form symbolic greeks. Speed of calculation is the main advantage of the adjoint over bumping, especially when the number of sensitivities is large. The speed-up will not be as noticeable in this implementation as the example is on a simple option.

This notebook has five main sections. The first section has the math for the algorithm. This is followed by implementation of the closed form Merton model. The third section will dive into the hand implementation of Delta, Vega and Rho greeks. The fourth section repeats the calculation of Delta, Vega and Rho but using finite differences. The fifth section will show an example of the python package "ad" for automatic differentiation by mechanical method.

3 Adjoint Math for Geometric Brownian Motion SDE

The Euler discretization is a follows:

$$S_{n+1} = S_n + rS_n\Delta_t + \sigma S_n\sqrt{\Delta_t}Z_n \; n = 0, ...N-1 \tag{1}$$

Given known values for S_0, r, σ, T, K.

The payoff is as follows:

$$P = e^{-rT}max(0, S_N - K) \tag{2}$$

The adjoint applies the chain rule $\frac{\partial V}{\partial \theta} = \frac{\partial P}{\partial S}\frac{\partial S}{\partial \theta}$ and has two sets of parameters: a set for the payoff $\frac{\partial P}{\partial S}$ and a set for the MC path $\frac{\partial S}{\partial \theta}$.

For each MC path, the adjoint has to do a forward pass: it steps through time going forward to the payoff. Some of the path parameters needed on the backward pass can be stored during this forward pass. For example, each S_n and random numbers Z_n of the path are stored.

Once the time step reaches the final simulation time N (maturity), the payoff parameters are activated if the option is in the money. The payoff parameters all have a bar above their letter. Each parameter is found by differentiating the payoff in turn. Here are the parameters and their values:

$$\bar{S}_N = \frac{\partial P}{\partial S} = e^{-rT} 1_{S_N > K} \tag{3}$$

$$\bar{r} = \frac{\partial P}{\partial r} = -e^{-rT} 1_{S_N > K} (S_N - K) T \tag{4}$$

$$\bar{\sigma} = \frac{\partial P}{\partial \sigma} = 0 \tag{5}$$

$$\bar{T} = \frac{\partial P}{\partial T} = -e^{-rT} 1_{S_N > K} (S_N - K) r \tag{6}$$

For positive payoffs only, the bar values are simulated backwards from time N-1 to zero. The MC path parameters are as follows:

$$\bar{S}_n = (1 + r\Delta_t + \sigma\sqrt{\Delta_t} Z_n) \bar{S}_{n+1} \tag{7}$$

$$\bar{r} = \sum_{n=N-1}^{0} (S_n \Delta_t \bar{S}_{n+1}) \tag{8}$$

$$\bar{\sigma} = \sum_{n=N-1}^{0} (S_n \sqrt{\Delta_t} \bar{S}_{n+1}) \tag{9}$$

4 Closed Form Generalized Black-Sholes/Merton Model

In [42]:
```
#Closed Form Generalized Black-Sholes
import math
import scipy.stats
stockPrice = 100
strike = 100
shortRate = 0.06
dividend = 0.03
Maturity = 1
volatility = 0.2

d1 = (math.log(stockPrice/strike) + (shortRate - dividend + volatility*volatility/2) \
      *Maturity)/(math.sqrt(Maturity) * volatility)

d2 = d1 - volatility * math.sqrt(Maturity)

delta = math.exp(-dividend*Maturity)*scipy.stats.norm(0,1).cdf(d1)

vega = stockPrice *  math.exp(-dividend*Maturity)*scipy.stats.norm(0,1).pdf(d1) * math.sqrt(Ma

rho = strike * math.exp(-shortRate * Maturity)*scipy.stats.norm(0,1).cdf(d2)

callValue = stockPrice * delta - rho

print("callValue " + str(callValue))
print("Delta = " + str(delta))
print("Vega = " + str(vega))
print("Rho = " + str(rho))
```

```
callValue 9.13519526935
Delta = 0.581011879666
Vega = 37.5240346917
Rho = 48.9659926973
```

5 Hand Implementation of Adjoint for Monte Carlo Option Greeks

Coded here are two versions: one is based on the 'for loop' structure and the other is based on the vector/matrix calculations of the 'NumPy' python package. The vector based calculation is much faster.

The reason for creating the loop based version is to compare it to the mechanical automatic differentiation method of the 'ad' python library, given the 'ad' package does not work with NumPy vector/matrix.

5.1 Loop Based Hand Adjoint

In [1]:
```
%time
'''
#########################################################################
#
#               Simple Monte Carlo looping with Adjoin Automatic
#               Differentiation by Hand
#               Version 1.0.0
#               David Dallaire, February 11, 2016
#
#               Regular inputs for vanilla option:
#                     Price, Strike, Rate, Dividend Yield, Volatility,
#                     Maturity
#########################################################################
#
#               OUTPUTS
#               delta, rho, vega
#########################################################################

'''
import numpy as np
import sys
import math as math
from time import time

t0 = time()
np.random.seed(3000)
#params
maturity=1; initialStockPrice=100; strike=100; sigma=0.2; rate=.06; dividend = 0.03
timeSteps = 365;simulations=250000

dt = maturity/timeSteps
sqrtDt = math.sqrt(dt)
nudt =(rate-dividend-0.5*sigma**2)*dt #drift
sigmasdt = sigma* math.sqrt(dt) #dW
lnSo = math.log(initialStockPrice)
sumCallValue = 0
sumDelta = 0
sumR = 0
```

```
sumSig = 0
D = math.exp(-rate*maturity)

Stockn = np.zeros(timeSteps + 1)
Stockn[0] = initialStockPrice

for j in range(1,simulations+1) :
    lnSt = lnSo
    #store random values on forward pass
    randVector =  np.random.randn(timeSteps)
    for i in range(1,timeSteps+1):
        lnSt = lnSt + nudt + sigmasdt * float(randVector[i-1])
        #store stock values on forward pass
        Stockn[i] = math.exp(lnSt)

    endStockValue = Stockn[timeSteps]
    callValue = max(0, endStockValue - strike)

    #start adjoint backward pass
    if endStockValue > strike:
        #initialize bar values at N
        SBarN = D  #Equation 3
        rhoBarN = -D * (endStockValue-strike)*maturity #Equation 4
        sigmaBarN = 0 #Equation 5
        #loop backwards
        for n in range(timeSteps-1,-1,-1):
            rhoBarN = rhoBarN + Stockn[n] * dt * SBarN #Equation 8
            sigmaBarN = sigmaBarN + Stockn[n] * sqrtDt * randVector[n] * SBarN #Equation 9
            SBarN = (1 + nudt + sigma * sqrtDt * randVector[n]) * SBarN #Equation 7
        sumDelta += SBarN
        sumR += rhoBarN
        sumSig += sigmaBarN

    sumCallValue += callValue

call_value = sumCallValue/simulations * D
print("----------------------")
print("Call Value = " + str(call_value))
print("delta = " + str(sumDelta/simulations))
print("rho = " + str(sumR/simulations))
print("vega = " + str(sumSig/simulations))
tnp1 = time() - t0
print("Elapsed time = " + str(tnp1))

#250,000 paths and 365 time steps
#CPU times: user 0 ns, sys: 0 ns, total: 0 ns
#Wall time: 5.25 µs
#----------------------
#Call Value = 9.18927287808
#delta = 0.571032536288
#rho = 48.4648524133
#vega = 37.4512229808
#Elapsed time = 221.39049291610718
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 5.25 μs
----------------------
Call Value = 9.18927287808
delta = 0.571032536288
rho = 48.4648524133
vega = 37.4512229808
Elapsed time = 221.39049291610718
```

5.2 Vector Based Hand Adjoint

```
In [2]: %time
        '''
        ###########################################################################
        #
        #               Simple Monte Carlo by vector calculation
        #                  with Adjoin Automatic Differentiation by Hand
        #               Version 1.0.0
        #               David Dallaire, February 11, 2016
        #
        #               Regular inputs for vanilla option:
        #                     Price, Strike, Rate, Dividend Yield, Volatility,
        #                     Maturity
        ###########################################################################
        #
        #               OUTPUTS
        #               delta, rho, vega
        ###########################################################################
        Created on Jan 18, 2016
        http://pythonhosted.org/ad/
        @author: ddu
        '''
        import numpy as np
        import math as math
        import sys

        def MC(stockValue,strike,rate,dividend,maturity,sigma,paths,timeSteps):

            D = math.exp(-rate*maturity)
            lnSo = math.log(stockValue)
            np.random.seed(3000)
            dt = maturity/timeSteps
            nudt = (rate - dividend - 0.5*sigma**2) * dt #drift
            randMatrix =  np.random.randn(paths,timeSteps)

            increments = nudt + randMatrix * sigma * math.sqrt(dt)

            colS = np.zeros((paths,1))
            colS[:]=lnSo
            #setup matrix with stock price in first column and increments in all future time step colum
            kSim = np.hstack((colS,increments))
            #calculate cumulative sum on forward pass of simulated stock prices
            cummulant = np.exp(np.cumsum(kSim,axis=1))
            #keep columns where payoff is positive
```

```
    inTheMoneyBool = cummulant[:,timeSteps]>strike
    endStockValue = cummulant[inTheMoneyBool]
    callValue = np.sum(endStockValue[:,timeSteps]-strike)/paths  * math.exp(-rate*maturity)

    #this is same as:
    #diff = 1 + (rate - dividend - 0.5*sigma**2) * dt + randMatrix * sigma * math.sqrt(dt)
    diff = 1 + increments
    diff = diff[inTheMoneyBool] #keep in the money rows

    nRows = endStockValue.shape[0]
    tmpCol = np.full((nRows,1),D)
    #setup matrix where last column is SbarN and all other columns are the decrements
    delta = np.hstack((diff,tmpCol))

    #this does the cummulative in reverse
    deltaSim = np.fliplr(np.cumprod(np.fliplr(delta),axis=1)) # Equation 7
    deltaValue = np.sum(deltaSim[:,0])/paths

    #rho
    rV = -D * (endStockValue[:,timeSteps]-strike) * maturity   #this is rBarN
    rVdiffs = dt * (endStockValue[:,0:timeSteps] * deltaSim[:,1:timeSteps+1]) # these are the r
    #put in one matrix
    rVAll = np.hstack((dt * (endStockValue[:,0:timeSteps] * deltaSim[:,1:timeSteps+1]),
                                    rV.reshape((nRows,1))))

    rho = np.sum(rVAll[:,1:(timeSteps+3)])/paths

    #vega
    vega = (endStockValue[:,0:timeSteps] * deltaSim[:,1:timeSteps+1] * randMatrix[inTheMoneyBool
    vegaValue = (np.sum(vega)/paths)

    #theta
    #thetaN = -D * (endStockValue[:,timeSteps]-strike) * r

    Sn = endStockValue[:,0:timeSteps]
    SBarn = deltaSim[:,1:timeSteps+1]
    #T = (((r*Sn + (0.5*sigma*Sn*randMatrix[inTheMoneyBool])/sqrt(dt))/timeSteps) * SBarn)
    #T2 = np.hstack((T,thetaN.reshape((nRows,1))))
    #print(np.sum(T2)/paths)

    return callValue,deltaValue,rho,vegaValue

t0 = time()
maturity = 1
paths = 250000
timeSteps = 365
dt = maturity / timeSteps
sigma = 0.2
dividend = (0.03)
initialStockPrice = 100.
strike = 100
rate = 0.06
answer = MC(initialStockPrice,strike,rate,dividend,maturity,sigma,paths,timeSteps)
print("call value = " + str(answer[0]))
```

```
        print("delta = " + str(answer[1]))
        print("rho = " + str(answer[2]))
        print("vega = " + str(answer[3]))
        tnp1 = time() - t0
        print("Elapsed time = " + str(tnp1))

        #250,000 paths and 365 time steps
        #CPU times: user 0 ns, sys: 0 ns, total: 0 ns
        #Wall time: 9.54 µs
        #call value = 9.18927287808
        #delta = 0.571032536288
        #rho = 48.308463572
        #vega = 37.4512229808
        #Elapsed time = 35.474997997283936
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 9.54 µs
call value = 9.18927287808
delta = 0.571032536288
rho = 48.308463572
vega = 37.4512229808
Elapsed time = 35.474997997283936
```

6 Finite Difference Greeks

```
In [3]: %time
        '''
        ################################################################################
        #
        #              Simple Monte Carlo looping with Greeks by
        #              Finite Difference
        #              Version 1.0.0
        #              David Dallaire, February 11, 2016
        #
        #              Regular inputs for vanilla option:
        #                    Price, Strike, Rate, Dividend Yield, Volatility,
        #                    Maturity
        ################################################################################
        #
        #              OUTPUTS
        #              delta, rho, vega
        ################################################################################
        Created on Jan 18, 2016
        http://pythonhosted.org/ad/
        @author: ddu
        '''
        import numpy as np
        import math as math
        import sys

        def MC(S,K,r,d,maturity,sig,paths,timeSteps):
            lnS = math.log(S)
            np.random.seed(3000)
            dt = maturity/timeSteps
```

```
    nudt = (r - d - 0.5*sig**2) * dt
    k = nudt + np.random.randn(paths,timeSteps) * sig * math.sqrt(dt)
    colS = np.zeros((paths,1))
    colS[:]=lnS
    k = np.hstack((colS,k))
    expL = np.exp(np.cumsum(k,axis=1))
    #keep columns where payoff is positive
    expLCall = expL[expL[:,timeSteps]>K]
    callValue = np.sum(expLCall[:,timeSteps]-K)/paths  * math.exp(-r*maturity)
    return callValue

t0 = time()
maturity = 1.0
paths = 250000
timeSteps = 365
dt = maturity / timeSteps
mu = (0.06)
sigma = 0.2
div = (0.03)
S = 100.
K = 100
sig = (0.2)
r = 0.06
ans = MC(S,K,r,div,maturity,sig,paths,timeSteps)

print("call value = " + str(ans))

#Delta
tweak = 0.01
cPlusTweak = MC(S + tweak,K,r,div,maturity,sig,paths,timeSteps)
cMinusTweak = MC(S - tweak,K,r,div,maturity,sig,paths,timeSteps)
#print(cPlusTweak)
#print(cMinusTweak)
delta = (cPlusTweak - cMinusTweak)/(2*tweak)
print("delta = " + str(delta))

#vega
sigTweak = 0.0001
cPlusSigTweak = MC(S,K,r,div,maturity,sig+sigTweak,paths,timeSteps)
cMinusSigTweak = MC(S,K,r,div,maturity,sig-sigTweak,paths,timeSteps)
vega = (cPlusSigTweak - cMinusSigTweak)/(2*sigTweak)
print("vega = " + str(vega))

#rho
rhoTweak = 0.0001
cPlusRhoTweak = MC(S,K,r+rhoTweak,div,maturity,sig,paths,timeSteps)
cMinusRhoTweak = MC(S,K,r-rhoTweak,div,maturity,sig,paths,timeSteps)
rho = (cPlusRhoTweak - cMinusRhoTweak)/(2*rhoTweak)
print("rho " + str(rho))
tnp1 = time() - t0
print("Elapsed time = " + str(tnp1))

#250,000 paths and 365 time steps
#CPU times: user 0 ns, sys: 0 ns, total: 0 ns
```

```
        #Wall time: 215 µs
        #call value = 9.18927287808
        #delta = 0.582553264756
        #vega = 37.8217644544
        #rho 49.0660537358
        #Elapsed time = 71.51109099388123
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 215 µs
call value = 9.18927287808
delta = 0.582553264756
vega = 37.8217644544
rho 49.0660537358
Elapsed time = 71.51109099388123
```

7 Greeks Using AD Package

The AD package as of writing does not work with NumPy vectors. Thus, the application of the AD package using the slower non-vectorized MC below. Based on the results and configuration here, the AD is rather slow in this case.

```
In [2]: %time
        '''
        ################################################################################
        #
        #               Simple Monte Carlo looping with package AD
        #               Version 1.0.0
        #               David Dallaire, February 11, 2016
        #
        #               Regular inputs for vanilla option:
        #                     Price, Strike, Rate, Dividend Yield, Volatility,
        #                     Maturity
        ################################################################################
        #
        #               OUTPUTS
        #               delta, rho, vega
        ################################################################################
        Created on Jan 18, 2016
        '''
        import numpy as np
        import sys
        #import math as math
        from time import time
        from ad import adnumber
        from ad.admath import *  # sin(), etc.
        from ad import jacobian

        np.random.seed(3000)
        to = time()

        #params
        maturity=1; S=adnumber(100); strike=100; sigma=adnumber(0.2); rate=adnumber(.06); dividend = 0.
        timeSteps = 365;simulations=10000
```

```
        dt = maturity/timeSteps
        sqrtDt = sqrt(dt)
        nudt =(rate-dividend-0.5*sigma**2)*dt #drift
        sigmasdt = sigma* sqrt(dt) #dW
        lnSo = ln(S)
        sumCallValue = 0
        sumDelta = 0
        sumR = 0
        sumSig = 0
        D = exp(-rate*maturity)

        for j in range(1,simulations+1) :
            lnSt = lnSo
            randVector =  np.random.randn(timeSteps)
            for i in range(1,timeSteps+1):
                lnSt = lnSt + nudt + sigmasdt * float(randVector[i-1])

            endStockValue = exp(lnSt)
            callValue = max(0, endStockValue - strike)
            sumCallValue += callValue

        call_value = sumCallValue/simulations * D
        print("call value = " + str(call_value))
        print("delta = " + str(call_value.d(S)))
        print("rho = " + str(call_value.d(rate)))
        print("vega = " + str(call_value.d(sigma)))
        timeDiff = time() - to
        print("timeDiff = " + str(timeDiff))

        #10,000 paths and 365 time steps
        #CPU times: user 0 ns, sys: 0 ns, total: 0 ns
        #Wall time: 5.48 µs
        #call value = ad(9.345810767043462)
        #delta = 0.5856242529215658
        #rho = 49.216614525112774
        #vega = 38.68742808702166
        #timeDiff = 1179.4010179042816
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 5.48 µs
call value = ad(9.345810767043462)
delta = 0.5856242529215658
rho = 49.216614525112774
vega = 38.68742808702166
timeDiff = 1179.4010179042816
```

8 Conclusion

The vector version of the MC Greeks by the hand adjoint is the fastest as expected, taking 35 seconds versus the 72 seconds for the vector version of MC Greeks by finite difference. The comparison of the AD package to non-vector version of hand adjoint shows that AD is very slow; however, the AD might not be setup properly or was not optimized for this type of MC calculation. Plus, the AD is not available with NumPy vectors. The objective here was to demonstrate the mechanics of the adjoint method with a basic

example. The speed comparisons are less important as the example is trivial one; that is, the speed up is more pronounced as the number of sensitivities/Greeks increase.

9 References

- [1] Giles, Mike. Adjoint methods in computational finance.(https://staff.fnwi.uva.nl/p.j.c.spreij/winterschool/slidesGiles
- [2] Homescu, Cristian. Adjoints and automatic (algorithmic) differentiation in computation finance.(http://arxiv.org/pdf/1107.1831.pdf)
- [3] Capriotti, Luca. Fast Greeks by algorithmic differentiation. The Journal of Computational Finance (3-35), Volume 14/3, Spring 2011. (http://luca-capriotti.net/pdfs/Finance/jcf_capriotti_press_web.pdf)
- [4] Capriatti, Luca, and Giles, Mike. Algorithmic Differentiation: Adjoint Greeks Made Easy. March 31, 2011. (http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1801522)